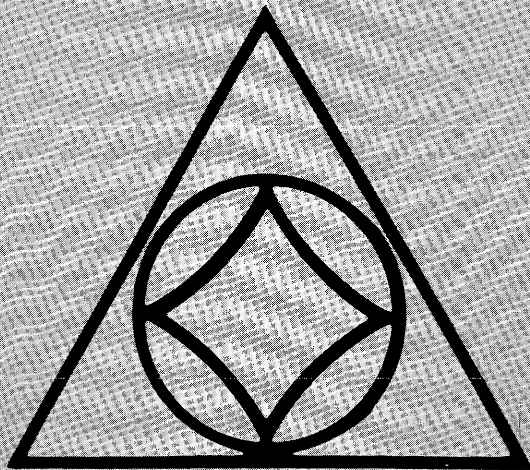


HOWARD LEE MORGAN
Editor and Program Chairman

RUSSELL K. BROWN
Conference Chairman

AFIPS PRESS

1815 NORTH LYNN STREET

ARLINGTON, VIRGINIA 22209

AFIPS

CONFERENCE PROCEEDINGS

1982

NATIONAL COMPUTER CONFERENCE

June 7–10, 1982
Houston, Texas

The ideas and opinions expressed herein are solely those of the authors and are not necessarily representative of or endorsed by the 1982 National Computer Conference or the American Federation of Information Processing Societies, Inc.

Library of Congress Catalog Card Number 80–649583
ISSN 0095-6880
ISBN 0-88283-035-X
AFIPS PRESS
1815 North Lynn Street
Arlington, Virginia 22209

© 1982 by AFIPS Press. Copying is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) reference to the AFIPS Proceedings and notice of copyright are included on the first page. The title and abstract may be used without further permission in computer-based and other information-service systems. Permission to republish other excerpts should be obtained from AFIPS Press.

Printed in the United States of America

Preface

RUSSELL K. BROWN
1982 NCC Chairman

The purpose of the National Computer Conference is to provide an atmosphere in which designers, suppliers, users, managers, educators, and representatives of government and society at large can meet and interact. Discussions of new technical developments, as well as national and international issues and challenges facing the information processing community, are encouraged.

This year's discussions and developments are contained, for the most part, in this anniversary Volume 51 of the Proceedings of the National Computer Conference, completing its first decade as the world's premier computer exposition.

The decision to chair a National Computer Conference may well be one of the more major choices one makes in even a complicated lifetime. Certainly, this choice was compounded by the change in site from New York to Houston, made only thirteen months prior to the Conference date. Perhaps a few words on that move are in order.

In spring 1981 the NCC Committee and Board were faced with a dilemma of some magnitude. The Conference exhibits had grown so large that plans to house them in New York became unrealistic. To have held NCC '82 there would have dictated a requirement to cut back the number of companies exhibiting, the maximum exhibit size, or both. After much arranging by the AFIPS staff, a plan was presented to use New York to its absolute limits. To do this, we would have had to split the show across the convention facility, some number of hotel ballrooms, and a covered pier on the East River. Even then, booth size would have had to be cut and the rather spectacular island concept with which you are familiar in our exhibit areas would have been affected severely.

As a long-time Houstonian, I was well aware of the potential abilities of my city to handle an NCC. In a very short time, we were able to arrange the use of the Astrohall and Arena, reserve 12,000 hotel rooms, and make other arrangements necessary to effect the move.

Naturally there were a few rough edges. Because of the timing, we had to spread out our hotels much more than will be the case when we return for NCC '84. But we feel that, given less than half the normal preparatory time accorded most Conference Steering Committees, you will see few shortcuts or shortcomings.

What you *will* see is a display of 650 companies filling 3,200 booth units for a new NCC record. You will be exposed to a high-quality program, high-quality Professional Development Seminars, four major invited addresses, a special Pioneer Day program, and numerous other attractions that we feel will make this a noteworthy week. It is the intention of the CSC to give you, the attending registrant, all the positive values of a move to our city and make any negatives as invisible as possible.

An example of this is the expenditure of nearly $200,000 for busing to assist you in the various round trips between your hotel and the Conference.

If I may return to our program, possibly I can elicit in you a feeling of satisfaction to match the pride I feel. The program is made permanent by the archival record of the Proceedings. Here we capture for posterity the most current reports on recent achievements and new applications, on advances at the frontiers of computer science and technology.

Dr. Howard Morgan of the Wharton School was buffeted in mid-preparation of this program and these Proceedings by the move. Through all the personnel shuffling and turmoil, he managed to steer a straight course toward a superior presentation.

Howard recognized, early on, that the registrant has only three days, on the average, to assimilate all aspects of an NCC. His first decision was to direct that, with a superior Professional Development Program together with ten football fields of exhibits, the program as defined in the past be intensely screened for shortcomings. His Committee introduced a much finer mesh in their screen than has ever been used before. The number of papers and sessions are down slightly from what you have seen in previous NCCs, but we are confident that their value to you will be high. We will be surprised if you depart early from any of our sessions.

Volunteers, for a Conference of this magnitude, number in the hundreds. They are members of the NCC Sponsoring Societies and the other AFIPS Constituent Societies. To these

groups and their participating members I would like to give my heartiest thanks, particularly in view of the truncated schedules on which we were all operating.

To the NCC Board and Committee, who well knew the danger to NCC '82 if plans were not well organized, my thanks for your confidence and support.

To the AFIPS Headquarters Staff and all the members of our CSC, thank you for your dedication, time, and effort. You have contributed to an ongoing tradition of excellence.

To my wife, who only once asked, "Why?" but a hundred times asked, "How can I help?" you know my thoughts.

And finally, to the nine NCC Chairmen of the past, thank you for your assistance, guidance, and inventiveness. Much of what you created is embodied here.

Introduction

HOWARD LEE MORGAN
1982 NCC Program Chairman

"Advancing Professionalism" is the theme for the 1982 National Computer Conference. It is our belief that these Proceedings represent a contribution to the professionalism of you who are reading them or those who attended the conference. The computing field now incorporates many types of professionals: designers, analysts, programmers, managers, and users of office and personal computing systems. Parts of our program are aimed specifically at each of these types of users. More important, we hope that people will integrate and broaden their knowledge with the help of the wide spectrum of sessions, panels, and papers presented, which cover all major aspects of the computing field as we know it today.

With this theme as the base, the NCC '82 program has been structured into eight major areas. These include the following:

1. Hardware and computer architecture: providing more power and newer structures than those that have been traditional for hardware designers.
2. Software engineering: techniques to aid in the building of correctly working and properly engineered software.
3. Personal computing: included this year for the first time in the main NCC program and undergoing explosive growth in both business and home use.
4. The social and organizational implications of computing: this area indicates how totally computers now impinge on our daily lives.
5. Office Systems: this area addresses the concerns of those involved in the growing office automation environment.
6. Decision support and management issues: to aid those whose job it is to manage computing or to provide services directly to top executives.
7. Language and database processing: two key applications systems tools.
8. Finally, the applications of computing themselves.

As a special feature, the history of computing and Pioneer Day focus on FORTRAN and its early development. We are fortunate to have several key papers in these Proceedings.

We have reduced the number of sessions this year to 86, as opposed to the 105 to 120 of previous years. This has had the favorable effect of permitting us to select and work with higher average quality levels, but some worthwhile paper and session proposals were not able to be included in the conference. We are sure that the panels and paper sessions in the program provide detailed, high-quality presentations in their specific areas. These Proceedings are organized according to the areas of interest, as noted in the conference program. The conference program contains a page number key to these specific papers, for easy reference by attendants. Because space is limited, summaries of the panel discussions are not printed in these Proceedings; but they are available in the conference guide.

The plan and organization of the 1982 NCC program required the concerted, dedicated, and extreme efforts of many individuals: the Program Committee members; the session organizers and leaders; the panelists, presenters, and authors of technical papers; and the referees, who helped us select the papers to be presented in these Proceedings. In addition, the entire NCC committee structure and the staff organizations at AFIPS have played an important role in the smooth operation of the conference. The committee assistants, Françoise Aubert-Santelli and Susan O'Leary, performed far beyond the call of duty. I wish to extend my sincere thanks to all of these individuals and most especially to our Program Committee. It is through their efforts that the NCC '82 program and these Proceedings have come alive. It is our sincere hope that your attendance at the program will prove a fruitful and enjoyable activity to those of you who were fortunate enough to come and that these 1982 NCC Proceedings will join their predecessors as a useful reference for many years.

CONTENTS

Preface iii
Russell K. Brown

Introduction v
Howard Lee Morgan

HARDWARE/COMPUTER ARCHITECTURE

Firmware quality assurance 3
Helmut K. Berg, Prakash Rao, and Bruce D. Shriver

The 5.25-inch fixed/removable disk drive 11
Don Minami

Practical CMOS microprocessor systems 19
Bill Huston

The MC68000 family and distributed processing 29
John F. Stockton

Using operational standards to enhance system performance 37
David R. Vincent

Distributed processing with the Z8000 family 53
Richard Mateosian and Janak Pathak

Distributed processing with iAPX 186 microprocessor systems 59
Tony Zingale

High-performance, high-capacity single-chip microcomputers 67
Ed Peatrowsky

Expanded single-chip principles in practical applications 73
Randy M. Dumse

Making the most of VLSI in microcomputers 81
Jerry L. Corbin

Single-chip microcomputers can be easy to program 85
Bill Huston

Speak software and carry a strip chip 95
Michael Shapiro

A distributed operating system for a powerful system with dynamic architecture 103
Steven I. Kartashev and Svetlana P. Kartashev

Software testing techniques for universal building blocks of multimicrosystems 117
M. Annaratone and M. G. Sami

A methodology for the development of special-purpose function architectures 125
Raymond A. Liuzzi and P. Bruce Berra

Applications of SIMD computers in signal processing 135
Laxmi N. Bhuyan and Dharma P. Agrawal

A list-processing-oriented data flow machine architecture 143
Makoto Amamiya, Ryuzo Hasegawa, Osamu Nakamura, and Hirohide Mikami

Lookahead networds 153
G. Jack Lipovski, Ambuj Goyal, and Miroslaw Malek

Reconfigurable multicomputer networks for very fast real-time applications 167
Carl Davis, Svetlana P. Kartashev, and Steven I. Kartashev

MPP: a supersystem for satellite image processing 185
Kenneth E. Batcher

Optimal design of a distributed supersystem 193
David F. Palmer, James P. Ignizio, and Catherine M. Murphy

Distributed processing with the NS16000 family 199
Leslie Kohn

SOFTWARE ENGINEERING

Exploiting parallelism for the performance enhancement of non-numeric applications 207
David J. Dewitt and Dina Friedland

Performance engineering of software systems: a case study 217
C. U. Smith and J. C. Browne

A systolic processor for signal processing 225
G. A. Frank, E. M. Greenawalt, and A. V. Kulkarni

Parallel-processing a large scientific problem 233
Robert Hiromoto

Design of software for distributed/multiprocessor systems 239
Terrence R. McKelvey and Dharma P. Agrawal

The use of performance models in systematic design 251
K. M. Chandy, J. Misra, R. Berry, and D. Neuse

Performance modeling in the design process 257
William Alexander and Richard Brice

MEDOC: A methodology for designing and evaluating large-scale real-time systems 263
Eric Le Mer

The research queueing package: Past, present, and future 273
Charles H. Sauer, Edward A. MacNair, and James F. Kurose

Audience identification for end user documentation 281
Janis G. Raymond

Computer-aided documentation 287
Saul Rosenberg

The development of software engineers: a view from a user 293
Walter P. Warner and Richard E. Nance

An industrial software engineering methodology supported by an automated environment 301
Michael S. Deutsch

An approach to the definition and implementation of a software development environment 309
James F. Elwell

A JOVIAL programming support environment 319
Edith M. McMahon

The impact of Ada on software engineering 327
Kenneth L. Bowles

The importance of Ada programming support environments 333
Thomas A. Standish

Challenges and requirements for new application generators 341
Alfonso F. Cardenas and William P. Grafton

Program generators and their effect on programmer productivity 351
Richard L. Roth

Application generators at IBM 359
Aaron M. Goodman

Application generators: a case study 363
James H. Waldrop

Requirements definition and its interface to the SARA design methodology for computer-based systems 369
James W. Winchester and Gerald Estrin

The role of requirements analysis in the system life cycle 381
Yuzo Yamamoto, Richard V. Morris, Christopher Hartsough, and E. David Callender

Application generators: an introduction 389
Jerrold M. Grochow

Software product quality assurance 393
John R. Ryan

A quality assurance program for software maintenance 399
John W. Center

The independent role: verification and validation, and compliance testing 409
Barbara J. Taute

Quality assurance in a large commercial data processing installation 415
C. W. Lybrook

PERSONAL COMPUTING

Data server design issues 429
Fred Maryanski

SOCIAL AND ORGANIZATIONAL IMPLICATIONS

Acceptance criteria for computer security 441
William Neugent

Private sector needs for trusted/secure computer systems 449
Rein Turn

Impacts of information system vulnerabilities on society 461
Lance J. Hoffman

Uniform help facilities for a cooperative user interface 469
Philip J. Hayes

Natural-language help in the Consul system 475
William Mark

Programs as data for their help systems 481
Elaine A. Rich

The implementation of a cryptography-based secure office system 487
Christian Mueller-Schloer and Neal R. Wagner

Criteria for a standard command language based on data abstraction 493
David Beech

Integration of bottom-up and top-down contextual knowledge in text error correction 501
Sargur N. Srihari, Jonathan J. Hull, and Ramesh Choudhari

Dialogue: Providing total terminal independence 509
David Vaskevitch

The Star user interface: an overview 515
David Canfield Smith, Charles Irby, Ralph Kimball, and Eric Harslem

MFS: a modular text formatting system 529
James D. Mooney

MANAGEMENT ISSUES/DECISION SCIENCE SUPPORT SYSTEMS

Complex business systems: a strategy for success 539
Naomi Lee Bloom

The role of the user at Standard Oil Company (Indiana) in the development of large-scale business systems 549
James E. Jackson

The role of data center personnel in the development of a large-scale business system 555
David A. Cox

What life? What cycle? 561
Nicholas Zvegintzov

LANGUAGE AND DATABASE PROCESSING

Data model processing 571
Matthew B. Koll, W. Terry Hardgrave, and Sandra B. Salazar

Automatic database system conversion: schema revision, data translation, and source-to-source program transformation 579
Ben Shneiderman and Glenn Thomas

Fair timestamp allocation in distributed systems 589
Said K. Rahimi and William R. Franta

Data abstraction for Pascal programmers 595
Viswanathan Santhanam and John R. Potochnik

SPIRIT-III: an advanced relational database machine introducing a novel data-staging architecture with Tuple Stream Filters to preprocess relational algebra 605
Noriyuki Kamibayashi and Kazuo Seo

Data language requirements of database machines 617
Dawei Luo, Daozhong Xia, and S. Bing Yao

Performance analysis of database join processors 627
Fu Tong and S. Bing Yao

Evaluating database management systems 639
Edward Davidson

Performance study of a dual CDC Cyber 170/750 system 649
M. Seetha Lakshmi and Tom W. Keller

Computational lexicology: A research program 657
Robert A. Amsler

Use of *Webster's Seventh Collegiate Dictionary* to construct a master hyphenation list 665
James L. Peterson

Models, languages, and heuristics for distributed computing 671
Robert E. Filman and Daniel P. Friedman

Weakest environment of communicating processes 679
Zhou Chaochen

Adaptive structuring of distributed databases 691
K. Dan Levin

Distributed scheduling of resources on interconnection networks 697
Benjamin W. Wah and Anthony Hicks

APPLICATIONS OF COMPUTING

A microcomputer system for color video picture processing 713
Yoshikuni Okawa

The importance and futility of device independence in computer graphics 719
Anders Vinberg

Optimal three-dimensional flight control of a supersonic fighter 727
Ching-Fang Lin and Khai Li Hsu

Structured D-chart: A diagrammatic methodology in structured programming 735
C. Jinshong Hwang

Planning for software tool implementation: experience with Schemacode 749
Pierre N. Robillard and Réjean Plamondon

Distributed processing of problem-solving applications for farmers 759
Robert Gammill and Lynn Thorp

RIPS net: The impact of an optical communication network 767
Koji Yada, Masanori Honda, and Seiji Fujino

A coherent scheme to support location-independent references in internetwork environment 775
Ray Cheng and J. W. S. Liu

Issues and methods for practical distributed data processing applications—I 785
Maurice Blackman and Hugh Ryan

Issues and methods for practical distributed data processing applications—II 793
Maurice Blackman and Hugh Ryan

PIONEER DAY

A technological review of the FORTRAN I compiler 805
F. E. Allen

Computing prior to FORTRAN 811
R. W. Bemer

History of FORTRAN standardization 817
Martin N. Greenfield

DYSTAL: Nonnumeric applications of FORTRAN 825
James M. Sakoda

HARDWARE/COMPUTER ARCHITECTURE

Firmware quality assurance

by HELMUT K. BERG and PRAKASH RAO
Honeywell Corporate Computer Sciences Center
Bloomington, Minnesota
and
BRUCE D. SHRIVER
University of Southwestern Louisiana
Lafayette, Louisiana

ABSTRACT

The paper reviews problems, solutions, and trends in the area of firmware quality assurance. Firmware quality assurance is considered to be the certification of the fact that a firmware system meets its requirements with respect to functional correctness as well as performance, operational, and implementational properties. The emphasis of the paper is on formal correctness proofs, firmware testing, and the automatic synthesis of microcode and associated hardware structures. Firmware specifications, high-level microprogramming languages, and automated support tools are discussed as they relate to these areas. The impact of advances and trends in very large-scale integration (VLSI) on the techniques and tools for firmware quality assurance is reviewed. The observation is made that valuable results have been obtained in the areas of firmware correctness proofs and firmware testing. However, further improvements are needed to cope with the complexity of VLSI. An alternative that may overcome the limitations of these two approaches is automated synthesis of firmware and hardware and design for testability.

A. INTRODUCTION

For products of any kind, assurance needs to be gained that they meet their product requirements before they are dedicated to serve their intended purpose. This need applies to end-user or consumer products as well as to the individual components to be integrated into such products. The product requirements may be stated in a variety of forms, including assessments of market needs, functional product specifications, and specifications of nonfunctional product attributes. The form of these requirement statements changes as the development of a product proceeds from the marketing product definition through the product design and implementation to the use and maintenance of the product. This process is referred to as the product life cycle, and it comprises various stages in each of which the ability of the product to meet its requirements is established. These certification steps may be summarized under the term quality assurance.

Ideally, firmware is developed and deployed in a life cycle that includes the following steps. The development begins with a step called requirements engineering. Given the purpose of the system, this step identifies the functional requirements and attributes of the system. Nonprocedural design formalizes these requirements and attributes in the form of functional and property specifications. The procedural design uses the specifications to produce "blueprints" for the implementation. The implementation step embodies the blueprints in system modules and microprograms. The integration step combines and tests the modules and microprograms so that the assemblage results in an operational firmware system. In the installation step, the firmware system is integrated into the overall system and submitted to operation. Maintenance is concerned with corrections and extensions to the operational firmware.

Every step in the firmware life cycle is associated with an appropriate validation step. For example, it needs to be demonstrated that the requirements and attributes conform to the statement of the purpose. The correspondence between requirements and specifications needs to be demonstrated. Obviously, it should be verified that each step in the development process was conducted correctly.

The *quality of the firmware* refers not only to the functional correctness of the integrated microprograms, but also to the performance, operational, and implementational properties of the system. Among the properties of a firmware system, we may find execution time, object microprogram size, reliability, robustness, and viability. Hence, *firmware quality assurance* may be defined as the certification of the fact that a firmware system meets its requirements in an optimal way. In this definition, optimality is not assumed to be an absolute measure. In fact, the optimization of some of the firmware properties listed above has been shown to be NP-hard.[1] Furthermore, absolute functional correctness cannot be established by testing, and formal correctness proof methods, which theoretically have the ability to demonstrate the absence of errors, have generally not reached the point of being rigorously applicable in firmware development environments.[2]

The need for firmware quality assurance has been discussed widely in the literature. The necessity for the verification of functional correctness[9] stems from two intrinsic characteristics of firmware:

- microprograms control all native hardware resources. Thus, microprogram errors result in an erroneous virtual machine, and
- microprograms very often reside in read-only storage media. Thus, modifications and error corrections can be both difficult and costly.

The need for the optimization of execution speed and object microprogram size[3] is dictated by the desire to

- realize faster and functionally more powerful machines, with available technology, and
- obtain the extensive benefits of high-level microprogramming languages in the firmware development process.

The need for firmware quality assurance is intensified by technological advances, most importantly by very-large-scale integration (VLSI).[4] Hardware performing specialized functions is being replaced by regular arrays of logic and memory. The functionality of the firmware is changing from conventional instruction set emulators to more extensive and powerful instruction sets, diagnostic programs, interpreters for high-level languages, and operating system functions. In addition, the integration of microprogrammed control schemes into VLSI places more stringent requirements on tools and techniques for firmware quality assurance, which cannot be met by traditional microprogramming aids.

In summary, the major concerns in firmware quality assurance are

- the specification of functions and properties of firmware systems,
- the realization of correct and optimal microcode, together with appropriate hardware structures, and
- automated tools that aid the designer in exploring alternative designs and in keeping track of design and implementation details.

It is not possible in this paper to treat entire methodologies and engineering environments for firmware development and

quality assurance. We restrict our attention to areas of formal firmware correctness proofs, firmware testing, and the automatic synthesis of microcode and associated hardware structures. Firmware specifications, high-level microprogramming languages, and automated support tools are discussed only as they relate to these areas.

B. VERIFICATION OF FIRMWARE

Firmware verification through formal correctness proofs is an area of firmware engineering that received considerable attention over the last decade.[6] Although several approaches to firmware correctness proofs have been developed and demonstrated, problems remain at all levels. Among these problems are the development of appropriate theoretical foundations, the definition of design disciplines that support correctness proofs, tool support for correctness proofs, and the education of users regarding formal techniques. Despite these problems and the controversies surrounding formal correctness proofs in general,[7] some workers report that no matter how expensive it is to find errors by firmware verification it is still a magnitude cheaper than finding errors when a product is shipped.[8,9] This section reviews firmware verification by summarizing some efforts in the area, discussing their contribution to quality assurance, and indicating possibilities and limitations in their use.

B.1 Approaches to Firmware Verification

Approaches to firmware verification generally draw from results obtained in software verification.[10] Given the current state of the art, correctness proof processes are inherently complex, and cannot be fully automated. Human interaction with verification systems is required to suggest proof goals, to partition proofs, and to interpret results obtained from the proof system to direct the continuation of the proof process.

The STRUM system[11] is an advanced verification system that is based on Floyd's inductive assertion technique[12] and uses a Pascal-like high-level microprogramming language. The automated proof process is integrated into the translation process.

Another successfully applied verification system is the IBM Microprogram Certification System (MCS).[13] This system is based on Milner's technique of the simulation between programs[21] and the symbolic execution of programs.[22] A similar approach is being pursued at the Information Science Institute of the University of Southern California.[23] The MCS approach considers both the description of the host system on which the microcode is to run and of the target system that is being emulated by the microcode. The verification system accepts the host microcode and its specification, including proof commands, to establish the equivalence of both firmware definitions by symbolic execution and the proof of simulation relations.

A third approach that has received considerable attention is the microprogramming language schema S*.[24] This approach defines an axiomatic basis for microprogramming similar to Hoare's axiomatic definition of programming languages.[25] The axiomatic basis is a set of schemas that define the semantics of a Pascal-like microprogramming language. These schemas constitute the axioms and inference rules of a deductive system in which formal correctness proofs can be carried out using the defined logical inferences. For each particular machine, the schematic semantic definitions can be instantiated to capture the machine-dependencies influencing the execution of the firmware on that machine. Thus, after the instantiation, the microprogrammer works with a machine-dependent, but high-level, axiomatic proof system.

For a survey of other approaches to firmware verification, the reader is referred to Davidson and Shriver (1980).[6]

B.2 Status of Firmware Verification

Successful application of firmware correctness proofs has established firmware verification as a viable approach to firmware quality assurance. Two major observations need to be made, however. First, correctness proofs are complex in nature; thus the verification process needs to be incorporated into an overall firmware engineering discipline that is supportive of correctness proofs; additionally, the proof process itself needs to be supported by automated tools. Second, quality assurance is not restricted to the functional correctness of firmware, but is also concerned with execution time and memory efficiency of the executable code as well as with the reliability of firmware systems in general. Thus, verification approaches need to be developed based on high-level microprogramming languages that facilitate abstract representations of firmware systems and allow code optimization. It is imperative that the user be able to understand what the system is supposed to do.

The systems described above partially satisfy these requirements. The STRUM system has been applied to the development of the emulator for the Hewlett-Packard HP-2115.[11] Besides the verification of the microcode written in the high-level language, the resulting code could be optimized to match an independently generated, hand-optimized version of the same microcode. The success of the microcode verification system used for the Fault-Tolerant Spaceborne Computer (FTSC) is partially due to the guiding principle of that project.[23] It concentrates on the practical side of the verification problem, solving the theoretical problems as they arise. Emphasis is placed on ways of making the user understand what the system is doing and writing the microcode with the subsequent verification in mind. The FTSC project is explicitly seeking an approach to a disciplined firmware design and development process. Additionally, software engineering approaches can be enhanced further for the production of more reliable microcode. For example, techniques such as code inspections, walk-throughs, or step-wise refinement may be integrated into the firmware engineering discipline.

In summary, languages and quality assurance techniques are needed that help the microprogrammer with machine dependent problems such as microparallelism, microcode optimization, and the variability in computer micro-architectures, on the one hand, and support abstract representations of firmware systems, on the other hand. Progress made toward these seemingly conflicting goals is best reflected by high-level microprogramming languages such as MPL,[26] STRUM,[11]

EMPL,[27] VMPL,[28] and S*.[24] Most of these languages effectively attack the problem of code optimization,[3] but problems remain in the area of verification.

B.3 Summary

Limitations in firmware verification techniques result from several fundamental weaknesses of the proposed approaches. The weaknesses include the inadequate specification of the timing characteristics of the control flow in semantic definitions of microinstructions. Furthermore, the deductive systems (i.e., the set of available logical inferences) for carrying out firmware correctness proofs are not related closely enough to the characteristics of the underlying hardware. These weaknesses limit the practicality of firmware correctness proofs even for moderately sized microprograms. The inclusion of parallelism, synchronization, and microinstruction execution subcycles will considerably increase the complexity of the correctness proof. Powerful interactive capabilities may mitigate this deficiency of verification systems. Correctness proof techniques need to acquire conceptual foundations that bridge the gap between high-level machine-independent firmware representations and specific microprogram running environments, in order to cope with the firmware complexity anticipated for VLSI.

Several approaches to the solution of this problem have been proposed. These approaches require that the firmware quality assurance process be carried out in a high-level-language environment in which mappings to machine specific environments can be automated. The approaches include axiomatization of the running environment,[24,30] explicit descriptions of the microprogram running environment,[23,31] and machine virtualization.[28,32] Although most of these approaches still need to be demonstrated for problems of the scope anticipated for VLSI, it is certain that they will be viable alternatives to firmware verification only if they are supported by sophisticated tools that automate design, coding and verification.

C. TESTING OF FIRMWARE

Firmware testing is the assurance of firmware functionality for a specified set of input values. By definition, then, firmware testing does not necessarily assure functional correctness for all legitimate input values and is not as strong an argument about the correctness of microprograms as that provided by formal correctness proofs. Under the restriction imposed by the absence of sufficiently versatile verification methods, firmware testing is one of the alternatives available for assuring the quality of firmware.[5,2]

Firmware testing has traditionally been viewed as an extension to the testing of software.[2] With the impact of VLSI, it is no longer possible to separate the functionality of the underlying hardware from the microprograms that control it.[4] The increasing degree of integration of microprograms with their hardware environment also requires a unified approach to the design of the firmware and its supporting hardware. This design approach, called "design for testability,"[33] incorporates the important concept of testability directly into the synthesis procedure and thereby enhances the assurance of firmware system quality. The discussion of design for testability is deferred to the next section.

C.1 Firmware Testing

Microprogram testing has been addressed by Berg.[2] Formal techniques of firmware testing and test data selection have been specified. Three levels of microprogram testing have been identified. These are

- Tests at the microprogram level that consider complete microprograms by either analyzing their code or investigating the machine states resulting from their execution.
- Tests at the microinstruction level that consider single microinstructions by either analyzing the assignment of micro-operations to them or investigating the machine states resulting from their execution.
- Tests at the micro-operation level that consider individual micro-operations by monitoring their execution.

An error detected at the microprogram level may be caused by any number of faulty microinstructions or micro-operations in the microprogram. The manifestation of such errors is defined by the identification of a set of faulty microinstructions and micro-operations. To identify faulty microinstructions or micro-operations in an erroneous microprogram, tests at the microinstruction level may be necessary. An error detected at the microinstruction level is located if the set of faulty micro-operations in the microinstruction can be identified.

The selection of tests and test data for firmware testing at each of these three levels is based on the methods described by Goodenough and Gerhart.[5] The basis for correct function is the program specification. The method distinguishes between test data and test predicates.

C.2 Hardware Testing

Hardware testing has traditionally been viewed in isolation, despite increasing trends towards the implementation of systems as microprogrammed control structures. With the integration of hardware and firmware in VLSI a major problem is the diminishing observability and controllability of the hardware due to pin limitations. Hardware testing will therefore entirely devolve on the micro-operations that it supports. As a consequence, hardware must be tested at the register transfer level, where a description of the system is specified in terms of the hardware resources and their interconnection. This requirement necessitates that the degree of encoding of microinstructions be restricted to keep testability as an objective during the design process.

A significant step toward the testing of hardware through the microprogramming level and the implementation of serviceability features was taken during the design of the IBM System/360 as reported in Carter et al. (1964).[14]

Hardware test strategies for microprogrammed units use a partitioning of the unit into an operative part and a control part.[18] The operative part consists of the hardware resources

and the control part of the algorithms for their operation and sequencing. A methodology for generating an internal microprogram to test a microprogrammed unit is described by Ciaramella.[19] The dynamic testing of control units has been reported by Robach and Saucier.[38]

Identification of the control and operative parts of the system is performed from a behavioral description of the system hardware function. This description can be given using a high-level design language such as a register-transfer language (RTL) or a multiple-level design language such as LALSD—Language for Automated Logic and System Design[16]—or SIMPL (Simple Identity Microprogramming Language).[17] Techniques for generating tests from such behavioral descriptions are mentioned by Su and Hsieh[20] and Levendel and Menon.[15] The area of behavioral-level testing of digital systems is still in a period of evolution. This problem is further complicated by the fact that hardware access is limited to the microprogramming level. Techniques for the development of hardware support functions needed for firmware system quality assurance still need to be formalized. The evolution of abstract fault models at the behavioral level will be of great assistance in the development of test-generation algorithms for testing microprogrammed units. Developments in this direction have been started in the area of microprocessor testing.[37]

C.3 Summary

The problem of firmware testing can be divided into two subtasks: the testing of microprograms and the testing of the underlying hardware. The problem of microprogram testing is addressed by borrowing concepts from software testing. Approaches to hardware testing using a high-level description of the hardware resources and their interconnection is still in its infancy. This problem will be increasingly aggravated as more and more firmware systems are implemeted in VLSI.

D. AUTOMATED SYNTHESIS AND DESIGN FOR TESTABILITY

In this section, we take a look at the future growth of firmware and microprogrammed systems and highlight issues of testability that arise owing to the increased complexity and the reduced access to the hardware as dictated by the restricted pin count.

D.1 Characteristics of the Design Environment

The future of microprogramming will be greatly affected by the characteristics of VLSI design. With the cost advantage and support of design tools for fast turnaround, single-chip designs in VLSI will become increasingly common. Complex structures such as parallel architectures will be designed routinely using sophisticated design aids for hardware and microcode.[4] Trends in the development of high level microprogramming languages as described in Section B contribute to this development. The microprogram verification problem has therefore been brought closer to a solution in terms of facilities for looking at analogs in the software field.

The migration of microprogrammed control units of very high complexity into VLSI brings in all the problems that VLSI designers of digital systems have been facing. The increase in complexity, the choice of design styles, architectures, and implementation technologies provide a design environment in which a number of tradeoffs have to be made between several objectives. The most important of these objectives are

- to minimize cost by minimizing silicon area and pin count,
- to maximize performance in terms of speed,
- to increase chip functionality by providing more powerful instructions,
- to increase chip fault tolerance,
- to provide an enhanced user interface,
- to achieve reasonably fast design turnaround time and minimize design costs, and
- to minimize power consumption.

Usually, a tradeoff is made between economy of silicon area and performance, fault tolerance, ease of use, and design turnaround time. This often implies that fault tolerance has to be traded off for real estate on silicon. Thus, there is a need for building fault tolerance into the design itself and to minimize the overhead that caused the tradeoff for real estate. Specific design rules that aid the testability of VLSI must be developed.

With the increase in complexity and with the design problem constrained to be an optimization of multiple objectives, designers of the future will have no other alternative but to employ automated tools. To be viable, these tools must aid in the design of the microcode as well as in the synthesis of the hardware implementing the design specification. The need for quality assurance, therefore, dictates the requirement that these tools have testability as an objective.

Research in microprogram design aids has been classified into three classes:[4]

- microcode verification
- microcode generation from a high level language
- synthesis of microcode and microcontroller hardware from high-level specifications.

These research areas have been dealt with in the literature.[11,34,35,36]

D.2 Testability and Automated Hardware/Firmware Synthesis

Automated synthesis has been attempted with varying degrees of success. The MIMOLA system[36] provides interaction between user and system. It generates microcode based on the input of user-declared data paths and hardware resources. Hardware measures are provided to aid in monitoring the efficiency of the design and to identify critical paths. The designer interactively varies hardware restrictions to satisfy the performance and cost requirements. There is no design for testability built into the generation procedure for microcode.

The microcode generated has a degree of encoding restricted to function-select lines and multiplexers.

The problem of providing adequate testability may be addressed at two levels by

- incorporating design for testability at the structural or implementation level of description when the logic is being committed; or by
- incorporating design for testability techniques at the behavioral or specification level.

Traditional methods of design for testability have taken the first approach. At the implementation level, additional hardware is provided to enable the input of test patterns and the breaking of feed-back loops. The latter reduces the testing problem to the more tractable task of testing combinatorial circuits. An extensive survey of design for testability methods of this type is reported in the literature.[33] In this approach, design rules are formulated with design edicts laid down to ensure that access for testability is provided.

The incorporation of design for testability at the behavioral or specification level is relatively new and results of a substantial nature are yet to be reported. In this approach, testability is incorporated into the synthesis procedure as an objective. The design of the micro-operation and microinstruction structures and the generation of microprograms is conducted with the testability of the registers and functional units in mind. Inaccessible registers are not permitted and the length limitation on the longest checking sequence needed to test functionality in functional blocks and registers is one of the constraints in the synthesis procedure.

D.3 Summary

We conclude that there is a future trend toward single-chip implementation of firmware systems bringing with it the use of automated-synthesis tools and microprogram-design aids. There is a need to build testability into the synthesis procedure at a high level in the design process. There is considerable need for further research in this area. Until substantial results have been obtained designers will continue to use the conventional techniques of utilizing additional hardware overhead to provide adequate testability.

REFERENCES

1. Robertson, E. L.. "Microcode Bit Optimization Is NP-Hard", *SIGMICRO Newsletter,* Vol. 8, 1977, pp. 40–43.
2. Berg, H. K. "Firmware Testing and Test Data Selection", *Proceedings of the 1981 National Computer Conference,* Vol. 50 AFIPS Press, Arlington, Va., 1981, pp. 75–80.
3. Landskov, D.; S. Davidson, B. Shriver, P. W. Mallett. "Local Microcode Compaction Techniques," *Computing Surveys,* Vol. 12, No. 3, 1980, pp. 261–294.
4. Parker, A. C., W. T. Wilner. "Microprogramming—The Challenge of VLSI," *Proceedings of the 1981 National Computer Conference,* Vol. 50, AFIPS Press, Arlington, Va., 1981, pp. 63–68.
5. Goodenough, J. B., S. L. Gerhart. "Toward a Theory of Test Data Selection" *IEEE Transactions on Software Engineering,* Vol. 1, No. 2, 1975, pp. 20–37.
6. Davidson, S., B. D. Shriver. "Firmware Engineering: An Extensive Update," in *Firmware, Microprogramming and Restructurable Hardware.* North-Holland Publ. Co., New York, 1980, pp. 1–40.
7. DeMillo, R. A., R. J. Lipton, and A. J. Perlis. "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM,* Vol. 22, No. 5., 1979, pp. 271–280.
8. Husson, S. S. (panel chairman). "Microcode Verification—Summary of the Panel Discussion," in *Firmware, Microprogramming and Restructurable Hardware,* North-Holland Publ. Col., New York, 1980, pp. 105.
9. Carter, W. C., "Microcode Verification," Presentation in the Session on Microprogramming—The Challenge of the 1980's, 1981 National Computer Conference, Chicago, Il.
10. Berg, H. K., W. E. Boebert, W. R. Franta, and T. G. Moher. *Formal Methods of Program Verification and Specification,* Prentice-Hall, Englewood Cliffs, N.J., 1982.
11. Patterson, D. A. "STRUM: Structured Microprogramming System for Correct Firmware," *IEEE Transactions on Computers,* Vol. 25, No. 10, 1976, pp. 974–986.
12. Floyd, R. W. "Assigning Meaning to Programs," *Proceedings of Symposia in Applied Mathematics,* American Mathematical Society, Vol. 19, 1967, pp. 19–32.
13. Joyner, W. H., W. C. Carter, and G. B. Leeman. "Automated Proofs of Microprogram Correctness," *SIGMICRO Newsletter,* Vol. 7, No. 3, 1976, pp. 51–55.
14. Carter, W. C., H. C. Montgomery, R. J. Preiss, and H. J. Reinheimer. "Design of Serviceability Feature for the IBM System/360," *IBM Journal,* 1964, pp. 115–126.
15. Levendel, Y. H., and P. R. Menon. "Test Generation Algorithm for Nonprocedural Hardware Description Languages", *IEEE FTCS-11,* 1981, pp. 200–105.
16. Su, S. Y. H., and C. L. Huang. "A Multi-Level Hardware Design Language LALSD II and its Translator," *Proceedings of the 1981 International Symposium on CHDL's and Their Application,* 1981.
17. Tsuchiya, M., and L. V. Ramamoorthy. "Design of a Multi-Level Microprogrammable Computer and a High-Level Microprogramming Language," University of Texas at Austin, Tech. Report 135, 1972.
18. Hill, F. J., and B. Hueg. "SCIRTISS: A Search System for Sequential Circuit Test Sequences," *IEEE Transactions on Computers,* Vol. C-26, No. 5, 1977, pp. 490–502.
19. Ciaramella, A. "Testing of Microprogrammed Units," *IEEE FTCS-9,* 1979, pp. 161–163.
20. Su, S. Y. M., and Yu-I Hsieh. "Testing Functional Faults in Digital Systems Described by Register Transfer Language," *1981 IEEE Test Conference,* 1981, pp. 433–439.
21. Milner, R. "An Algebraic Definition of Simulation between Programs," *Proceedings of the 2nd International Joint Conference on Artificial Intelligence,* 1971, pp. 481–489.
22. Darringer, J. A., and J. C. King, "Application of Symbolic Execution to Program Testing," *Computer,* Vol. 11, No. 4, 1978, pp. 51–60.
23. Crocker, S. D., L. Marcus, and D. van-Mierop. "The ISI Microcode Verification System," *Firmware, Microprogramming and Restructurable Hardware,* North-Holland Publ. Co., New York, 1980, pp. 89–103.
24. Dasgupta, S. "Some Implications of Programming Methodology for Microprogramming Language Design," *Firmware, Microprogramming and Restructurable Hardware,* North-Holland Publ. Co., New York, 1980, pp. 243–252.
25. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming," *Communications of the ACM,* Vol. 12, No. 10, 1969, pp. 576–583.
26. Eckhouse, R. H. "A High Level Microprogramming Language (MPL)," *Proceedings of the 1971 Spring Joint Computer Conference,* AFIPS Press, Arlington Virginia, 1971.
27. Dewitt, D. J. "Extensibility—A New Approach for Designing Machine Independent Microprogramming Languages," *SIGMICRO Newsletter,* Vol. 7, No. 3, 1976, pp. 33–41.
28. Malik, K., and T. G. Lewis. "Design Objectives for High-Level Microprogramming Languages," *SIGMICRO Newsletter,* Vol. 9, No. 4, 1978, pp. 154–160.
29. Carter, W. C.; W. H. Joyner, and D. Brand. "Microprogram Verification Considered Necessary," *Proceedings of the 1978 National Computer Conference,* AFIPS Press, Arlington, Virginia, 1978, pp. 657–664.
30. Berg, H. K., and W. R. Franta. "Firmware Engineering: Critical Remarks and a Proposed Strategy," *Firmware, Microprogramming and Restructurable Hardware,* North-Holland Publ. Co., New York, 1980, pp. 41–64.
31. Richter, L. "High-Level Language Extensions for Micro-Code Generation and Verification," *Firmware, Microprogramming and Restructurable Hardware,* North-Holland Publ. Co., New York, 1980, pp. 233–242.

32. Davidson, S., and B. D. Shriver. "MARBLE: A High Level Machine Independent Language for Microprogramming," *Firmware, Microprogramming and Restructurable Hardware,* North-Holland Publ. Co., New York, 1980, pp. 253–266.
33. Williams, T. W. and K. P. Parker. "Design for Testability—A Survey," *IEEE Transactions on Computers* Vol. C-31, No. 1, 1982, pp. 2–15.
34. Marwedel, P. "A Retargetable Microcode Generation System for a High-Level Microprogramming Language," *SIGMICRO Newsletter,* Vol. 12, No. 4, 1981, pp. 115–123.
35. Van Mierop, D., S. Crocker, and L. Marcus. "Verification of the FTSC Microprogram," *SIGMICRO Newsletter,* Vol. 9, No. 4, 1978, pp. 118.
36. Zimmermann, G. "The MIMOLA Design System: A Computer Aided Digital Processor Design Method" *Proceedings of the 16th Design Automation Conference,* 1979, pp. 53–58.
37. Thatte, S. M. and J. A. Abraham. "Test Generation for Microprocessors," *IEEE Transactions on Computers,* Vol. C-29 (1980), pp. 429–441.
38. Robach, C., and G. Saucier. "Dynamic Testing of Control Units," *IEEE Trans. on Computers,* Vol. C-27, July 1978, pp. 617–623.

The 5.25-inch fixed/removable disk drive

by DON M. MINAMI
DMA Systems Corporation
Santa Barbara, California

ABSTRACT

The fixed/removable 5.25-inch Winchester drive provides combined computer peripheral support functions, such as mass storage, input/output, and backup. The 13.5-MByte total capacity (6.75 MBytes fixed/6.75 MBytes removable) is packaged in a unit about the size of a shoebox.

Reliability has been the major factor in determining the design parameters of the fixed/removable drive. Not only has Winchester reliability been enhanced, but preventive maintenance has been eliminated.

INTRODUCTION

Reliable mass storage at a relatively low cost is the driving force behind the trend toward increased use of Winchester disk technology for small computer systems. Although the conventional Winchester drive offers high reliability due to its nonremovable media, it requires some form of data file backup. One solution is to use a tape drive for backup; this allows adequate backup storage capacity, but it is too slow and not form-factored for many small computer systems. Another solution is to use a flexible disk drive, but it does not provide sufficient mass-storage capacity without resorting to multiple diskettes.

A better solution is a Winchester disk drive for both fixed and removable media in a single unit that provides mass storage, input/output, and backup. The 5.25-inch Micro-Magnum 5/5 (see Figure 1) from DMA Systems is the first such drive with a fixed disk and a removable disk cartridge built to the proposed ANSI standard. The Micro-Magnum 5/5 offers significant reliability and performance advantages which include the following:

Figure 1—Micro-magnum 5/5

1. No preventive maintenance or head alignment required
2. Heads retracted from the media surface, thus preventing damage
3. The necessity of external backup devices and their controllers is eliminated
4. Reduced space and power requirements
5. Faster backup time because of a higher disk transfer rate
6. Reduced component count
7. Reduced overall system cost

The origin of the Micro-Magnum 5/5 and the product design specifications were derived from a market survey. This survey included inputs from system manufacturers, system integrators, component suppliers, and computer industry consultants. The result of this survey was a product specification which emphasized reliability in terms of product life, data integrity, data interchange, and freedom from preventive maintenance.

GENERAL SPECIFICATIONS

The Micro-Magnum 5/5 is designed with 6.75 MBytes (5.0 formatted) fixed and 6.75 MBytes (5.0 formatted) removable. The drive uses an ANSI-proposed 5.25-inch removable disk cartridge (see Figure 2) with 5.0 MBytes of formatted data. The front panel face is 3.25 inches high by 5.75 inches wide, which is typical of 5.25-inch Winchester drives.

The 5 MBytes per disk formats 306 tracks with 33 sectors

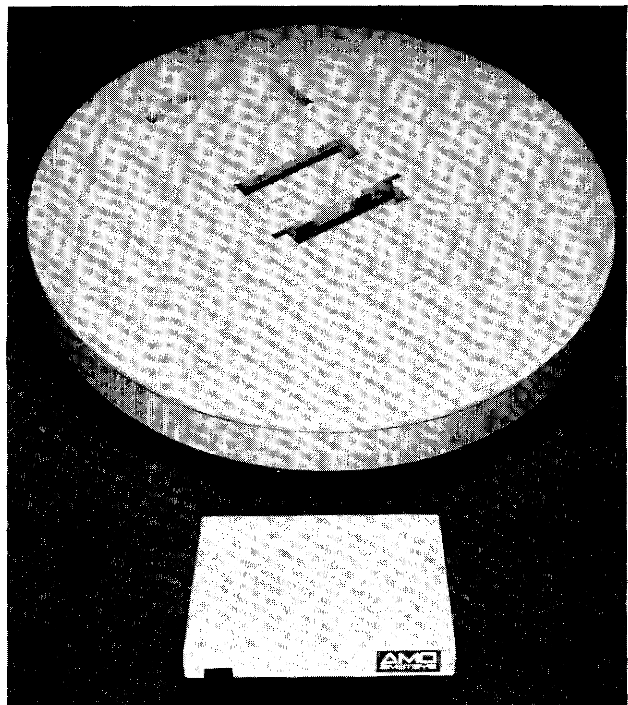


Figure 2—Micro-magnum cartridge compared to the traditional 14-inch disk cartridge

(one spare sector) and 256-byte sectors on each surface. The recording density is 8600 fci using MFM encoding, and the track density is 450 tpi.

TRACK FOLLOWING

Accurate and repeatable positioning of the read/write heads is a necessity in the fixed/removable drive in order to maintain data interchangeability. The primary complication is the removable cartridge being used as a means for data interchange and transportability.

The drive, in conjunction with the disk cartridge, must allow for mistracking errors, cartridge registration errors, temperature gradients, spindle runout, and head-track width tolerances. This mechanical error budget requires a track-following system which will compensate for these variations. The elimination of precise head alignment is also as important; the market survey indicated that any such field maintenance procedures would not be tolerated.

To overcome the errors due to thermal expansion, the problems of cartridge interchange, and the elimination of field maintenance, embedded servo positioning was selected for the Micro-Magnum 5/5. Embedded servo data (see Figure 3) is prerecorded during the manufacturing of the drive and the cartridge, and it is contained in the 26 bytes at the start of each sector. The embedded servo format has been submitted to ANSI for standardization. (A copy of the proposed servo format can be obtained by contacting DMA Systems or ANSI.)

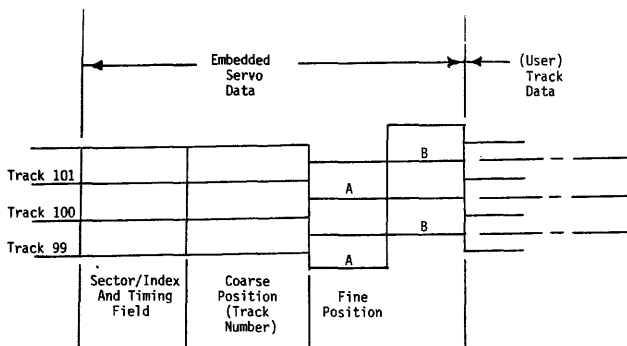


Figure 3—Embedded servo data

Embedded servo positioning is a two-step process. First, course positioning allows the proper track to be located; second, the fine positioning locates the read/write over the center of the desired track. As the desired track is being sought, the course-positioning process is activated. The course-positioning process uses a Gray code for each track number and is prerecorded as part of a 26-byte servo format. As the desired track is approached, within half a track, fine positioning takes over. Prerecorded signal segments A and B (see Figure 3) define the fine-positioning servo bursts. The edges of A and B are along the centerline of the tracks, so that a head centered exactly on a track will read equal amplitudes from both segments. If the head is off-center, one amplitude will read higher and the other, lower. The difference is detected and used as an error signal to drive a linear motor positioner to seek a zero error to maintain the proper track centerline position.

LINEAR MOTOR

The linear motor, in conjunction with the embedded servo track-following system, provides not only fast access time (40 msec average) but also reliability. Figure 4 shows the Micro-Magnum 5/5 linear motor positioner assembly.

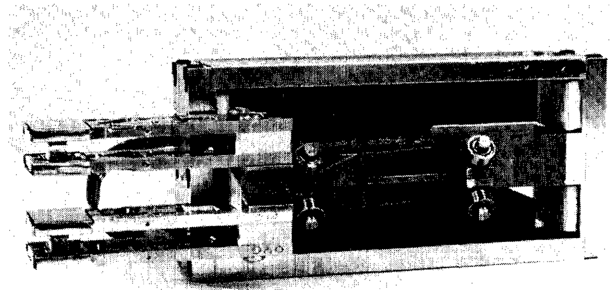


Figure 4—Carriage head and linear motor

The reliability features of the linear motor positioner are the following:

1. The heads are allowed to be fully retracted off the disk surface and latched into position inside the drive.
2. Contamination control is improved due to the smaller cartridge and drive-door openings.
3. Head gaps move in a radial line, giving the best possible tolerance for cartridge interchange.
4. The structural resonance is better controlled.
5. Manufacturing of the head-carriage assembly is simplified.

HEAD-MEDIA CONTACT

Two problems can result from head-media contact; the head and/or the media surface can be damaged. Therefore, optimal data reliability can only be obtained by making it impossible for the head to ever make contact with the media. In the Micro-Magnum drive, the heads are never allowed to make contact with the disk. This is achieved by a patented head design (see Figure 5) which allows a Winchester air bearing to be loaded dynamically onto a spinning disk. (Forty-thousand load/unloads have been successfully completed with no damage to head or media.) The heads are also retracted completely off the media when the drive is shut down.

Reliability is significantly enhanced using a dynamic load/unload head design. Avoided are reliability compromises that exist with typical Winchester drives, which allow heads to

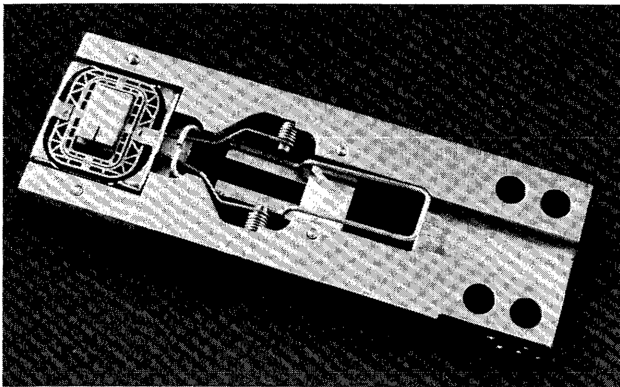


Figure 5—DMA systems composite head assembly

start/stop on the media. Eliminated problems are the following:

1. The heads wringing onto the media
2. The heads landing on top of contaminants even after a purge cycle
3. Heads and/or media being damaged during transit, during shipment, or when the system is transported from one desk top to another

CONTAMINATION CONTROL

In typical office environments, contaminants such as smoke and dust can cause severe damage to the heads and media. Contamination control is therefore a very important reliability consideration. Figure 6 shows the Micro-Magnum's high-capacity closed-loop air system.

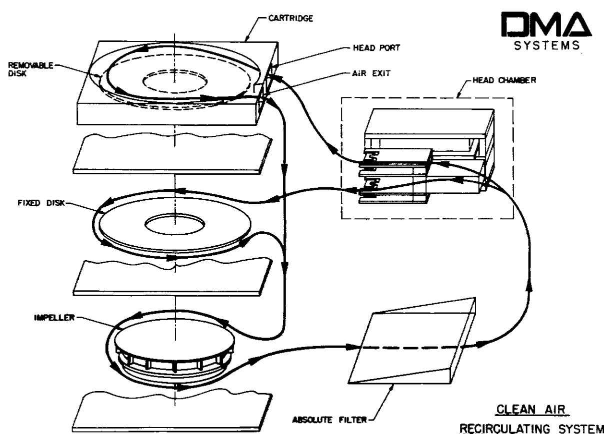


Figure 6—Closed-loop air filtration

The closed-loop air filtration system is designed so that an impeller generates sufficient system air flow to move a volume of air through the recirculating filter once per second. (The filter has a design life of five years, with no filter change required in a normal office environment.) During a purge cycle, this allows efficient removal of contaminants that may have been introduced during the cartridge insertion.

The Micro-Magnum 5/5 drive, as well as the cartridge, have self-sealing doors to preclude contaminants from entering their respective compartments. The drive has a door that seals the head port opening and keeps contaminants from entering the drive's clean air compartment. It is not necessary to take any precautionary measures to assure that the cartridge insertion door is secured and closed. The cartridge also has a door that closes the head opening and a clamp that secures the hub against the cartridge to prevent contaminants from entering the cartridge. Because the drive compartment is sealed and not accessible, the total volume of contamination that can enter the clean-air system is limited to the cartridge at the time of insertion.

ELECTRONIC SYSTEM

The electronic packaging of the Micro-Magnum 5/5 was no minor task, considering all the electronic functions that had to be housed in a 3.25-inch by 5.75-inch by 10.50-inch volume. Complicating the design was the necessary circuitry for the embedded servo and voice-coil positioner.

The Micro-Magnum's electronic block diagram is shown in Figure 7. A dual–microprocessor system was employed to conserve space and partition functions in order to make firmware design simpler. MPU1 is dedicated to the interface and status functions that include all controller input and output lines, front panel functions, safety checks, and fault algorithms. MPU2 receives embedded servo information from the servo decode circuit (LSI2). This serves the basic servo functions, such as track follow, seek, re-zero, load, and retracking of the heads.

To achieve the required packing density, two CMOS gate array custom IC's were developed. LSI1, a 200-gate array, is used to control spindle servo. LSI2, a 500-gate array, is used to perform decoding of digital information in the embedded servo fields.

An all-important electronic function is the control of the write operation to prevent overwriting the embedded servo fields. Overwriting the embedded servo field could result in the loss of removable and/or fixed data. The Micro-Magnum 5/5 drive, therefore, has a series of hardware and software safety checks that are performed before a write operation is allowed. Hardware functions are gated directly to the write current enable function of the head read/write chip. Also, all the following conditions must be true simultaneously before the logic circuits allow write current to be enabled:

1. Spindle speed must be within 0.1%.
2. Heads and head circuits must be in a safe condition; i.e., no shorts or opens, only one head selected, MFM data being received.
3. All power supplied must be in tolerance.
4. Power must be safe, spindle must be on, write gate signals must be enabled on the interface, and the drive must be selected.

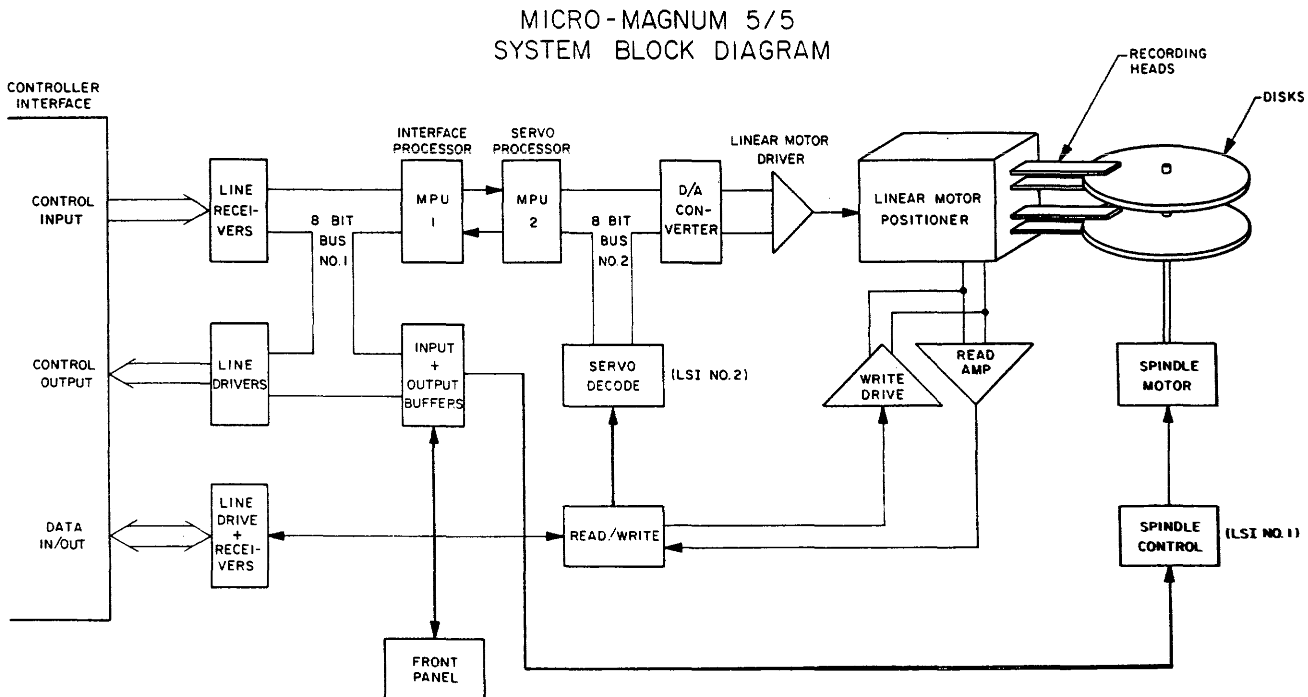


Figure 7—System block diagram

5. The previous embedded servo field must be decoded properly, including a correct sector/index field and clock-shift check code.
6. Servo system must indicate that the head is within the "on track" limits as determined from the fine position information.
7. A redundant spindle-speed check circuit must indicate that the spindle is within the allowed 0.5% of normal.
8. The write protect switch for the disk to be written must not be activated.

REDUNDANCY

There is always a possibility that data errors can occur during system operation. Therefore, the disk drive must have the capability to provide data redundancy and error correction in a manner that is transparent to the user. This can be achieved by providing spare sectors and alternate tracks on the disk, as well as data formats that allow the controller or host computer to provide user-transparent correction techniques.

In the Micro-Magnum 5/5, one spare sector per track and five spare tracks per surface are provided to replace those found to be defective. This allows 4.5% media redundancy for the accommodation of defects. The defect-tolerant system is further enhanced by provisions for CRC (cyclic redundancy checking) and ECC (error-correction coding) in the data formats.

Error-correcting technology serves to verify header and data field accuracy, plus providing the capability for correction of errors. Most errors can be corrected by the combination of CRC and ECC techniques, and no data will be lost. This is accomplished by using an intelligent controller or the host computer in conjunction with the CRC and ECC formats of the Micro-Magnum system.

Defective track correction can be handled in two ways by the intelligent controller of host computer. These are:

1. After a seek to a defective track has been completed, the Bad Track Flag in the first sector tells the controller that an alternate track has been assigned. The data field information is then read and a new seek is issued to the assigned alternate.
2. Alternatively, the alternate track catalog can be read and stored by the controller upon the initial spindle-up sequence after a cartridge is installed. If a seek to a bad track occurs, the controller automatically issues a seek to the assigned alternate track. The same algorithm can be implemented by the host computer.

"Hard" errors can usually be corrected to protect data, using the error-correction techniques. If a defect cannot be error-corrected, it should be mapped into the defective sector category and spared out by the appropriate method. If the sector is spared while it is still a correctable defect, no data will be lost.

When a new defect is spared and an alternate track is required, the alternate–track catalog must be updated along with the data field information on the bad track.

DATA TRANSPORTABILITY/ INTERCHANGEABILITY

Use of a removable cartridge using embedded servo permits a reliable mass-storage system that is transportable and inter-

changeable with other similar systems. The disk cartridge has been accepted as a proposed ANSI standard for the mechanical configuration of the removable cartridge; this allows the mechanical standard to be used in all similar systems. However, no standard has yet been established for the data formats on the 5.25-inch fixed/removable Winchester drive. With the hope that a standard can be established that provides data file compatibility, the following information on the data formats for the Micro-Magnum 5/5 is presented.

Using MFM (modified frequency modulation) encoding, the disk is organized into tracks of 10,890 bytes each of unformatted capacity. Each track is divided into 33 sectors of 330 bytes each. When formatted, each sector contains 256 bytes of data, 48 bytes of format information, and 26 bytes of embedded servo information. Figure 8 shows the organization of each sector. It is detailed below:

1. *Embedded Servo Field*—Track and sector location information is embedded in 26 bytes.
2. *PLO (Phrase-Locked Oscillator) Sync*—Consists of 12 bytes of 000's transmitted for data separator synchronization.
3. *ID and Data Address Marks*—a 1-byte address mark, made unique by omitting the clock transition between bits 4 and 5, precedes both the ID and Data Addresses. The 1-byte, FE (hex), identifies the ID Address Mark; and the 1-byte, F8 (hex), identifies the Data Address Mark.
4. *Write Splice*—This byte is provided between the ID field and Data field PLO Sync to turn on the write current if data is to be recorded in the Data field.
5. *Data Pad*—To guarantee data integrity, a 1-byte pad is provided between the final ECC field and the speed buffer area.
6. *Speed Buffer*—A 5-byte buffer at the end of the sector accommodates spindle-speed variations up to ± 0.75%.
7. *Sector Interleave*—As recorded at the factory, a Sector Interleave factor of 4 is applied to the sector ID field. This sequence of sector ID fields is as follows: 0, 8, 16, 24, 1, 9, 17, 25, 2, 10, 18, 26, etc.
8. *CRC*—This 2-byte field is used to implement the CCITT CRC polynomial, $(X^{16} + X^{12} + X^5 + 1)$, for error detection in both the ID and Data fields.
9. *ECC*—A 3-byte field reserved for appending an error correction code to both the ID and Data fields.

The three bytes associated with format information provide additional data:

10. *Cylinder/Head/Sector*—Provision is made to address up to 1024 cylinders, 8 heads, and 64 sectors. The head byte also contains the two MSBs of the 10-bit cylinder code and three 1-bit control flags listed in the following items (11 and 12).
11. *Write-Protect Sector Flag*—A ONE set in this bit location indicates to the controller that this sector is "write-protected" and cannot be overwritten by the host computer.
12. *Bad Sector/Bad Flag Track Flags*—These 1-bit flags alert the controller that either a bad sector or a bad track has been detected; this allows them to be replaced by space sectors or tracks.

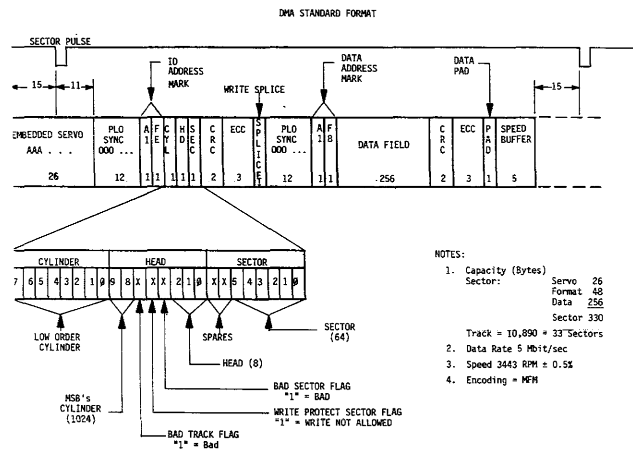


Figure 8—Data format

CONCLUSIONS

By virtue of its relatively low cost, high data reliability, and small volume, the Micro-Magnum 5/5 5.25-inch Winchester disk drive is destined to be a widely used mass-storage device for small computer systems. The drive employs a number of proven, mature technologies that are integrated for the first time to provide a capability never before available. The only remaining consideration is the establishment of standard data formats that will allow universal interchangeability and transportability.

Practical CMOS microprocessor systems

by BILL HUSTON
Motorola, Inc.
Austin, Texas

ABSTRACT

Many have felt that complementary metal-oxide silicon (CMOS) has not yet become a practical semiconductor technology for microprocessor-based systems. Recent progress has made that impression obsolete. A selection of CMOS microprocessors is available at speeds matching N-channel metal-oxide silicon microprocessor units (NMOS MPUs). CMOS memories have also become broadly available in the last few years. The needed peripheral circuits are now appearing. A CMOS parallel interface peripheral provides 24 interface pins and is bus-compatible with practically all the new-generation CMOS microprocessors. The last element needed to assemble practical all-CMOS microprocessor systems are the small-scale integration/medium-scale integration (SSI/MSI) logic functions. Gates, decoders, latches, and flip-flops are typically needed to operate a bus structure of a multichip system.

This report concentrates on the newest methods of achieving a full-performance all-CMOS microprocessor system. The focus is on the parallel interface peripheral and on using CMOS logic functions in practical bus connections.

PRACTICAL CMOS MICROPROCESSOR SYSTEMS

CMOS, as a semiconductor technology, has for years had a series of recognized benefits. Microprocessors have of course created whole arrays of new electronic uses as well as reconfiguring many conventional electronic products. But until now, combining the CMOS traits with the proliferation of microprocessors has occurred in only a small percentage of the applications. Most of the reasons for the slow acceptance of CMOS as a practical microprocessor technology have now dissipated.

CMOS AS A PRACTICAL MICROPROCESSOR TECHNOLOGY

Most of the attraction of CMOS is associated in some way with battery powering or power saving. There are other CMOS benefits—better noise immunity is a key one—but most CMOS microprocessor applications use batteries for primary or backup power. Some use other low-power energy sources, such as solar cells or very large capacitors.

There is a long-standing impression that CMOS is too slow for many microprocessor uses. The CMOS-is-too-slow image is no longer valid. Metal gate MOS, whether single-channel (NMOS or P-channel MOS [PMOS]) or complementary, is much slower than silicon gate MOS. Most of the high volume MOS processes today are silicon gate, which have the same throughput capability in N-channel (NMOS and high-density N-channel [HMOS]) as in CMOS, given the same device sizes. However, many CMOS users intentionally slow the system down to extend battery operating time.

Some prospective CMOS microprocessor users may have hesitated because of a narrow choice of processors. Until a year or so ago, only two processors were available. Some considered the architectures difficult to accept when compared to the many familiar 6800 and 8080 types of processors available in NMOS. Now, in addition to the traditional (such as the 1802), users have 8080 derivatives (NSC-800 and 80C35) and 6800 derivates (the MC146805E2) to choose from. Higher-performance CMOS processors are rumored in both traditional NMOS camps. Hesitating to use CMOS microprocessors because of limited architectural choices is outmoded.

Another potential source of hesitation is the fear that CMOS microprocessors are produced only in low volume and thus will always be high-priced. It surprises some to learn that the 1802 is Number 5 in production microprocessor volume, according to *Dataquest,* behind only the 8080, 6800, Z80, and 6502. CMOS is attractive in certain volume automotive situations. Some of the newer CMOS microprocessors are part of a family of single-chip' microcomputers. Read-only-memory (ROM)-based single-chips are built for dedicated volume applications. The ROM-less MPUs benefit from the volume-driven learning curve of the single-chips when they use the same processor and production process. There are volume applications in a number of fields for 8-bit CMOS single-chip microcomputers. Production volume allows costs to be lowered, which should reduce any hesitancy to consider CMOS a practical microprocessor technology.

PRACTICAL SYSTEM NEEDS

Assembling a multichip all-CMOS microprocessor system is now practical. The various elements of such a system are considered in turn, including the microprocessor, memory, peripherals, and interconnecting glue (SSI/MSI logic functions).

A typical mid-range CMOS microprocessor is the MC146805E2. The instruction set is a control-optimized derivative of the MC6800, including single-instruction bit modify and test and low-power stand-by instructions. The interface bus connects to external memory and peripherals using address-then-data multiplexing, as on many newer N-channel processors. Included are 112 bytes of on-chip RAM for stack and data storage. A 15-stage counter is used for timer functions, such as periodic interrupt generation, pulse width measurement, and event counting. Sixteen bidirectional I/O pins are addressable as individual bits or as 8-bit ports. In smaller systems the only external element needed is a program ROM or electronically programmable ROM (EPROM).

The second set of elements for an all-CMOS system is memories. Bus-compatible ROM is available with the MCM65516, which contains 2K bytes of mask ROM in a compact 18-pin package. The multiplexed bus is compatible with the MC146805E2, as well as the NSC-800 and 80C35 microprocessors. Nonmultiplexed bus EPROMs such as 27C16s are now available for program storage in lower-volume applications. One source offers an address-latched version called the 67C16 for use with multiplexed bus microprocessors. For data storage 4K CMOS static RAMs have been available for some time in industry standard 4K × 1 and 1K × 4 configurations. The availability of 16K static CMOS RAMs may improve soon. Both bus-compatible and industry-generic CMOS memories are available for microprocessors. It is beyond the scope of this report to survey memories deeper.

The third major element of an all-CMOS microprocessor system is peripherals. Fortunately, CMOS users do not need as many peripheral integrated circuits (ICs) as NMOS MPU users. CMOS MPUs are seldom used with mechanical and

electrical devices that consume large amounts of power, such as floppy disks and cathode ray tubes (CRTs).

The most basic interface element is parallel-port input/output (I/O) connections. The MC146823 provides three 8-bit parallel interface ports along with handshaking port control signals in a bus interface peripheral. The latter portion of this report focuses on this parallel interface peripheral, since its bus interface allows direct connection to all new-generation CMOS microprocessors announced to date.

A frequent function of a CMOS MPU system is to keep the time of day, and often a calendar as well. The MC146818 Real-Time Clock Plus RAM maintains the time in seconds, minutes, and hours. It maintains a 100-year calendar, including day of the week, leap year, and daylight-saving changes. Some of the auxiliary system functions included are 50 bytes of uncommitted RAM, a periodic interrupt, an alarm interrupt, a square-wave output pin, and a microprocessor clock oscillator.

Other peripheral functions are available in the market. Generic asynchronous universal receiver and transmitters (UARTs) have been available for some time from two or three sources. Though not directly bus-compatible, they are easily interfaced. The RCA 1802 family includes some peripheral functions that could be useful with other CMOS processors. Some examples are a multifunction timer and an arithmetic chip. A little interface adapting is needed to use such parts with the MC146805E2 type of buses, but there are situations where it would be worthwhile. The two peripherals introduced with the NSC-800 are usable on other processors with very little adapting. The RAM plus I/O part is useful in low-volume cases where the needed memory-to-I/O ratio is close to that included in the part.

The fourth major element needed to assemble an all-CMOS system is the interconnect SSI and MSI logic. Few complex systems can be assembled totally with large-scale integrated (LSI) circuits. Until recently the CMOS standard logic functions have been slow; this trait frustrated attempts to take advantage of the available microprocessor performance. Full-speed "glue" parts are now becoming available in the 74HC00 family from Motorola and National. This line includes all the popular 74LS00 family functions, at the same speed as the low-power Schottky transistor transistor logic (LS-TTL) family, but with CMOS power usage.

The next section of this report looks at some of the bus interface uses for the CMOS SSI glue parts in the 74HC00 family.

BUS INTERFACING

The practicalities of putting together an all-CMOS MPU system are, of course, applications-dependent. This section outlines a few techniques that might be useful. The goal is to trigger user creativity with ideas, not to establish a standard way to interface to a microprocessor. For example, some applications need more memory than the program can directly access, so memory expansion techniques are appropriate. Many generic memories cannot accept the MPU bus control signal formats provided.

Bus Control Signals

Frequently the bus control signals that emanate from a microprocessor need to be modified for use by memories and peripherals. Figure 1 shows that the MC146805E2 microprocessor creates an Address Strobe (AS), a Data Strobe (DS), and a Read/Write level. Figure 2 shows how these three control signals are associated with read and write bus cycles. The memory and I/O cycles are identical, since a common address architecture is used. Figure 3 shows how three new control signals may be generated. Generic memories (industry standard Joint Electronic Device Engineering Council (JEDEC) pin functions, as opposed to directly bus-compatible) usually require a low-going read pulse, frequently called output enable. RAM writes are indicated by a low-going write pulse. Many peripherals accept similar signals. Figure 3 shows that two gates and an inverter create the needed read and write pulses.

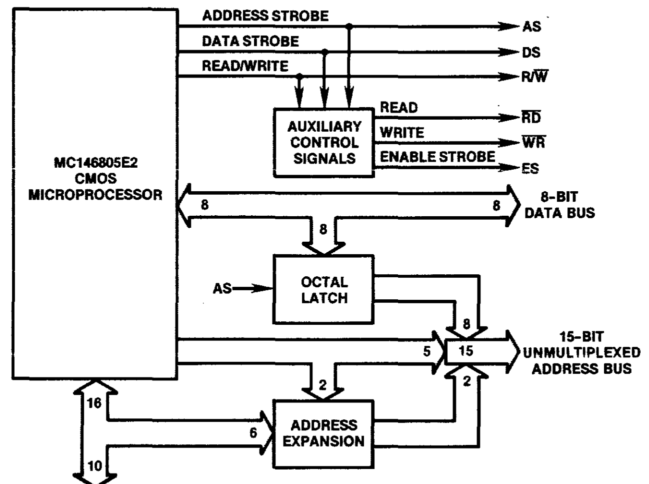


Figure 1—CMOS microprocessor bus interface adaptations

Static memories are not always fully static today. Though memory retention may be static, the internal decoders and buses may need to be cleanly waked up to initiate a successful bus cycle. Careful reading of data sheets reveals that chip enable inputs can no longer accept address decoding transitions. Once chip enable is asserted, it must remain, without bounce, until the access cycle is complete. The Data Strobe microprocessor signal is a clock with clean edges that can be ANDed with address decoding to create a clean chip enable. The problem is that fast memories are then needed, since DS begins quite a while after the addresses are stable. Figure 3 shows how an earlier chip enable strobe can be generated using less than one-and-a-half packages of SSI logic. Figure 2 shows that the Enable Strobe (ES) begins with the falling edge of Address Strobe and lasts until the end of Data Strobe. The address is stable at the leading edge of ES, and the unmultiplexed address remains stable for the duration of ES.

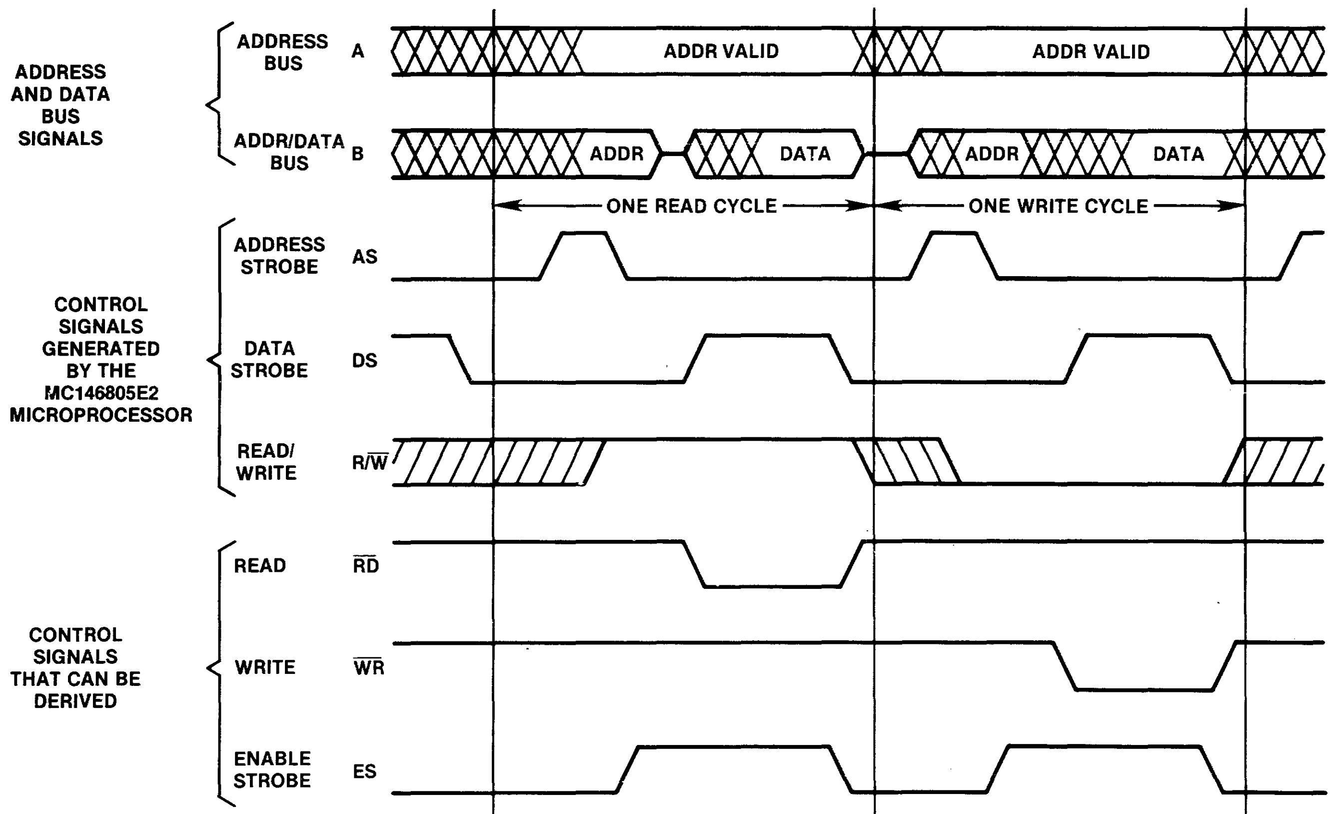


Figure 2—Auxiliary bus control signal timing

Address Expansion

The second step in generating a larger system is to create an extended unmultiplexed address bus. Figure 4 shows a typical example. Most new-generation microprocessors time-multiplex portions of the address onto the data bus. Address bits appear on the bus during the first part of the bus cycle and are identified with an Address Strobe signal (Address Latch Enable [ALE] in other processors). Then, after the memory begins its access time, the bus is switched over to carry data during the latter portion of the bus cycle. Data Strobe identifies the data portion of the bus cycle (Read and Write with other processors). Figure 2 shows the time-multiplexing relationships.

An increasing percentage of available memories and peripherals are including address latches. Some accept an AS

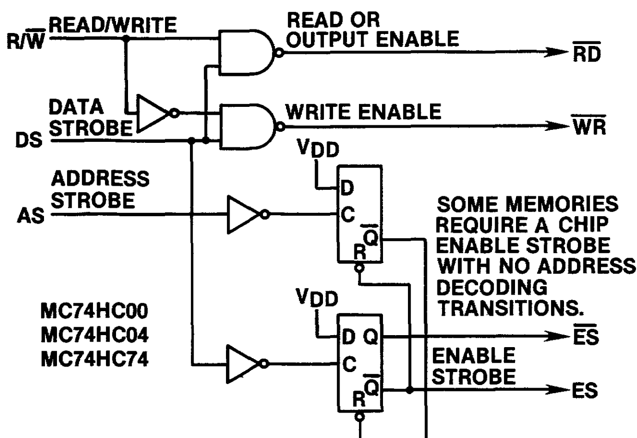


Figure 3—Creating auxiliary bus control signals

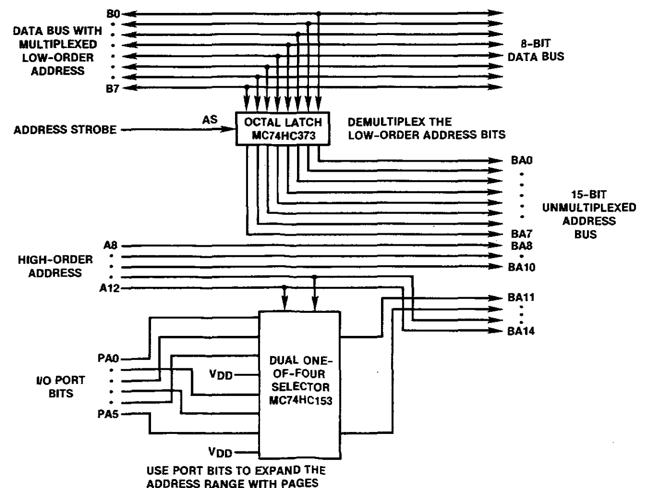


Figure 4—Extended unmultiplexed address bus

(ALE) signal directly, whereas others need an ES signal as a part of a chip enable input. Since many memories still need unmultiplexed addresses, Figure 4 shows an octal latch to save the low-order eight address bits.

The address expansion technique illustrated in Figure 4 uses one MSI part, a dual one-of-four selector, and six port bits to add two bits to the 13 address bits created by the microprocessor. Figure 5 shows the resulting page-addressing arrangement. A program sees 8K of logical address space split into four 2K byte pages. A common convention would be for the interrupt and other I/O routines to be in the fourth page, along with a centralized page-changing routine. The remaining three 2K byte pages could freely include programs and data in whatever mix was appropriate. Each of the three logical user pages can be mapped to one of four physical pages of 2K bytes. The physical address space would include 12 pages of 2K bytes each (24K bytes total), of which only 6K bytes are visible at any time. When a program needs data that are not visible, it calls the executive in the fourth page (probably with a software interrupt instruction) to have the pages switched. The executive can switch pages via the I/O port without losing control, since the fourth page is reserved as always visible.

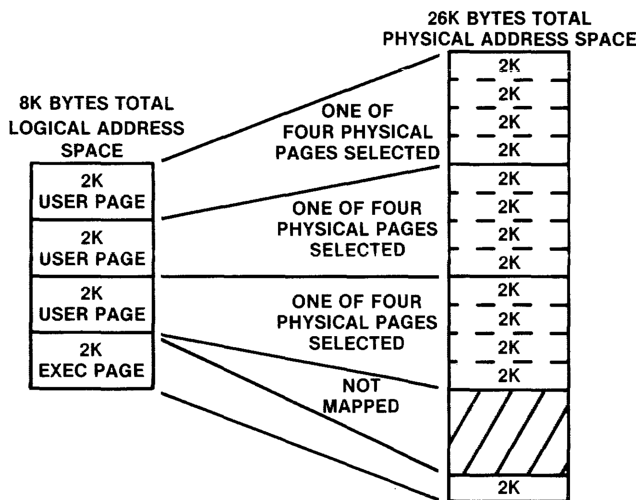


Figure 5—Expanded addressing with paging

The address expansion scheme outlined above is an example of one of the possible techniques. Figure 6 shows that other variations of the method can allow addressing to over a quarter of a million bytes by using only three glue parts and only the 16 I/O pins included on the MC 146805E2 processor. Other expansion techniques could be as easily used, but they are beyond the scope (or space) of this report. The processor program may only have a 13-bit address, but it can access as much memory as is needed.

Typical Expanded System

It has been shown that generalized bus control signals can be easily created and that an unmultiplexed address bus of 15

TO OBTAIN THIS TOTAL ADDRESS-ABILITY	ONLY THESE GLUE PARTS ARE NEEDED		USING THESE MC146805-E2 PORT PINS	THE ADDRESS BUS SIZE IS	THE SIZE OF THE LOGICAL PAGE IS	THE NUMBER OF LOGICAL PAGES IS
	MC74-HC151	MC74-HC142				
8K	0	0	0	13	8K	1
15K	1	0	7	14	1K	8
26K	0	1	6	15	2K	4
29K	2	0	14	15	1K	8
44K	3	0	16	16	1K	8
98K	0	2	12	17	2K	4
290K	0	3	16	19	2K	4

Figure 6—Various extended memory sizes

bits or more can be easily generated. The next step is to look at a typical extended system. Figure 7 shows an extended address map that fills the space available with a 15-bit address bus created as in Figure 4. Most of the first 128 bytes are the on-chip RAM, I/O, and timer locations. Then 16K bytes of off-chip RAM is included, of which two 2K bytes pages would be mapped into the logical address space at a time. The third logical page consists of one of four 2K byte EPROMs. The fourth logical page is not mapped and contains the control programs in a 2K byte ROM. Of the 2K of ROM space, 192 bytes are used for I/O functions, in particular a real-time clock and eight parallel interface peripherals. The latter would interface to 192 pins, allowing for large systems.

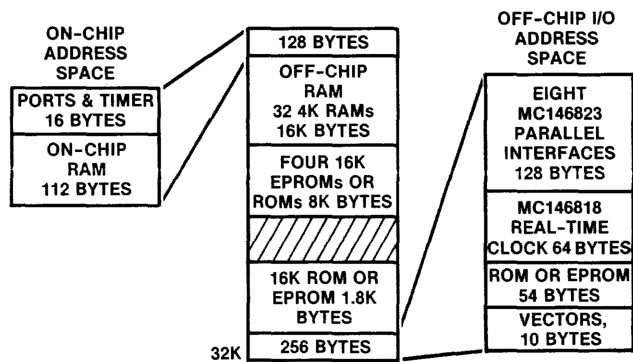


Figure 7—Typical expanded address map

Figure 8 shows how the above address map could be easily decoded with available glue parts. Three-to-eight decoders are shown to decode chip enable signals for the RAM, ROM/EPROM, and the multiple parallel interfaces. NAND gate address range detection is shown for the peripherals. The fourth page ROM is disabled when the peripherals are accessed. In systems where demultiplexing is not needed, the decoders could include an address latch—the MC74HC137, for example.

Bus Interface Flexibility

The typical extended system above includes one microprocessor, nine LSI peripherals, and 41 memory parts. Only 10 SSI/MSI glue parts are needed to assemble a practical

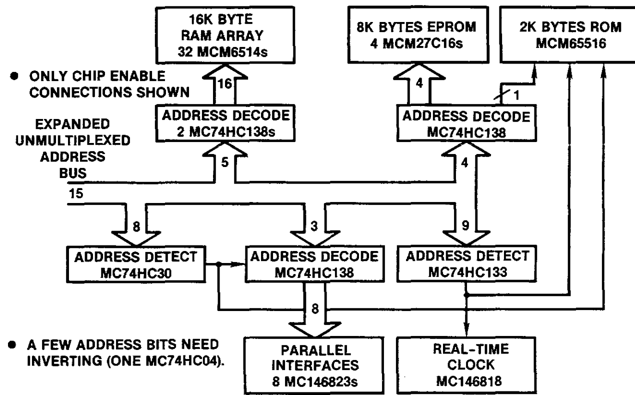


Figure 8—Typical expanded address bus

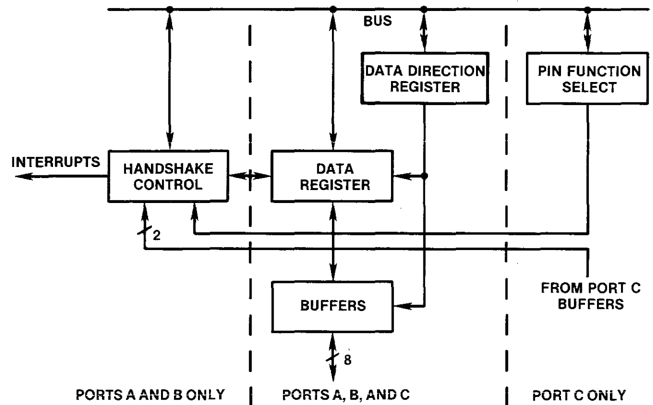


Figure 10—Parallel interface ports

system with an impressive set of features. The processor is a powerful 8-bit control-oriented MPU. Interfacing to 192 interface pins is provided, of which up to 32 could be interrupt sources. The memory includes 16K bytes of RAM and 10K bytes of EPROM or ROM. In addition, the time of day is automatically maintained in a peripheral.

With all that system power, it is not valid to consider an advanced larger all-CMOS microprocessor system to be impractical or too expensive.

PARALLEL INTERFACE PERIPHERAL

CMOS microprocessors such as the MC146805E2 include some parallel I/O pins on the MPU part. When the on-chip I/O is insufficient, a parallel I/O peripheral part is needed. The MC146823 provides 24 I/O pins to an MC146805E2, NSC-800, 80C35, or 80C48.

Figure 9 shows the 8-bit bus interface on the left and the 24 parallel I/O pins on the right. The I/O ports are looked at first, followed by the handshake functions and the generalized processor bus interface.

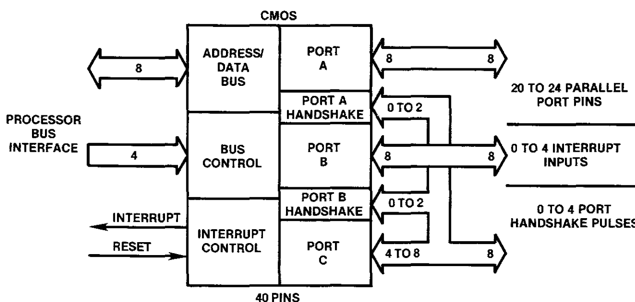


Figure 9—MC146823 Parallel Interface peripheral

Three Parallel Ports

The processor program establishes the purpose of the 24 parallel I/O pins. Figure 10 illustrates the program selection features.

Each 8-bit port includes a data direction register. With power-on reset the data direction is initialized so that all pins are inputs. All output drivers are in the high impedance state to avoid a situation in which two circuits try to drive the same pin to opposite states. After power-on, the program establishes each pin as an input or an output. Each data direction register bit establishes the corresponding port pin as an output or an input. There are no restrictions on the number of input or output pins, nor on which pins of an 8-bit port are inputs or outputs.

The data register associated with each port is used primarily for output storage. When the data direction register bit indicates output, the state of the corresponding data register is driven onto the I/O pin by the output buffers. When a pin is designated as an input, a program read transfers the state of the I/O pin to the processor bus, bypassing the data register. For output bits the data register is a read/write register. A program read of an output pin gets the state of the data register, not the I/O pin. This permits read/modify/write cycles, such as the MC146805E2 bit manipulation instructions, to read the port, change one or more bits, and write the result back to the port data register.

The data register in Port A has one additional use. A handshaking pin causes input data to be latched for subsequent program reading when this feature is enabled by the program.

Four of the pins on Port C may be either handshake control signals or Port C parallel pins. Port C thus has a pin function select register that allows the processor program to establish which pins serve handshaking rather than parallel I/O purposes. Many system applications need only parallel interface pins and can thus disable the handshake features.

Port Handshake Control

The four handshaking control signals on Port C may serve the following functions:

A. Digital inputs
B. Digital outputs
C. Interrupt inputs
D. A latch enable for Port A input data
E. An output pulse when Port A to read
F. An output pulse when Port B is written

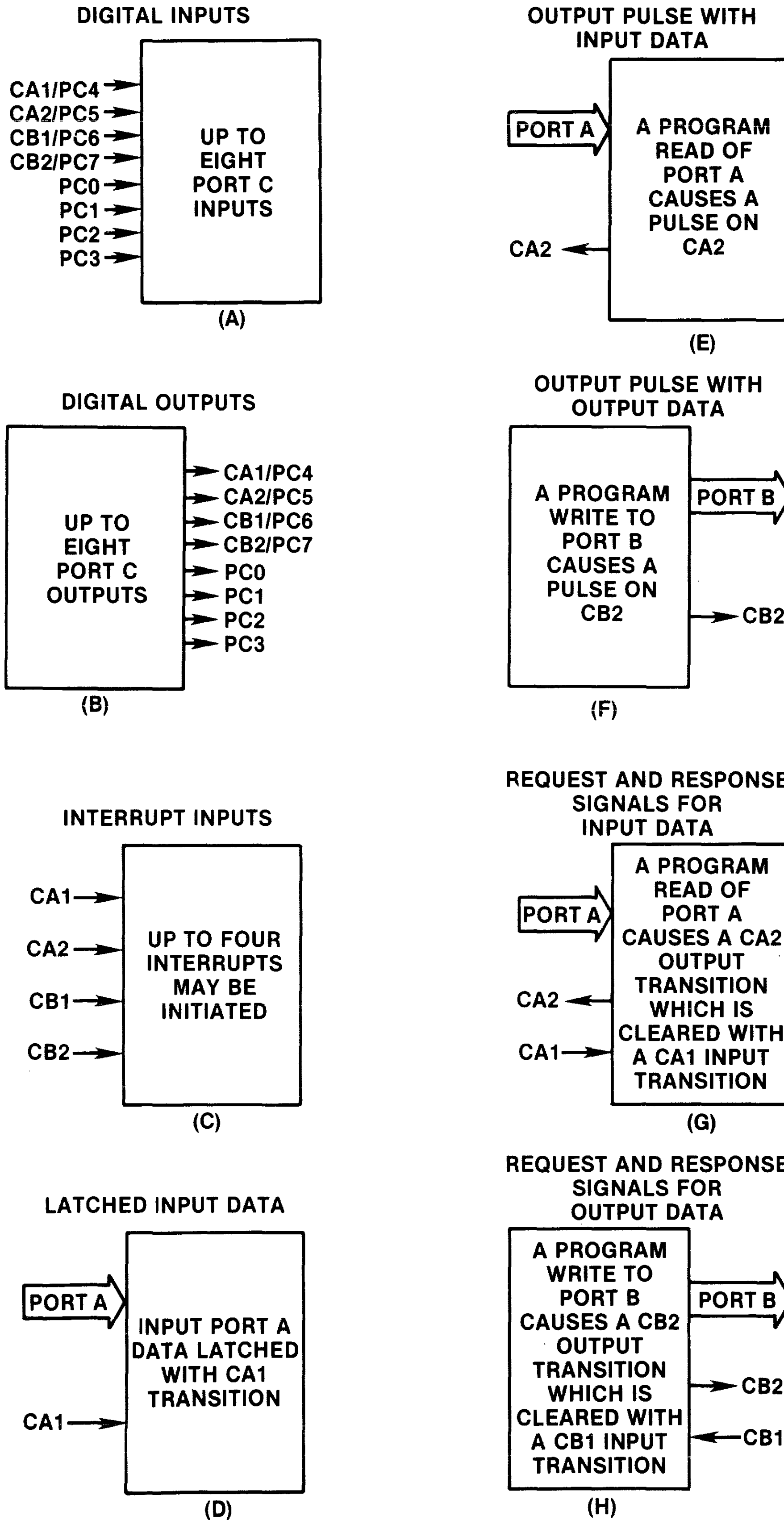


Figure 11—Port handshaking modes

G. Request and response signals for Port A input data
H. Request and response signals for Port B output data

A series of compound combinations of the above functions are useful. For example, Items D, C, and E allow an external source to write a byte into the MC146823, which initiates an interrupt; then, when the program reads the data, a pulse is returned to the external source. Figure 11 shows the eight handshaking functions graphically. The digital input and output modes are the Port C usage as already discussed.

Part C of Figure 11 shows that all four handshake pins may initiate interrupts. Each interrupt is separately enabled; control bits are provided by the program in control registers and separately identified in a status register available to the program. Interrupt overrun is also separately indicated in a status register for consecutive interupts that are not serviced fast enough. The four interrupt functions are ORed together onto the one output interrupt pin to the processor.

Function D in Figure 11 allows an externally provided signal edge to latch input data into Port A. This is a convenient way to accept asynchronous data bytes from a serial I/O, another processor, a mechanical peripheral, or other parts.

In Parts E and F of Figure 11, a program read or write causes the Parallel Interface to send a pulse one bus cycle wide. In the case of a Port A read, the output pulse is a response signal. The output pulse with a Port B write allows a byte of data to be latched into external hardware.

Closed-loop handshaking is provided with Functions G and H. One input and one output handshake control pin is associated with input data on Port A and two separate pins handshake with Port B output data. One signal requests data flow with an edge; the other signal responds with an edge on the other pin. One use for handshaking interlinking like this is to interconnect two processors.

There is not enough space in this report to look at all the useful combinations of the eight modes outlined above.

Bus Interface

The bus interface consists of eight bidirectional bus pins and four input control pins.

During the first portion of the bus cycle the 8-bit bus includes four address bits to select one of the 15 addressable MC146823 register locations. During the latter portion of the bus cycle, the processor provides a write data byte or the peripheral provides a read data byte.

A Chip Enable (CE) input pin tells the peripheral to accept or ignore the current bus cycle. As such, CE must be true after the address is stable and remain until the data is transferred. The Address Strobe pin allows the address on the bus to be latched within the peripheral.

Figure 12 shows the above bus functions as well as the two interpretations of the other two bus control pins. The two logical interpretions are called the MOTEL concept (for MOTorola and IntEL compatible). This allows direct connection to processors, creating control signals in either de facto bus standard.

In the Motorola MOTEL mode the DS input is a positive pulse during the data portion of each bus cycle. The R/$\overline{W}$ pin indicates during the DS pulse whether a read or a write cycle is in progress. The other MOTEL interpretation is a low-going

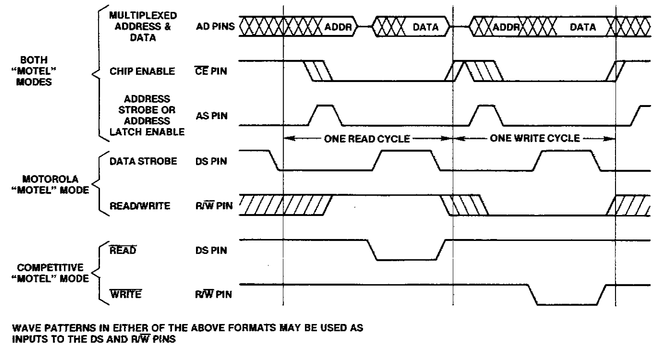


Figure 12—Bus interface signals

read pulse on the DS pin and a low-going write pulse on the R/$\overline{W}$ pin. The peripheral automatically decides which interpretation to use by sampling the DS pin at the time of the AS pulse.

Used With Any CMOS Microprocessor

The MOTEL bus interface concept allows a peripheral or memory IC to interface directly with any new-generation multiplexed bus microprocessor. The MC146823 peripheral was designed for use with the MC146805E2 processor. But the MOTEL concept allows it to also be used with other CMOS microprocessor and expandable single-chip microcomputers. Figure 13 shows the interface on a Motorola type of bus, while

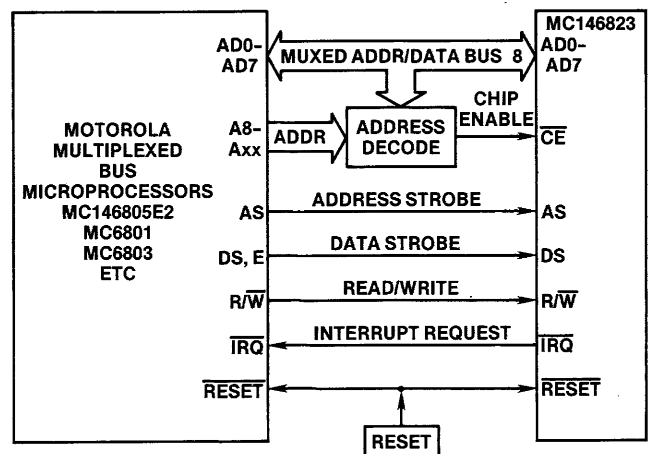


Figure 13—Typical processor interface, Motorola bus

Figure 14 shows the same peripheral directly connected to other processors. No intervening glue is needed to adapt the peripheral to the other processor buses. Universal peripheral

and memory applicability has finally been achieved. (Incidentally, the MC146818 Real-Time Clock Plus RAM and the MCM65516 2K × 8 ROM also use the MOTEL concept.)

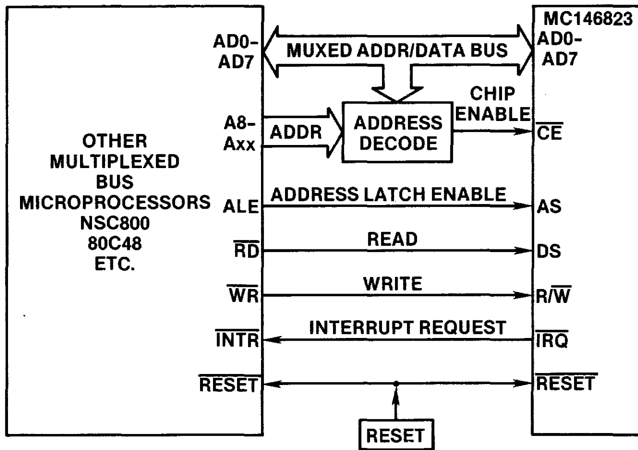


Figure 14—Typical processor interface, other bus

Memory-Mapped Registers

The port hardware functions and the bus interface capabilities of the MC146823 have been reviewed. The remaining element is the program use of the features.

From the program vantage point, the Parallel Interface is a block of 16 addressable locations, of which 15 are used. Figure 15 lists the seven major functions, shows which of the three ports the function is used with, and lists the hexidecimal address of the register within the 16 addressable locations.

FUNCTION	HEX ADDRESSES PORT A	PORT B	PORT C
PORT DATA REGISTER	2	3	4
PORT DATA DIRECTION REGISTER	6	7	8
PIN FUNCTION SELECT REGISTER	—	—	B
HANDSHAKE CONTROL REGISTER	9	A	—
PORT DATA REG./CLEAR INTERRUPTS	0, 1	C, D	—
INTERRUPT STATUS REGISTER		E	
OVER-RUN WARNING REGISTER		F	

Figure 15—Register functions

The port data registers are read/write locations for each of the three ports. Each port also has a read/write data direction register to establish pins as inputs or outputs. Port C has a 4-bit function select register to individually enable the handshaking function.

Two 5-bit registers establish the handshaking modes for Ports A and B. These bits establish whether the active input edge of the handshake function is a rising or a falling edge. Each of the four interrupts include an enable bit. The input latching, output pulsing, and closed-loop response functions are also established with bits in these registers.

A read-only status register contains flag bits for each of the four interrupt sources. Each flag bit represents a status condition denoting whether the corresponding interrupt has been enabled by the program. When a status flag is set and the corresponding interrupt has been enabled, the interrupt pin is asserted and an interrupt OR bit appears in the status register.

The program clears an interrupt by reading or writing a specific port data byte. When a handshaking byte transfer is to be acknowledged, the data read and write do so automatically. But in many cases, the data port is not directly associated with the interrupt function. In such cases a port read and write could cause a pending interrupt to be lost. Therefore the ports can also be read and written without effecting the interrupts. Three port data addresses are associated with each of the two handshake ports. One address does not affect the interrupt flags. The second address clears the interrupt status associated with one interrupt source as well as performing the data transfer. The third address does the same thing with the second port interrupt. An interrupt can thus be cleared with a psuedo-read of a port. The test (TST) instruction in the MC146805E2 does so without disturbing the accumulator.

The last addressable location is the overrun warning register, which indicates that a previous interrupt had not yet been serviced when a new one appeared. The 4-bit read-only register allows the program to find out that one or more events or data elements have been lost.

Feature-Packed Parallel Interface

Comparisons can be made to the popular N-channel parallel interfaces to see that the MC146823 includes more features than most. Three full 8-bit ports are included in a 40-in package where some others have fewer bits. The program establishes every bit separately as an input or an output, rather than establishing direction by groups of bits. Four separately selected interrupts are included, and the interrupts may be separately cleared. All registers are directly accessed by the program; none are hidden. Many handshake modes are included to allow easy interfacing to existing equipment.

CMOS IS NOW A PRACTICAL MICROPROCESSOR TECHNOLOGY

Many have waited a long while for full-performance microprocessor systems to be implementable entirely in CMOS. The CMOS MPU era has finally arrived. The microprocessors and memories are available. This report has shown that the needed parallel interface peripheral function is now covered, and the gate and flip-flop glue functions needed in larger MPU systems are also now practical in CMOS.

The MC68000 family and distributed processing

by JOHN F. STOCKTON
Motorola Semiconductor Inc.
Austin, Texas

ABSTRACT

The key philosophy today is to build parts that will be upward compatible with multiple processor systems of the future so that there is a migration path from existing single-bus systems to the higher-performance, multiple-local-bus systems of the future. An important parameter of these systems will be system performance, and the need for this performance is increasing faster than vendors can increase single-processor performance.

The need for multiple-processor systems is clear in the future. Knowing this, the designers of the MC68000 made sure to include all the necessary hooks into the processor design to support multiple processor architecture in the future. Some features of the existing processor that might not be used often today will become very important to future members of the MC68000 peripheral family. Some of these features and systems will be discussed here.

THE NEED FOR DISTRIBUTED PROCESSING

As office-oriented computer systems become more user-friendly, and as more of the operating systems and applications programs are written in high-level languages, there is a much higher demand placed on microprocessor vendors to keep offering ever increasing amounts of performance for approximately the same cost as before. To meet this higher performance requirement, microprocessor vendors cannot simply rely on single-bus structures to keep increasing performance. The solution to increasing performance will be to rely heavily on multiple processors, each having its own local bus, operating independently. To take advantage of this solution, microprocessor vendors must build in this upgradability early in the design of their microprocessor families.

FAMILY PHILOSOPHIES

The key philosophy today is to build parts that will be upward compatible with multiple-processor systems of the future so that there is a migration path from existing single-bus systems to the higher-performance, multiple-local-bus systems of the future. The key parameter of systems of the future will be system performance, and the need for this performance is increasing faster than single-bus processor systems can increase their performance. The need for multiple-processor systems is clear in the future. Knowing this, the designers of the MC68000 made sure to include all of the necessary hooks into the processor design to support multiple-processor architectures in the future. Some features of the existing processor that might be used often today will become very important to future members of the MC68000 peripheral family. Some of these things will be discussed specifically here.

The MC68000 was specifically designed to support high-level languages; the register set of the processor was intentionally kept general-purpose, with no dedicated registers that compilers have a difficult time using. Each register was defined so that it could be used as a pointer register as well as a data register. Special-purpose instructions were added to increase efficiency of procedure calls and re-entrant routines. These instructions were the "link," "unlink," "load effective address," and "push effective address" instructions. As well, instructions were added to streamline context switches via the "move multiple" instruction, which can stack any portion of the register set onto the stack with one single instruction. Operating system support was also an important design consideration in the design of the MC68000. Distinctions like user/supervisor separation were included to help increase system reliability without a large amount of software overhead. Another important feature is the "TEST-AND-SET" instruction, which allows for truly indivisible read-modify-write cycles on the 68000 local bus, even when there are multiple bus masters. This instruction depends heavily on the asynchronous nature of the 68000 bus, since it is possible to lock out other accesses by maintaining ownership of the bus control lines. Because of this, it is possible to keep other bus masters off and make the read-modify-write cycle truly indivisible.

Another important philosophy was that the processor extensively checks to insure that only legal instructions are being executed, that word operations occur on word boundaries, and that users do not try to execute privileged instructions.

The new family of peripherals will also consistently support these philosophies. One very important philosophy that the 68000 supports is the notion of an address space. The function code lines and the bus grant acknowledge (BGACK) lines form four additional address lines that are used to indicate which address space is currently being used. The Memory Management Unit (MC68451) uses these function codes to provide translation and protection according to the current address space in use. These function codes are shown in Figure 1. The advantage of using these is that all transfers can take place in logical space and be mapped and privilege-checked by the Memory Management Unit, thus increasing system reliability.

The family of peripherals will all consistently support the asynchronous bus structure of the 68000 as well. These philosophies will allow systems to be built with the MC68000 family

FC3	FC2	FC1	FC0	STATE
0	0	0	0	RESERVED
0	0	0	1	USER PGM
0	0	1	0	USER DATA
0	0	1	1	RESERVED
0	1	0	0	RESERVED
0	1	0	1	SUPV PGM
0	1	1	0	SUPV DATA
0	1	1	1	IAK CODE
1	0	0	0	BUS SLAVE SPACE
1	0	0	1	" " "
1	0	1	0	" " "
1	0	1	1	" " "
1	1	0	0	" " "
1	1	0	1	" " "
1	1	1	0	" " "
1	1	1	1	" " "

Figure 1—Support of bus masters and bus slaves in logical and physical memory space by function codes supported by the 68000 family

that will be upward expandable and not require redesigning to keep increasing performance in the future.

In the future, as mentioned previously, the two things that will need processor performance will be high-level languages, and user-friendly software. The high-level languages place a high demand on systems because of their inefficiencies relative to assembly language programming and the protection and checking that they offer at run time. A fairly efficient compiler today still produces between 2 and 2.5 times as much code as a comparable program written in assembly languages. Many times the compiler-generated code can be optimized, but the ratio still rarely drops below 2. As more of the operating systems are written in high-level languages, these inefficiencies are carried along and compounded, since both the application program and the operating system are much larger than they need to be. These performance degradations are the cost of easing demands on programmers and making software more portable. The other thing that will affect system performance will be user-friendly software, which has extensive error checking/recovery, and user aids in the sense of online documentation and "help" commands. These things were not a problem previously, because the available processors simply did not have enough performance. Now microcomputers offer performance comparible to minis and low-end mainframes, so it is reasonable to employ these practices. The problem now is that the growth of inefficiency is faster than the increase of system performance offered by microprocessor vendors. The solution is to get on a faster performance growth curve than single-processor systems can offer.

The way to do this is obviously to depend on multiple processors, each having its own local resources, and a communication mechanism between each two elements. To take full advantage of this, the problem being solved must be highly parallel; fortunately, today in the office environment, the problems are fairly parallel. Additionally, in an effort to reduce the cost of CRT terminals, by using microprocessors with local resources as the heart of the CRT controller design, the basis of a distributed processing system has been established. Tightly coupled multiprocessor systems will be developed for solving specific problems that are limited in scope, and both moderately and loosely coupled systems will be developed because of economic pressures.

WAYS TO SOLVE THE PERFORMANCE PROBLEM

As previously mentioned, multiple processors will offer the raw performance required to do the job in the future, but their interconnection topology is a critical issue. The basic three ways to use multiple processors are (1) tightly coupled, (2) moderately coupled, and (3) loosely coupled. The tightly coupled systems typically share one instruction bus and rely on each processor's taking a large number of internal cycles for each external (bus) cycle. The moderately coupled systems have multiple processors, each with its own resources on its own local bus, and depend on some mechanism for communications with the rest of the system. Usually this mechanism is a high-speed DMA channel or a dual-ported mailbox. Each solution offers a fairly high bandwidth communications channel to the processor. The loosely coupled processors depend again on multiple processors, each with its own resources; but this time the communications mechanism is a serial data communications link, which typically has about one-tenth of the bandwidth of the moderately coupled solution. The loosely coupled solution does have the advantages of allowing each processing node to be some distance from the other nodes.

WHY NOT TIGHTLY COUPLED SYSTEMS?

There are several disadvantages to tightly coupled multiprocessor systems, the main one being that the system quickly becomes bus-bound. Figure 2 shows a typical tightly coupled system block diagram. Each processor added competes for an ever smaller percentage of the available bus bandwidth until there is none left. An example of this would be to try to tightly couple two MC68000s. Each MC68000 executes on the average 5 cycles internally for each 4 external cycles. The fact that the average instruction time is close to what the bus cycle time is means that one 68000 uses between 80% and 90% of the available bus bandwidth. The MC68000 makes better use of the bus than many other processors, and because each processor will try to get as much of the available bandwidth as possible, the addition of a second processor on the bus would allow it to have a maximum of 20% of the bus bandwidth. The second processor would at best be running at 25% of the throughput that it could have if it were on its own local bus. The net improvement in performance resulting from the addition of the second processor would be at best a 25% increase, and more than likely would not be more than 10% because of bus arbitration overhead. In some instances it does make sense to tightly couple processors on one local bus, but this is the case when the second processor can execute some particular instructions much faster than the current processor. An example of this would be the addition of a floating-point coprocessor, which can do floating-point calculations an order of magnitude faster than the current MC68000 can. The effect on performance is positive in this instance rather than negative because there is an inherent isolation in what each processor would be trying to do, so processors would not compete heavily for the bus. The guideline for deciding to add a coprocessor to the system should be that the problem be isolated well enough that the communications overhead would not be more than 10% of the total time taken to solve the problem. This insures that the additional performance of the dedicated coprocessor is not offset by the communications overhead of the addition.

The trend in the future will be for processors to take fewer cycles on the average for each instruction, thus trying to occupy 100% of the available bus bandwidth. When the processor has instructions that execute that quickly, performance improvements must then come from providing more memory bandwidth. This is usually done by either adding a hierarchial memory scheme or widening the bus interface. These trends will help the problem; but the message is still clear that the bus is a scarce resource and that the way to get more performance in the future is not to try to tightly couple processors, except when they do not compete for memory bandwidth resources.

Another way of solving the performance problem is to depend on a loosely coupled network of processors, all commu-

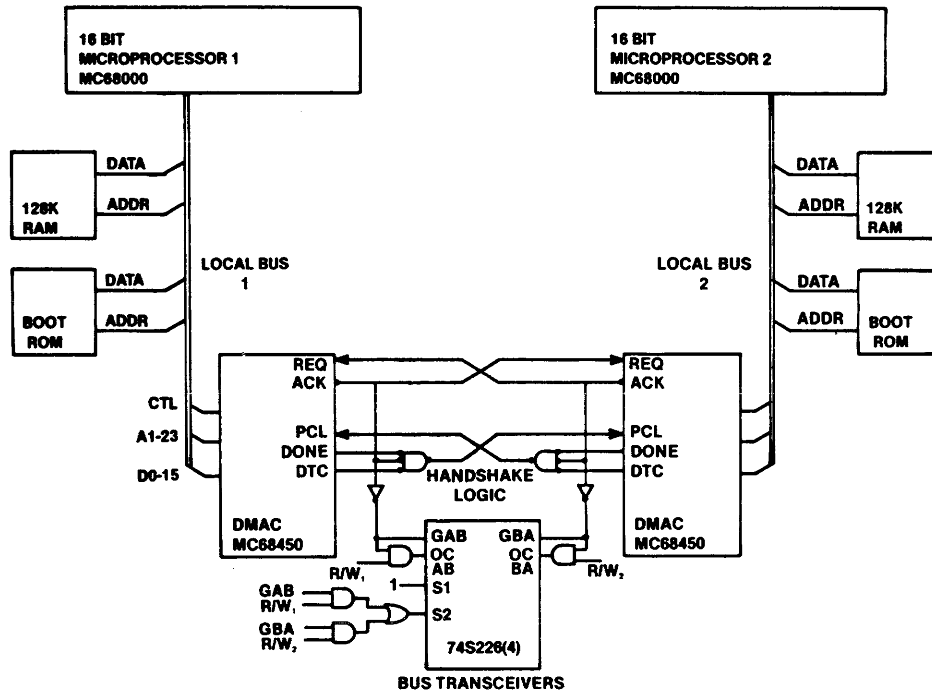


Figure 2—Tightly coupled multiple 68000 system

nicating over a serial data communications link. This topology works particularly well when the problem to be solved is highly parallel and isolated, with a low communications requirement. An example of this would be a distributed word processing system, where editing is done locally, with a local processor and local memory resources, and a datacom line to link the work station to a file server or a printer. The system performance in this instance is much higher than previously, because the problem is parallel enough to allow concurrent operations. It is intuitively known that this solution works best in applications like distributed word processors but starts to suffer from contention problems when the application is heavily dependent on the distant resources. An example of this second application might be an airline reservation system, where the time spent editing the data locally is small in comparison to the time required to transmit it. Figure 3 is a block diagram of a loosely coupled multiprocessor system.

Yet another solution has the advantages of the loosely coupled topology and does not suffer from the low bandwidth interconnection between processing elements. This is the moderately coupled system. In this instance each processor is still on its own local bus, but the interconnection to the other processors is done through either a dual-port RAM or a DMA channel. This approach again lends itself to problems that are inherently concurrent, but does not suffer as much when the problems are communications-dependent. Figure 4 shows a typical moderately coupled multiprocessor system.

Motorola has two products that depend on this topology to allow for concurrent processing. These products are the MC68120 Intelligent Peripheral Controller and the MC68122 Cluster Terminal Controller.

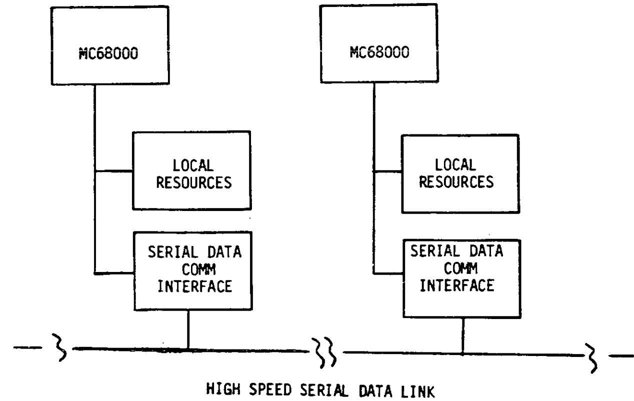


Figure 3—Loosely coupled processor topology

THE MC68120 INTELLIGENT PERIPHERAL CONTROLLER

The MC68120 Intelligent Peripheral Controller is a general-purpose peripheral controller that consists of an 8-bit CPU, 2 Kbytes of read-only-memory, 128 bytes of RAM, a 16-bit timer, a serial communications interface, and 23 parallel I/O lines. These I/O lines can be used to connect to peripherals directly, or, more importantly, can be used to form an MC6800-type bus that can be used for general-purpose I/O processing. With the 68120 in this mode, I/O burdens can be removed from the central CPU and more time can be devoted to instruction processing, resulting in increased performance

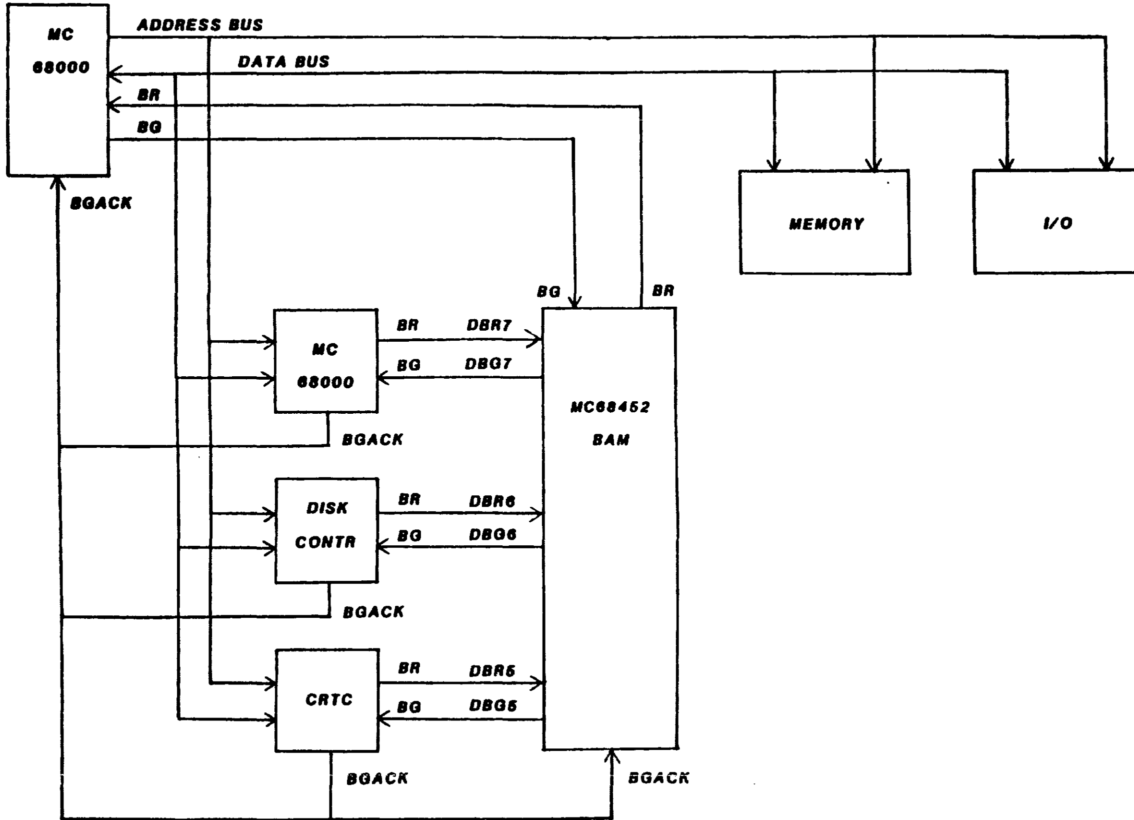


Figure 4—Moderately coupled multiprocessor topology

due to the parallelism. Communications is done through the dual-port RAM that is on board the 68120, and access permission is controlled via the six semaphore registers. Each register contains a bit that indicates whether the resources it describes are currently in use and a bit that identifies which processor (master or slave) used it last. These registers are set up under software control to correspond to common resources between the MC68000 and the MC68120 and are not strictly limited to the dual-port RAM.

Figure 5 is a block diagram of the MC68120 connected in a system with a private bus, acting as an I/O processor.

The Cluster Terminal Controller

The MC68122 Cluster Terminal Controller is an example of an MC68120 that has been programmed to act as an interface processor between a cluster of terminals and a host processor. The CTC uses the private bus to communicate with multiple Asynchronous Communications Interface Adapters and the dual-port RAM as a message buffer. The CTC can support four terminals at 9600 baud, or as many as 32 at 300 baud. This restriction comes about as a result of using the dual-ported RAM as a mailbox mechanism. If the mailbox were larger, a correspondingly larger number of terminals could be supported; however, it was found that this ratio of terminals to processors was quite acceptable. The performance advantage

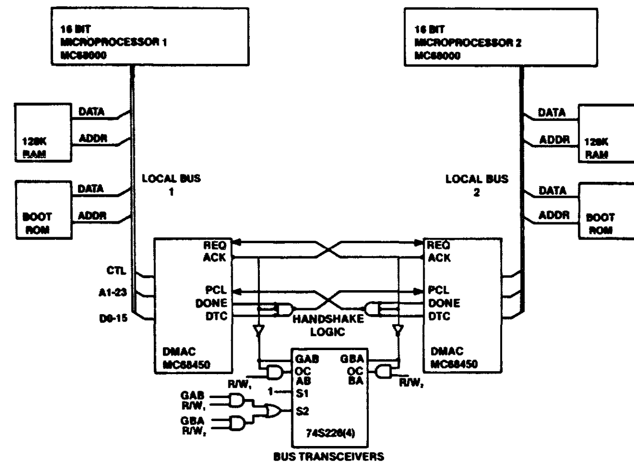


Figure 5—System block diagram of an MC68120 being used as an I/O processor

is obvious, since now the Cluster Terminal Controller has effectively reduced the number of interrupts to the host system from around 4000 per second to 60 per second. Assuming that the interrupt latency of the 68000 system was around 30 microseconds per interrupt and the return overhead was around 20 microseconds per interrupt, the operating system

overhead can be reduced from 19% to 3% (this calculation assumed four terminals each running at 9600 baud, shipping average buffers of 64 characters through the dual-port RAM buffer).

The performance increase speaks for itself in this instance. Figure 6 shows the Cluster Terminal Controller in a typical system environment.

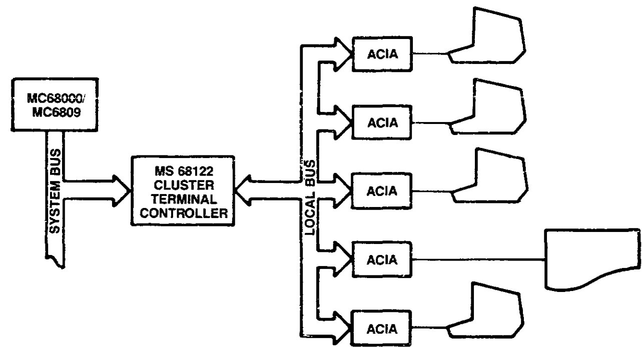


Figure 6—Block diagram of the CTC system

SUMMARY

In summary, the processors and peripherals of tomorrow will be more performance-oriented and will have to be well thought out so that they can be upwardly expanded without requiring a major system redesign. In this situation the customer will be in a critical position, since it will become increasingly more difficult to mix vendors' parts and the vendor will have a stronger influence over the customer's system. For these reasons the customer should give special consideration to the vendor chosen to make sure that there is a consistent, well-thought-out growth path from current products to the products of the future.

Using operational standards to enhance system performance*

by DAVID R. VINCENT
Boole & Babbage
Sunnyvale, California

ABSTRACT

There are only three reasons for a data center to vary from service level objectives, i.e., volume, mix, and efficiency. The first two are aspects of user behavior, while only the third is under the full control of the data center. With the proliferation of online systems, user behavior is affecting the data center in *realtime*. Management processes, technology and software tools exist to provide the basis for scientific management of the data center. Key to this endeavor is the ability to model the system and present in graphic form the relationship between the system and user behavior characteristics. This paper points out the source of data, existing software tools, and graphic methods and includes data and the results from a study and simulation of a data center.

*Published in the *1981 CMG-12 Conference Proceedings,* New Orleans, December 1981. Phoenix: Computer Measurement Group, Inc., 1981.

One of the most outstanding individuals to come out of the American Industrial Revolution was a young man by the name of Frederick W. Taylor.† Taylor's "scientific management" had a large effect on the tremendous surge of affluence we have experienced in the last seventy years which has lifted the working masses in the developed countries well above any level experienced before, even that of kings and queens of old. Taylor's analysis of work and resultant method improvements resulted in production gains unequaled in history.

His analysis of work in the 1880's was exemplified by his study of shoveling iron ore in a steel mill. In this study, he found gross inefficiencies in the actual process of shoveling. By breaking the motions and actions of the workers down into measurable segments, he was able to develop better work methods (something like tuning a computer) and standards of performance so that output could be judged on a day-to-day basis.

This work was finally completed, as we now know it, shortly after he passed on during World War I, at which time American industry in general began to adopt his principles. Some of Taylor's disciples carried on his work thereafter; notably, Frank and Lillian Gilbreath (who were the subject of a movie "Cheaper by the Dozen") and Henry Gantt, who initiated project scheduling in a Gantt Chart. The common interest uniting those people was the analysis of work and translating that analysis into more productive and efficient procedures and flows of work.

The analysis of work involves the following:

1. The identification of all processes necessary to produce an end product or result
2. The rational (and manageable) organization of the sequence of operations so as to make possible the optimal flow of work
3. The analysis of each individual operation or process including measurement and historical trending
4. The integration of the above into an overall process of producing a product or result

This established process has been used successfully for decades in American industry (and Japanese, and German, and . . .). It is my contention that the size and complexity of the data center has now evolved to a point where this methodology may be applied in greater depth to realize significant economic benefits.

The role of the data center in the organization to which it belongs has increased from simple payroll and accounting applications in the 60's (remember "tab runs"); through inventory and distribution systems in the 70's; to online, real-time management and operational applications in the 80's. A good example of the online operational application of the 80's is the automated teller systems in banks. In this case, the data center has gone to a point where it actually interacts with bank customers.

As the data center evolves towards a utility-like process, the end product of the data center is service. With the proliferation of online systems, service has become highly visible to the user and a much more tangible element in top management consideration. In fact, service levels have become so visible to the world outside the data center that, to a large extent, they are considered *the* measure of the data center's performance. The perception of service falls into three basic categories:‡

1. Online response time
2. Batch turnaround
3. Availability

When one of these is outside a user's expectation, the DP manager's phone begins to ring with compliants. Moreover, the business environment in which the DP manager now lives is one that expects him to manage his operation the same as any other functional unit of the enterprise. At this point, many DP managers are beginning to feel the strain of trying to negotiate and maintain service levels without the benefit of having fully implemented *traditional* scientific management principles in their data centers.

One major element of scientific management is the need to understand the elements of the service to be performed and the variables that can affect them. These variables have one of two sources: data center operation or user behavior. There are many data centers that have not yet quantified the difference in, say, response time caused by data center inefficiency as opposed to that caused by user behavior. This is because the analysis of work has not been related to these two factors.

These data centers are usually perceived by their users as being poorly run because all variances from negotiated response times are attributed to the inefficiency of the data center. When analyzing work, there are in fact only three universal causes of deviation from a standard which will cause response time to be better or worse than plan:

1. Volume
2. Mix
3. Efficiency

†Actually, Taylor was preceded in philosophy by another "irascible genius," Charles Babbage, who documented some of the earliest forms of scientific management in his most successful book, *On the Economy of Machinery and Manufactures (1832)*. However, it took Taylor to rediscover and implement practical scientific management in America.

‡These were the first three items listed in a poll done in Arizona with 150 DP managers. Appendix A contains a full list prepared during the *1981 EDP Performance Management Conference*.

TABLE I—Workload utilization report

Job	Workload *P*	Workload *A*	Workload *V*	Standard work units (000)	Total SWU's (000) *P*	Total SWU's (000) *A*	Total SWU's (000) *V*
A123	4	6	(2)	170	680	1020	(340)
B357	8	3	5	80	640	240	400
C896	10	15	(5)	300	3000	4500	(1500)
Totals	22	24	(2)		4320	5760	(1440)

The financial term for these deviations is *variance* and simply means the difference between planned versus actual results. The first two variances are attributable to user behavior. The third variance is the only one that can be attributed to the operating of the data center. In essence, the planning and anticipation of the volume and mix considerations (user behavior) are much of what capacity management is all about.

In this paper, I will explore methods of determining the relationship of volume, mix, and efficiency in the current system and how we might predict performance at various levels. The goal will be to define performance curves giving standard levels of service at varying levels of volume and mix.

Volume is a measure of activity in terms of jobs or transactions by job or transaction type. The various jobs or transactions in types result from the differences in computer resources consumed to process that particular job or transaction. Differences in volume can be measured by comparing forecast or plan to actual. For example, the running of Job A123 resulted in the following volumes:

Job	Plan times run	Actual times run	Variance (Plan–actual)
A123	4	6	(2)

The volume variance in this case is expressed as 2 jobs over plan. In terms of resources expressed in standard work units** it is 2 times planned resource per run of Job A123. Let's say Job A123 should use:

	Units (000)
CPU standard work units	= 170
I/O	*
Memory	*
Total standard work units	= 170

Therefore, the volume variance is 2 times 170,000 SWU's units or 340,000. This may also be expressed in terms of dollars if a standard cost per SWU's can be calculated and applied.

Now let's take an example where there is a volume *and mix* variance. Let's suppose that we have 22 jobs that require a total of 4,320,000 SWU's (an average of 196,364 per job).

**For this example, a standard work unit will be considered for the case of CPU only. The standard work unit is basically CPU time factored for the relative power of the CPU.

The actual activity turns out to be 24 jobs requiring a total of 5,760,000 SWU's for a variance of 1,440,000. The matrix of the above with additional information is then constructed as shown in Table I.

The volume variance is calculated:

$$\text{Variance of job runs} \times \frac{\text{Planned SWU's}}{\text{Planned Workload}} = \text{Volume Variance}$$

The volume variance is calculated:

$$\text{Variance of Job Runs} \times \frac{\text{Planned SWU's}}{\text{Planned Workload}} = \text{Volume Variance}$$

1. The variance of job runs is calculated by subtracting the actual total jobs run (24) from the plan (24) with a resultant 2 jobs run over plan. The brackets indicate that this variance will have an unfavorable impact on data center performance.
2. Planned SWU's divided by planned workload is in fact the planned rate of SWU consumption for a job from this workload.
3. The volume variance is then a function of the jobs run over or under plan times the planned resource (SWU) consumption rate. This isolates those resources consumed over or under plan as a result of users running a different number of jobs than planned.
4. The numeric calculation of volume variance is then:

$$(2) \times \frac{4320}{22} = (392.8)$$

The mix variance is calculated†† as follows:

1. The planned SWU rate $\left(\frac{\text{Planned SWU's}}{\text{Planned Workload}}\right)$ was just calculated for use in the volume variance.
2. The actual SWU rate is the key element in the calculation of the mix variance because the difference between the planned and actual SWU rates is a result of a workload with different elements.
3. The difference in the SWU rates is multiplied by the actual workload.
4. The numeric calculation of this example is then:

$$\text{mix variance} = \left(\frac{4320}{22} - \frac{5760}{24}\right) \times 24 = (1047.2)$$

The mix variance is the amount of standard work units consumed over or under plan as a result of having differ-

††In this case, the SWU's per job are not varied. As can be seen in the next example, they are varied and this will result in an efficiency variance. The formula for calculating mix variance would be modified to reflect the fact that both mix and efficiency variance contribute to the difference between the planned and actual SWU rate. The revised formula would then be:

$$\frac{\text{Planned SWU's}}{\text{Planned Workload}} - \frac{\text{Actual SWU's}}{\text{Actual Workload}} \times \text{Actual Units}$$

$$-\text{Efficiency Variance} = \text{Mix Variance}$$

TABLE II—Workload utilization report

Job	Workload P	Workload A	Workload V	Standard work units (000) P	Standard work units (000) A	Standard work units (000) V	Total SWU's (000) P	Total SWU's (000) A	Total SWU's (000) V
A123	4	6	(2)	170	165	5	680	990	(310)
B357	8	3	5	80	100	(20)	640	300	340
C896	10	15	(5)	300	500	(200)	3000	7500	(4500)
Total	122	24	(2)				4320	8790	(4470)

ent jobs run than were planned which, in turn, consumed different amounts of resources. The summary of the variance is then:

Volume Variance	(392.8)
Mix Variance	(1047.2)
Total Variance	(1440.0)

Now we can expand the same example to demonstrate an efficiency variance. Let's say that we have the data in Table II. In this case, the SWU's consumed per job were different than planned. Since we have kept everything the same except the actual SWU's per job, the volume and mix variances are exactly as previously calculated.

The efficiency variance is calculated by:

$$\text{(Planned Job SWU}_1 - \text{Actual Job SWU}_1) \times \text{Actual Jobs}_1 = \text{Efficiency Variance}_1 + \text{(Planned Job SWU}_n - \text{Actual Job SWU}_n) \times \text{Actual Jobs}_n = \text{Efficiency Variance}_n$$

1. The total efficiency variance is a sum of the individual efficiency variances calculated for each job run.
2. Each job variance is calculated by multiplying the actual runs of the job times the job's SWU variance (planned minus actual SWU's to process each run of the job).
3. The calculation would then be:

Job	Actual times job ran	SWU Variance	Efficiency Variance
A123	6	5	30
B357	3	(20)	(60)
C896	15	(200)	(3000)
Total Efficiency Variance			(3030)

The new total variance summary is then:

Volume variance	(392.8)
Mix variance	(1047.2)
Efficiency variance	(3030.0)
	(4470.0)

This says that the data center performance of running these jobs, especially Job C896, suffered either because of (1) a bad application program, (2) some system deficiency, (3) a bad estimate of what it would take to run the job, or (4) a bit of all three.

It should be noted that user behavior caused 32% of the total variance. Often, the user behavior element contributes even more to the total variance, especially at peak periods.

Where can SWU's be obtained? Most computer systems have some kind of log that accumulates resource usage. CPU time is the easiest measure, but relative CPU power differences dictate that adding pure CPU time from different CPU's might be erroneous over time. IBM has offered a solution with MVS by providing service units which are internally calculated. These service units are indeed CPU time times an internal power factor which result in theoretically compatible service units over a variety of IBM CPU's.

The Institute for Software Engineering has also provided a great deal of literature in this area which deals with software physics. The direction of this work is somewhat similar to the service unit methods built into IBM and PCM systems, but is much more complex.

So far, the analysis of variance has focused on service units or the amount of work going through a data center. How do these translate to levels of service?

The data processing work plan basically consists of *putting out the required volume and mix of work* as a basic requirement, and *on a timely basis* as a second but equally important requirement in most shops. We can be sure that capacity has been exceeded when the data center physically cannot process the required workload even if it were to run a full three shifts per day, seven days a week. But below this level, there are other considerations that become practical limitations of available hours to do data processing work. The variances analyzed earlier in this paper will affect the timeliness of the data turnaround, especially in online systems. Plans of user workload are especially important during the peak online requirement that occurs during the normal five-day work week. This is especially important if the data center is very involved in the basic business of the company with which it works, such as a bank or department store.

What we're really talking about here are the end user's work schedules and how that affects the ability of the data center to plan and deliver services matching their schedules as

postulated in the *User Behavior Elasticity Theorem,* ‡‡ which states that *the degree to which the data center can influence end-user behavior is inversely proportional to the degree that data processing is involved in the basic business of the organization.* This can be illustrated by two examples.

The first is in the banking industry where automated tellers are being implemented. The data center cannot influence the end users of this application to any noticeable degree because the bank's basic business depends upon having the automated tellers operating when *its* clients (e.g., depositors, withdrawers) want to make a transaction. Chargeback schemes, management pressure, and the like will have little or no effect. On the other hand, a company producing buggy whips that has not integrated data processing into its basic business will tend to be much more flexible in terms of end-user behavior because production and distribution will continue whether the data center runs or not. Hence, it is more elastic as charging schemes and management pressure are applied.

We are also talking about the necessity for conscious management decisions regarding the economic benefits of achieving a given online response time versus the system costs that will be required to provide that level of response. Or, as in another case, the adding of another application to the online system in light of its potential impact on the response time of current users and applications may or may not be possible under the current configuration because of the impact it will have on the service required for other applications. This is where performance management comes in, specifically, the analysis of performance data. What we want to know is which user-controlled variables, i.e., volume and mix, will affect service level performance. Those of us who have been tracking this type of data over time know what happens when a CPU gets over 90% busy and are well aware of the exponential degradation of service levels that occurs beyond this level. This is also true for many other areas within the system, depending on where the system bottleneck is. Are there tools that will identify such sensitive areas in the system? If so, how might they be applied?

The software tools and sources of data needed to perform this kind of analysis are readily available. You probably have some of them installed on your system already.

For the purpose of this paper, I will deal with those products that operate in the IBM MVS environment. However, other systems collect data on a similar basis; so the concepts presented here will apply to other environments as well.

In 1972 IBM introduced MVS with almost no software tools for control. Even today, IBM provides only a minimum of such tools. However, as MVS (and such subsystems as TSO, IMS, and CICS) has matured and taken hold in the marketplace, a number of independent software firms have developed and are marketing many tools which are available for the data center to use. These tools also go a long way towards freeing the expensive and scarce systems programming staff to concentrate on day-to-day system optimization and productivity, rather than long-term system monitoring and capacity management.

‡‡This is my theorem developed from personal observations and many discussions about chargeback systems.

Since their initial introduction in the late 1960's, these tools have become easier to use and the outputs easier to interpret. The expertise of a veteran systems programmer is no longer required to implement effective performance controls.

Another major change is a conceptual one. In the beginning these control tools were introduced on a piecemeal basis. Now, however, many have been integrated into a cohesive architecture for capacity management at the system level. Exhibit 1 [Appemdix B] is an example of such an integrated approach with the following distinct levels of data center control activities:

Level 1— Systems programmers and operations personnel are working at this level on the day-to-day task of getting the work out and maintaining system availability and response to user-specified service levels. In this level of effort, realtime monitors show system and subsystem internal status so that problem areas may be detected and resolved. Early warning mechanisms driven by operator-defined thresholds simplify the task of identifying problem areas and systematically alert console operators. Another objective of this level is to optimize system performance by tuning.

Level 2—The objective of this level is aimed at the next step above monitoring system internals. The major concern is achieving end-user service levels and establishing the extent of system availability for processing the various types of work. The basic orientation of this level is towards the fulfillment of end-user response and availability requirements. Standards of performance derived from the configuration capability and user requirements are established and monitored here. Basic cost accounting concepts are applied at this level to track financial performance.

Level 3—This ultimate level is aimed at the DP manager's ability to predict the results of future workload and service level objectives based on various alternative hardware and software configurations. This is a level of vital concern to him because a job well done here will substantially increase his chances of success in future demand situations.

If effective and sufficient effort has taken place at all three levels outlined above, the data center manager will have a properly tuned system and will be ready to take the next step into capacity management. Let's stop here for a moment and cover these basic tools:

What Are Realtime System Monitors?

The word "realtime" is broadly defined to mean techniques relating to online display capability. In this case, a realtime monitor displays what is currently going on in the internals of the system. This is different from realtime inquiry to a data base of performance data that has been collected and can be displayed at any given time. A good realtime monitor will have the following major characteristics:

1. Early-warning mechanisms
2. User-defined threshold values
3. Clear and easy-to-understand screens
4. Graphic displays showing current system performance in current time periods
5. Menu-driven screen selection

6. Availability of comprehensive data for system event monitoring
7. Sufficient information for problem definitions, resolution, and corrective action

Early-warning mechanisms and user-defined threshold values will relieve the system programmer of the task of continuously monitoring the system and looking for areas of potential or actual problems. By having warning messages flashed to the console operator, the system programmer would be called in only in the case of an actual problem where his skills are needed.

For example, a master terminal operator using an IMS realtime monitor may get a warning that an important short-running transaction cannot be processed because a long-running job of lesser importance is using the data base record needed by the short-running transaction. In this case, the problem can be resolved by the operator rather than calling in an IMS system internalist. The operator may cancel the long-running job, thereby allowing the short-running transaction access to the data base, and then reschedule the long-running job to be run later.

On the other hand, the master terminal operator may get a warning that scheduling failures are up 30%. Now an IMS system internalist is needed to determine why. By using the realtime monitor and the screen menu, he may quickly call up that information he will be needing to resolve the problem. Clear and easy-to-understand screens along with graphic displays will assist the operator in defining problem areas when he calls for assistance. Menu-driven screen selection assists everyone using the monitor to get where they want to be in terms of displays. The availability of comprehensive data in sufficient detail will help to minimize problems when they occur. Exhibit 2 [Appendix B] shows some of the major realtime software monitoring products available in the market today.

What Are Continuous System Monitors?

Whereas realtime monitors show information on system internals as they happen, continuous monitors gather statistics on user-defined key variables all the time that the system is operating. These statistics are available for presentation as batch reports or may even be available on a realtime inquiry basis.

These statistics show how the system is being utilized over time and what kinds of demands are being made by the end users. A good continuous monitor will have the following prime characteristics:

1. Low overhead to operate
2. User-selectable areas to be monitored
3. User-defined units of work and/or centers of activity
4. Exception reporting
5. History file
6. Summary and detail reporting
7. Extensive graphics and management reporting
8. Batch reporting as well as realtime inquiry
9. Low maintenance

When acquiring a monitor, whether realtime or continuous, a significant consideration is what system overhead will be incurred as a result of installing the monitor. Some monitors in the market require substantial system overhead and, in effect, disturb what they are measuring. *The object is to increase performance, not further degrade the system.*

The system overhead to run an MVS/TSO monitor, for example, should not exceed 1%–2% in the continuous monitoring mode. Some monitors may also collect much more detailed data on an intermittent monitoring basis. In such an intermittent mode, the overhead should not exceed 5%. In the case of a subsystem monitor such as IMS or CICS, the overhead will depend on transaction volume but generally ranges from 5% to 10% of that particular subsystem.

By having user-selectable areas to be monitored with user-defined units of work or centers of activity, the monitor will collect data that end users can understand in those areas which need attention. For example, a user will generally understand such units as job or transaction and centers that relate to the accounting system. This means that the reporting may be in terms of turnaround or response time by work area.

Related to the user-defined variables, standards of performance that can be monitored will yield reporting on an exception basis. Why collect detailed data when the system is meeting standards? A history file of the collected data (both date and time stamped) will provide workload data over time. This will assist the data center management in characterizing the user workload and relating this to future system demand. For example, online peak periods and growth rate can be determined by the user, and response time for each daily time period can be reported. This data is the basis for a service level contract between the data center and the end user. Furthermore, it is measurable.

Summary and detail reporting enable the data to be presented to the system programmer or the operations manager, depending on the type or report needed. Extensive graphics and management reporting will be directed at service level performance and workload behavior.

Batch reporting as well as realtime inquiry allow immediate or day-after data analysis and reporting. Finally, because the monitors extract system data (and there are many techniques for doing this), low maintenance is a critical item. The product should be usable through releases of the operating system and its subsystems. Some of the major continuous monitoring software products are also shown in Exhibit 2 [Appendix B].

When the data center has analyzed the effects of user behavior and workload, the next logical step is to relate *them* to the capability of the system. Peak-period performance will probably be the main ingredient in any service agreement. The first step in predicting system capability is to know what the system can do now. By implementing performance reporting, there will be data relating to user workload characteristics. These can then be related to service level achievement.

First, we must assume that the system is properly tuned. A second assumption is that the workload has been shifted to the extent possible (i.e., batch work at night so as not to interfere with online work). At this point, the theorem of user behavior elasticity applies. The next logical step is to develop the operational standards of performance in terms of service levels

based on the current configuration and end-user service-level objectives.

In the case of predictive models, such questions as "What effect on service levels will be experienced by adding another channel or DASD device?" can be modeled and the results calculated against the current workload. Future anticipated workload growth can also be modeled to see future service level achievement with the current system. Shifts in volume and mix as compared to plan, when modeled, will illustrate service degradation caused by user behavior.

Alternative hardware and software options may be considered to find what is needed to maintain negotiated end-user service-level requirements. Additions to the current system or other alternative systems may also be modeled to measure the impact. Exhibit 2 [Appendix B] shows some of the models in use at present.

The major characteristics of a good predictive model are the following:

1. Results can be easily validated
2. Easy to use
3. Will distinguish the various hardware and software characteristics of available products
4. Can easily integrate user historical data to define workload characteristics
5. Economical to run
6. Easy to interpret reports

The first and foremost requirement is that the model be easy to validate. The value of a predictive model is in its prediction!

The second and very important requirement is that the model be easy to use. This will reduce the need for highly technical systems people or, at the very minimum, a mathematical theoretician to use the product. Rather, the model should be usable by a trained business analyst. Most errors made by models are created from erroneous input. Complicated models tend to be resplendent with such opportunities for error.

The predictive model ideally will easily incorporate hardware and software alternatives available to the data center, so that the analyst can play "what if?" games. That is, various configurations can be matched against various workload and service-level requirements to determine which configurations would be optimal. This type of analysis lends itself extremely well to the decision-making needed in the process of appropriating capital goods (i.e., data center hardware or software). From this data, various suitable configurations can be selected and the cost effectiveness of each evaluated. The cost of various user service levels may also be calculated. Both the capital and service analysis will assist in establishing the financial requirements for the data center.

It is important that the model be fed from a historical data base fed by the various system monitors so that workload data can be easily integrated. This may be used to define current standards of performance as well as defining trends for predictive modeling.

The model should be economical to run, that is, it should not consume much computer time. Since this process will be interactive and many passes of the model will be required to determine the new performance curve, each pass should run in a couple of minutes or less.

And finally, the output must be easy to interpret and readily presentable for management reporting. Again, a business analyst should be able to interpret the output and be able to input various alternatives as a result of the output from any given pass of the model.

The objective of this measuring analyzing, and modeling will be to derive performance curves that can become operational standards of performance that show the relationship of service-level achievement versus user behavior. Exhibit 3 shows the process involved in establishing such standards. It is an interactive process that involves tuning and end-user negotiations until finally an agreed standard of performance has been set. However, the standard is dynamic. Exhibit 4 [Appendix B] is a generalized performance curve with user-controlled variables along one axis and service-level performance along the other. User-controlled variables are, in fact, the various levels of volume and mix for each of the various categories of work.

In this case we will discuss TSO transaction response as a function of the volume of user activity. The data for this graph may be obtained from CMF, RMF, or SMF. In the case of a DOS or in non-IBM environments, the system log that records the volume of activity and system resources consumed should provide the necessary data. Exhibit 5 [Appendix B] is an example of a System Workload Summary§§ from which the following data can be obtained:

1. Volume of transactions by performance groups
2. Service units used in each system area

Exhibit 6 [Appendix B] is the CPU Utilization Report, which shows:

1. CPU busy data
2. CPU queue data

Exhibit 7 is a TSO Subsystem Performance Report, which shows:

1. TSO response by time period
2. TSO response by command
3. Concurrent TSO users

The above data was fed into the SAS statistical program to determine which data showed a relationship between TSO response and another variable. The variable with the closest relationship turned out to be the CPU queue time,*** which is directly affected by user volume and mix. In the case of our system, we are currently CPU-bound, so this is not a surprising bottleneck. As you can see in Exhibit 8 [Appendix B], a linear regression line has been drawn with the standard error shown as a dotted line on either side. In this case the standard error is 1.1 seconds on either side of the regression line. Basically this says that two-thirds of the time, observances of

§§Exhibits 5, 6, and 7 were produced by CMF for an IBM MVS system.

***As calculated by Little's Rule, $L = \lambda Q$, or the mean length of a queue is equal to throughput times queueing time.

TSO response versus CPU queue time will fall within the area bounded by the dashed lines.

The same analysis was done for TSO mix in Exhibit 9 [Appendix B]. This line turned out to be flat and was essentially meaningless because of the variation in data. Again, this corresponded to what we wish the system to do. TSO has a high priority; so even at high CPU busy levels, the TSO service should not suffer.

Exhibits 10 and 11 [Appendix B] show the effect on TSO response caused by total service units consumed and concurrent users respectively. In both cases there is a direct relationship, even though the standard error is larger. As you can see, using historical data, system interrelationships with user behavior can be derived. However, this kind of data tends to be linear, and does not answer the question of how the system will react to user behavior at levels not yet experienced.

A third problem is that the system is never exactly the same from one period to the next. For this reason, it is extremely important to correlate performance data from the system to data logged from a change management tracking system. A change management tracking system is basically a problem reporting system for all hardware, software, and applications problems and changes made to the system. There have been many times that a one-byte code change on Sunday night caused a system to come to its knees on Monday morning with resultant days of analysis before the change was found. Each system change needs to be documented and available for analysis when performance data shows a major deviation. This analysis goes a long way towards explaining the efficiency variance we discussed earlier and should be mapped to each change in the performance of the system.

Using a Model to Plot a Curve

Because graphing historical data will not always answer the questions of future system behavior for future user workloads, a model is often used to simplify the process. By using a model, curves may be defined for each system limitation as well as an overall curve for the present system capability. This requires many iterations of a model to define the points on the curve and even more iterations to define other possible curves. It is also important to validate the model to actual system results. That means, if we pick a point on the performance curve and take the corresponding volume and response, how close does this match reality? In other words, will the point fall within the bounds defined in the linear projection and standard errors graphed in the analysis of historical data, as was shown in Exhibit 10 [Appendix B]?

If indeed the model can be validated and the results are consistent, we have in fact defined an "operational standard." "Operational standard" as used here means that there is a standard performance characteristic for a given level of user activity in the current system. This becomes an extremely powerful tool for negotiating service levels with users. A major misunderstanding with users can be avoided if they realize that for some levels of user activity, response will be lower. This is especially true if there are peak periods with extreme activity for short periods during the day and that activity causes the "knee" of the curve to be reached.

Exhibit 12 [Appendix B] comes from an actual case where a factory made a union agreement to clock out all employees in 10 minutes. This agreement brought the system to its knees. The exhibit shows a performance curve for a 370–148 which averages 2.0-second response time for about 10,000 transactions during an 8-hour period. If, however, 625 transactions come in between 4:00 and 4:10 and must be processed in the same 2-second response time, a different system will be required. This is due to the fact that 625 transactions in a 10-minute period are equal to 30,000 transactions in an 8-hour period. A curve is drawn for a 3033 to illustrate the system expansion needed to accommodate the 10-minute traffic at 2-second response. In this case, the company management will have to weigh the service requirement against the additional investment. If user behavior is relatively inelastic, there will soon be a 3033 installed.

Exhibit 13 [Appendix B] is a performance curve of the Boole & Babbage data center that was made as an example for a case study. This system is composed of:

CPU	M80 (370–148) plus 6MB memory
Disc	6x 3330
	8x STC 3630 (3350)
Tape	3x STC 4534
Channels	4 plus byte multiplexor
Operating System	MVS/JES2/TSO/SPF/VAM/CICS

Response time shown by the model degraded significantly after 20 concurrent users and 1.1 transactions per second. In further analysis of this case, we found that we are indeed CPU-bound. This means that as the CPU resource is consumed by a workload or variations in user behavior patterns, degradation of service will be a direct result. A 4341 Model 2 with expanded CPU capacity is on order, and we hope to see some relief this fall when it is installed. The next step in this process will be to model the performance curve of the new system. We may then find some other system bottleneck such as I/O or DASD. In the meantime, we now have our operational standard of performance for TSO.

The Future of Performance Curves

The modeling of a workload against a given system is not only necessary, but is feasible with currently available tools and technology. The computation of and comparison to planned performance curves has value in determining:

1. Data center efficiency
2. The effects of user behavior (volume and mix)
3. Benefits of tuning
4. The ability of the data center to meet various levels of activity
5. The benefits of various system alternatives

Because of these benefits and the fact that future models will get even more involved in simulating operating system parameters (i.e., SRM under MVS), performance curves should be available on a realtime basis. This means linking the Change Management Tracking System and realtime system monitor/

model so that early warning mechanisms can be implemented. Realtime monitors will in effect simulate system changes and signal when the performance curve has changed from plan. This will provide an effective tuning tool by displaying these changes, much like the IPS parameters, in which domains and multiprogramming levels are displayed, in the IBM *Initialization and Tuning Guide.*

By monitoring the effects of user behavior, both on a continuous after-the-fact basis as well as in models, the data processing operation has implemented an important element of scientific management. The work analysis is done by the system itself while the manager deals with the question of user behavior and the integration of DP into the business.

APPENDIX A—Measures of productivity in DP operations

Results of Voting‡‡‡

Measurement factor	Ranking	Votes
Online response time	1	198
On-time reports	2	160
System uptime	3	141
User satisfaction	4	137
Rerun performance	5	101
Reports distributed w/o mistakes	6	56
Number of interrupts in online service	7	55
Problem resolution time	8	47
CPU utilization	9	45
Cost of operation	10	43
Actual vs. scheduled run time	11	39
Demand batch turnaround	12	34
Recovery time from failure	—	34
Time sharing interactive response	—	30
Late jobs fault of operator	—	30
Outages by category	—	29
Application program abends	—	27
Number of projects within budget	—	24
Hardware/software reliability, INTFAC	—	23
Number and time of tape mounts	—	20
Jobs completed per computer hour	—	19
People turnover	—	16
Absentee rate	—	15

‡‡‡Based on a survey of a meeting of data center managers at the 1981 EDP Performance Conference in Phoenix, Arizona, February 23–26, 1981.

APPENDIX B—Exhibits

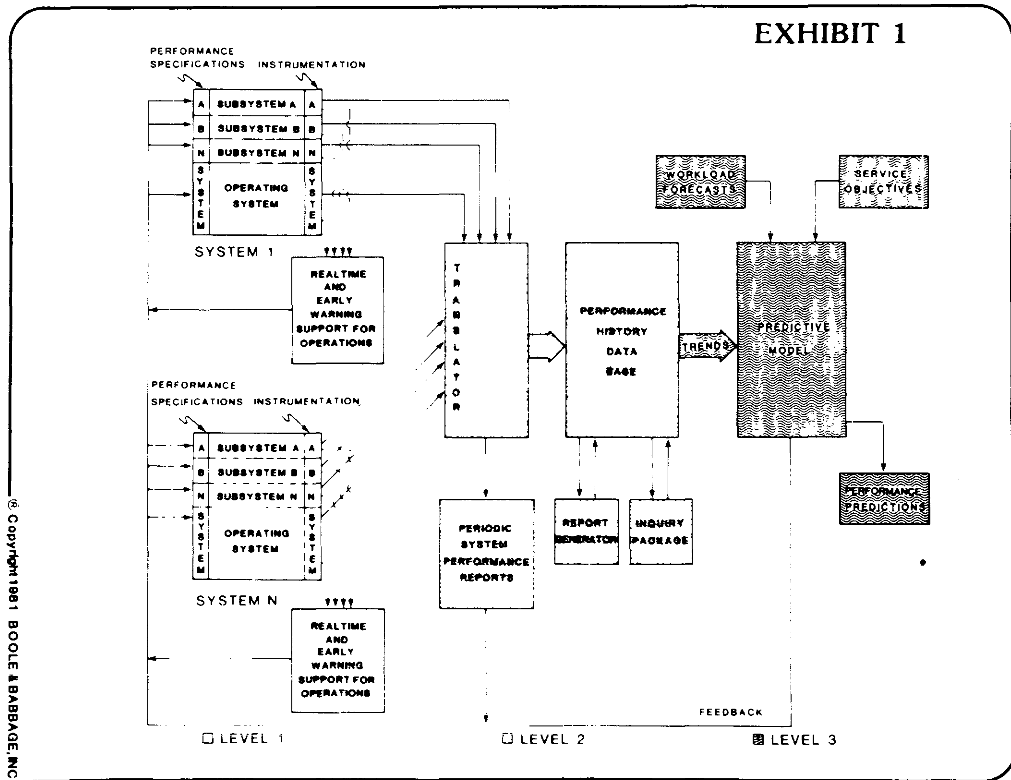


Exhibit 1—Integrated S/W approach with levels 1, 2, 3

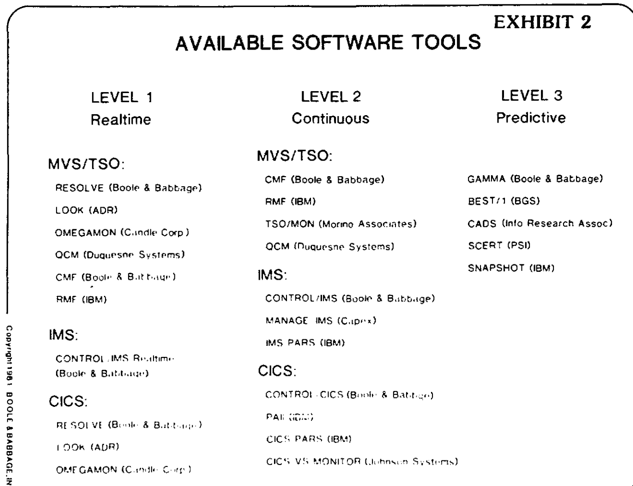


Exhibit 2—Available software tools

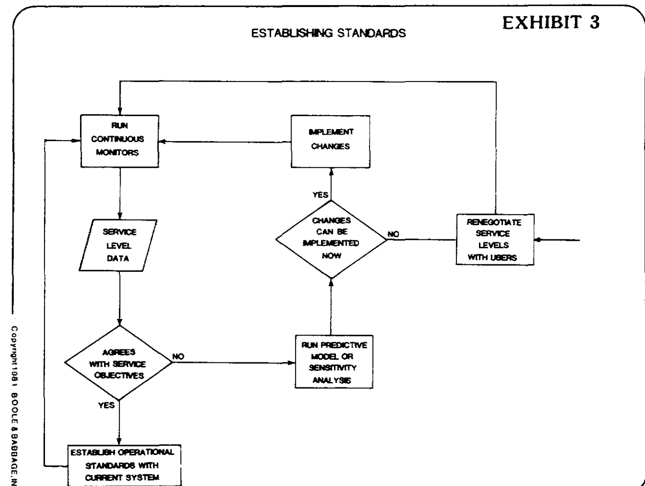


Exhibit 3—Establishing operational standards

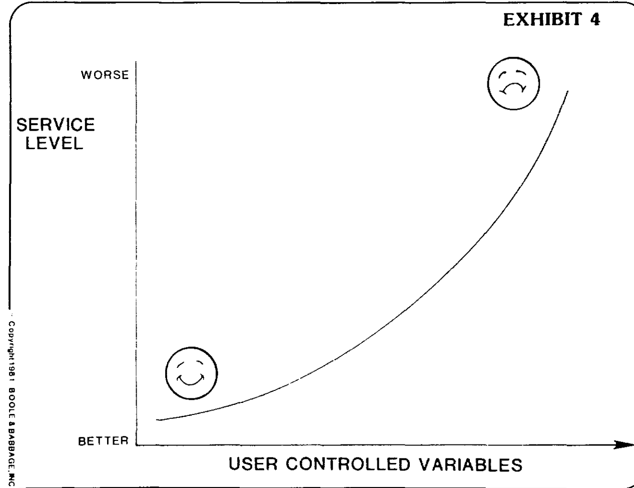


Exhibit 4—Generalized performance curve

EXHIBIT 5

SYSTEM WORKLOAD ACTIVITY REPORT

SERVICE USAGE BREAKDOWN BY PERFORMANCE GROUP

PERFORMANCE GROUP 2

PERFORMANCE GRP/PER/TS	TCB SERVICE SU/1000	PCT	SRB SERVICE SU/1000	PCT	I/O SERVICE SU/1000	PCT	MEM SERVICE SU/1000	PCT	TOTAL SERVICE SU/1000	PCT	SERV RATE	ABS RATE	SWAP COUNT	ENDED TRANS	AVG TRX TIME HH.PM.SS.TTT
2/1 /**	159.0	[illegible]	N/A	N/A	7.7	14.3	49.5	23.4	214.2	25.4	19.0	14.6	227	972	0.00.04.125
2/2 /**	19.6	3.3	N/A	N/A	[illegible]	1.5	5.1	2.1	24.4	2.9	19.3	15.4	57	203	0.00.07.[illegible]
2/3 /**	15.2	2.7	N/A	N/A	.9	2.3	3.4	1.6	19.6	2.3	26.3	21.0	17	49	0.00.21.336
2/4 /**	11.9	2.1	N/A	N/A	1.7	4.2	2.0	.9	15.7	1.9	10.5	9.4	5	11	0.00.05.102
SUMMARY	203.9	36.0	N/A	N/A	10.9	26.8	59.1	24.9	273.9	32.4	17.9	14.2	304	1235	0.00.06.120

© Copyright 1981 BOOLE & BABBAGE, INC.

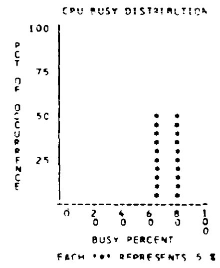
Exhibit 5—CMF system workload summary

EXHIBIT 6

PRODUCED BY CMF(1.4.0) BOOLE AND BABBAGE, INC.

CPU UTILIZATION REPORT

SUMMARY SECTION



® Copyright 1981 BOOLE & BABBAGE, INC.

Exhibit 6—CMF CPU utilization report

EXHIBIT 7

PRODUCED BY CMF(1.4.0) BOOLE AND BABBAGE, INC.

TSO COMMAND SUMMARY REPORT

COMMAND SUMMARY

INTERVAL SUMMARY

® Copyright 1981 BOOLE & BABBAGE, INC.

Exhibit 7—CMF TSO subsystem performance report

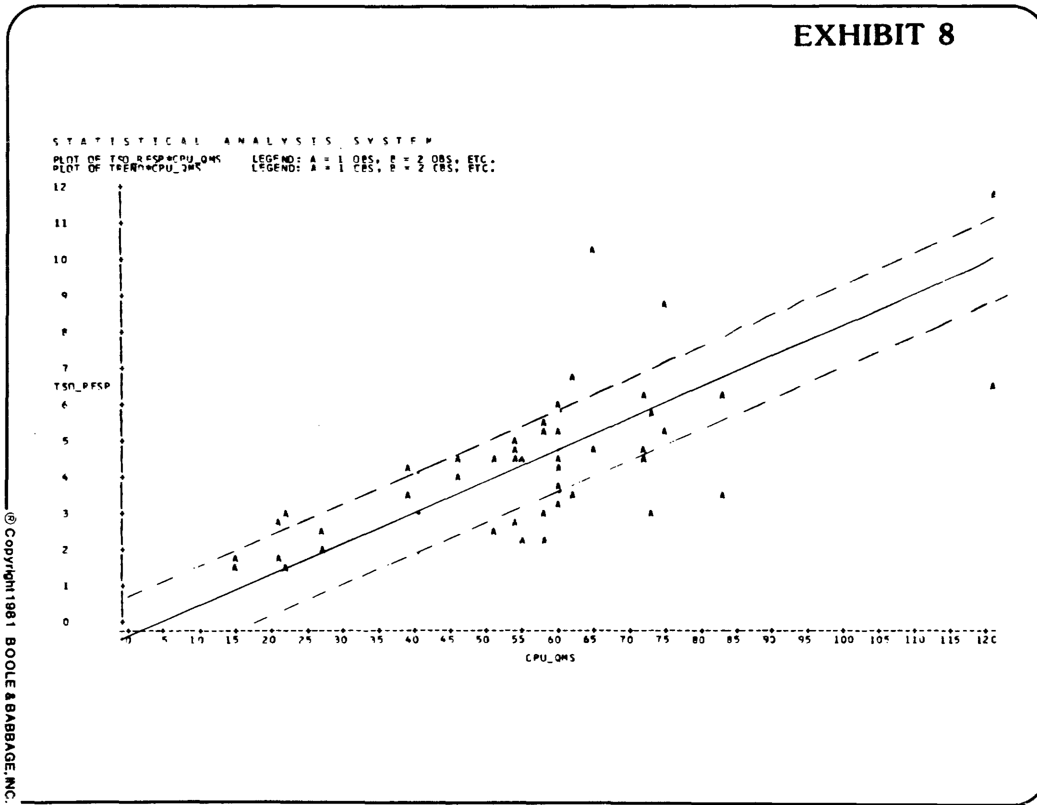


Exhibit 8—Linear regression—TSO response/CPU Q

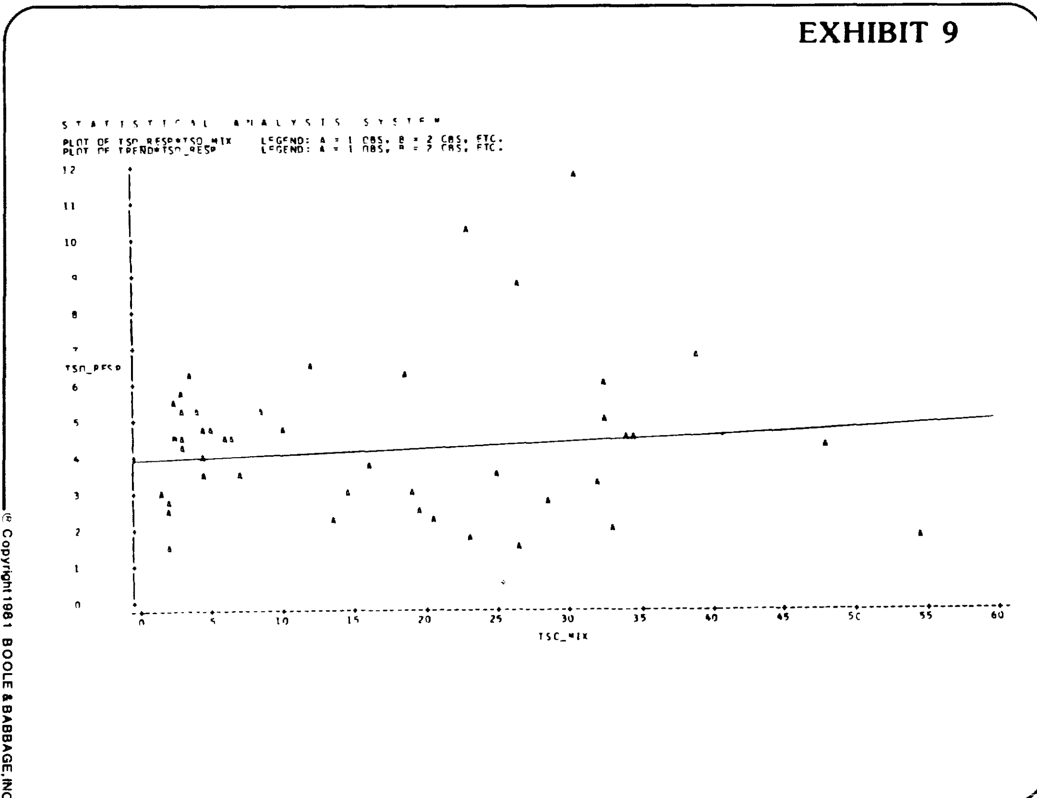


Exhibit 9—Linear regression—TSO response/TSO mix

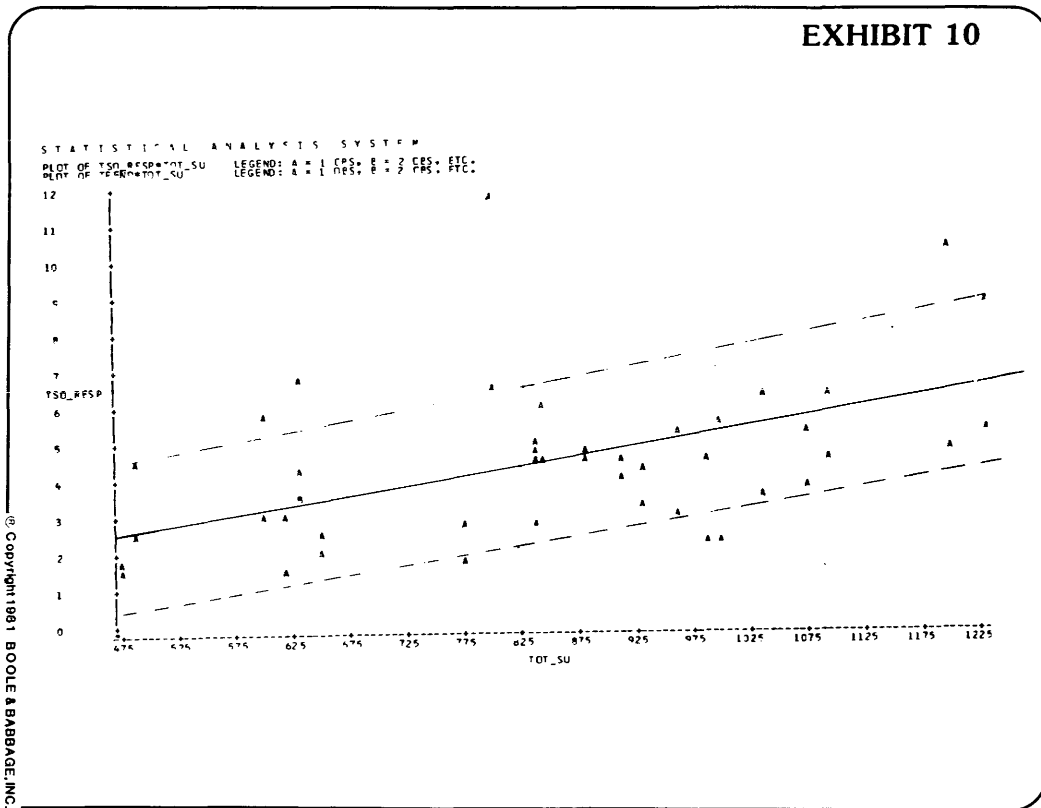


Exhibit 10—Linear regression—TSO response/SU's

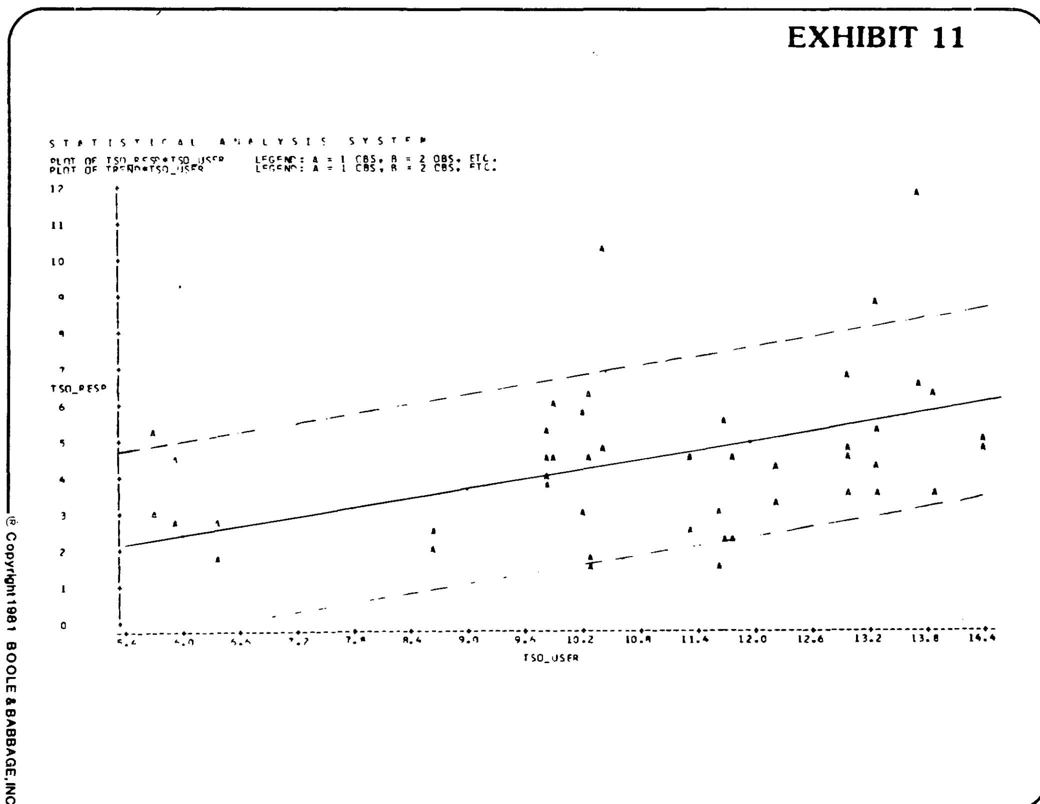


Exhibit 11—Linear regression—TSO response/concurrent users

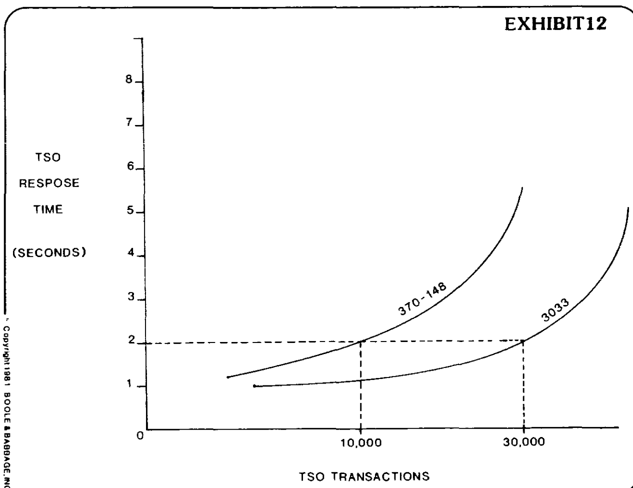


Exhibit 12—Performance curve for 370-148

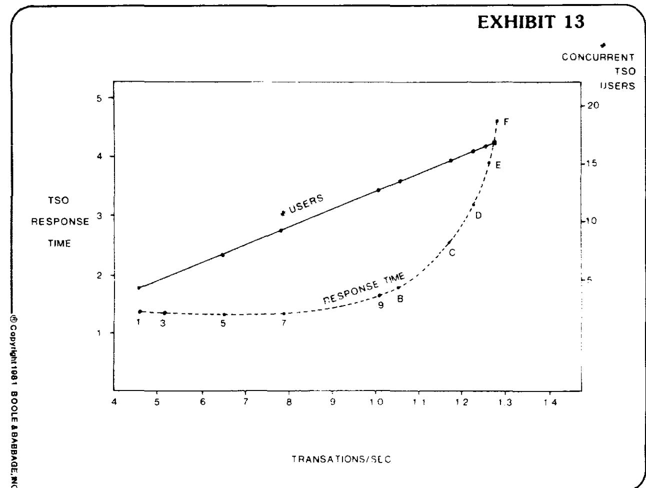


Exhibit 13—Performance curve for B/B DC

APPENDIX C—A bibliography for further study

1. Arnold O. Allen, *Probability, Statistics and Queueing Theory with Computer Science Applications,* Academic Press, New York, N.Y. (1978).
2. C. Warren Axelrod, *Computer Effectiveness: Bridging the Management/Technology Gap,* Information Resources Press, Washington, D.C. (1979).
3. L. Bronner, *Capacity Planning: An Introduction,* IBM Technical Bulletin GG22–9001–00 (January 1977).
4. L. Bronner, *Capacity Planning: Implementation,* IBM Technical Bulletin GG22–9015–00 (January 1979).
5. J. P. Buzen, "Queueing Network Models of Multiprogramming," Ph.D. Thesis, Harvard University, Cambridge, Mass. (1971).
6. *Computer,* April 1980, IEEE Computer Society (contains several articles of interest).
7. Peter F. Drucker, *Management: Tasks-Responsibilities-Practices,* Harper & Row, New York, N.Y. (1974).
8. Jeffrey L. Forman, *Change Communication: A Management System,* IBM Technical Bulletin GG22–9154–00 (July 1979).
9. *An Architecture for Managing the Information Systems Business, Volume I: Management Overview,* IBM, GE20–0662–0 (January 1980).
10. *Problem and Change Management in Data Processing—A Survey and Guide,* IBM GE19–5201–0 (August 1976).
11. *IBM Systems Journal,* Volume Nineteen, November 1, 1980, "Installation Management, Capacity Planning."
12. H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology,* Addison-Wesley, Reading, Mass. (1978).
13. J. D. C. Little, "A Proof of the Queueing Formula, $L = \lambda W$," *Operations Research 9,* pgs. 383–387 (1961).
14. J. Martin, *Design of Real-Time Computer Systems,* Prentice-Hall, Englewood Cliffs, N.J. (1972).
15. J. Martin, *Systems Analysis for Data Transmission,* Prentice-Hall, Englewood Cliffs, N.J. (1972).
16. Montgomery Phister, Jr., *Data Processing Technology and Economics,* Santa Monica Publishing Company, Santa Monica, Calif. (1977).
17. Charles H. Sauer/K. Mani Chandy, *Computer Systems Performance Modeling,* Prentice-Hall, Englewood Cliffs, N.J. (1981).
18. David R. Vincent, "Software Tools for Service Level Management," *Data Management,* pgs. 25–29, March 1981.
19. David R. Vincent, "Measuring Performance Online," *ICP Interface, Data Processing Management,* Summer 1980.
20. David R. Vincent, "Service Level Management," 1980 *CMGXI Proceedings,* pgs. 196–207.
21. David A. Wren, *The Evolution of Management Thought,* John Wiley & Sons (1979).

Distributed processing with the Z8000 family

by RICHARD MATEOSIAN and JANAK PATHAK
Zilog
Campbell, California

ABSTRACT

The Z8000 Family plan philosophy envisions a distributed processing approach to many Z8000 applications. The Z8000 Family consists of CPUs, CPU support circuits, and a full complement of VLSI peripherals. These components are all integrated by the Z-BUS, which defines the interconnections and transactions among them. The basic philosophy of the family plan is that of distribution of intelligence and function among complementary VLSI components. Of the several possible realizations of this philosophy, the one chosen has the following major aspects:

1. Synchronization primitives in bus and component architectures
2. Extensively programmable VLSI peripherals and CPU support circuits
3. Bus support for cooperative transactions
4. Built-in support for interprocess message passing

SYNCHRONIZATION PRIMITIVES

The Z-BUS has two features specifically designed for inter-component synchronization in a distributed processing environment:

1. The "bus lock" status code
2. The resource request lines

Each of these bus features is designed to work with specific CPU instructions.

The "Bus Lock" Status Code

The "bus lock" status code is one of the 16 possible codes representable on the status lines ST_3-ST_0 of the Z-BUS. This status occurs during the fetch cycle of the Test and Set (TSET) instruction, which is available on all Z8000 CPUs. The TSET instruction is used to implement semaphores. Its job is to test a specified memory location for a predefined "available" code and to set the contents of the memory location to "not available." The inclusion of these two actions in a single instruction prevents any access to the specified location between the testing and the setting. That is, it prevents access by any other process running on the same CPU, which might happen if an interrupt occurred between separate testing and setting instructions. When other devices, such as another CPU or a DMA controller, have access to the same memory as the CPU executing the TSET instruction, the testing and setting operations must be inseparable at the bus transaction level. This inseparability is implemented through use of the "bus lock" status code.

The Resource Request Lines

In some distributed systems, several CPUs that do not share a common memory may need to share a common resource. In this case, the TSET instruction cannot be used. For such situations, the resource request lines of the Z-BUS have been provided. Figure 1 shows a prosaic example of their use: three CPUs sharing a line printer. When a CPU needs to use the line printer, it executes the MREQ instruction, which conducts a transaction on the four resource request lines; condition code settings indicate to the program whether or not the CPU gained control of the line printer through this transaction. If not, the MREQ instruction is executed again; if so, the line printer is used, then released through execution of the MRES instruction. If another CPU executes an MREQ instruction while the line printer is being held, the resource request transaction results in a "not available" indication.

PROGRAMMABLE VLSI COMPONENTS

The use of extensively programmable VLSI peripherals and CPU support circuits brings aspects of distributed processing into most Z8000 applications, even those with only a single CPU. The principal programmable VLSI components of the Z8000 Family are summarized below.

Memory Management Unit (MMU)

The MMU provides address translation and access protection, using internal tables transmitted from the CPU. Because of the Z8000's segmented addressing, which allows segment

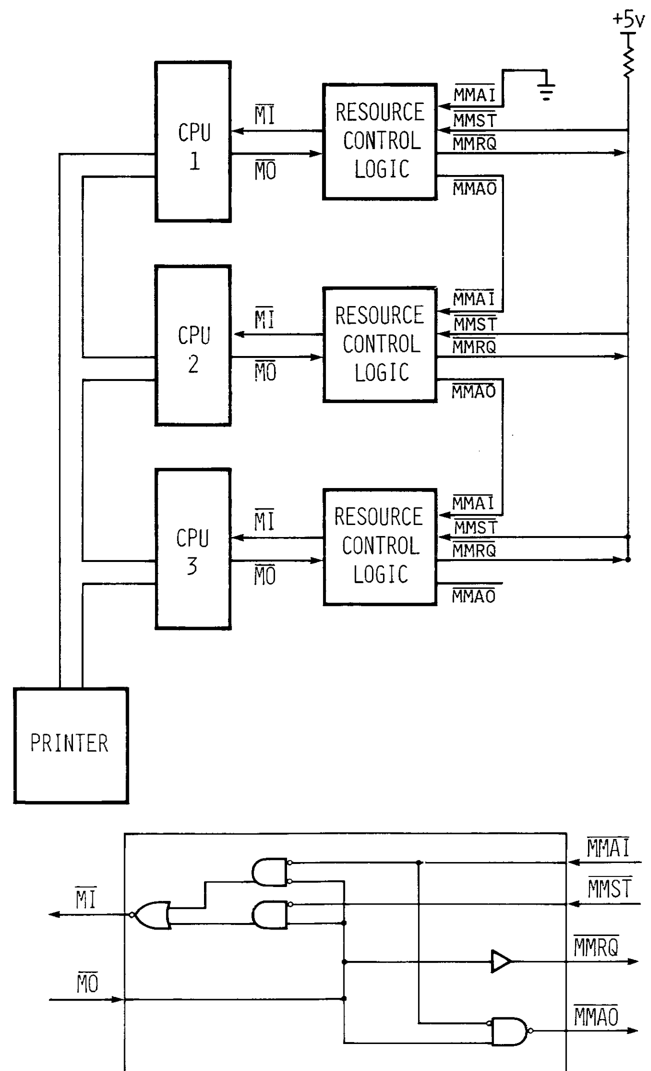


Figure 1—Resource lines provide non-memory-based synchronization

identity to be output by the CPU before completion of the indexing portion of address computations, the segment-related address processing done by the MMU occurs in parallel with the CPU's indexing. This parallel processing approach minimizes the overhead of external address translation and access protection.

DMA Transfer Controller (DTC)

The DTC can carry out high-speed block data transfers and searches independently of the CPU's operation. Control of the DTC by linked lists of command blocks in memory allows the DTC and CPU to carry out joint functions asynchronously. When an MMU is in the configuration, the DTC can work with logical or physical addresses. A special control line and a bit in the MMU access control registers allow the MMU to protect certain blocks of memory from DMA transfers and to prevent CPU access to blocks of memory while they are being changed by a DMA transfer.

FIFO Input/Output Interface Unit (FIO)

The FIO allows asynchronous parallel data transfers between processors, making it a key element in distributed multi-processor systems (see Figure 2).

The FIO is simply a 128-byte, first-in-first-out buffer, expandable in width and depth, equipped with bidirectional parallel interfaces at each "end" of the buffer and a set of message registers for interprocessor communications that bypass the buffer. The FIO is designed to cooperate with the DTC in "flyby" transfers (described below) to initiate DMA transfers without CPU involvement and to terminate DMA transfers on the basis of patterns recognized in the transferred data.

The Counter/Timer and Parallel I/O Unit (CIO)

The CIO has many functions related to real-time I/O processing. It is not a separate I/O processing CPU for use with the Z8000, but it does perform many of the same functions: bidirectional parallel I/O with a variety of handshake modes, counting and timing of external signals, and priority interrupt control.

The Serial Communications Controller (SCC)

The SCC, like the CIO, carries out many of the functions of a dedicated CPU working with the Z8000. It performs all of the tasks associated with serial communications on two independent 1Mbit/second channels, using any of a variety of protocols.

COOPERATIVE TRANSACTIONS

An essential element of the Z8000's distributed processing Family plan is the use of cooperative transactions. The principal examples are:

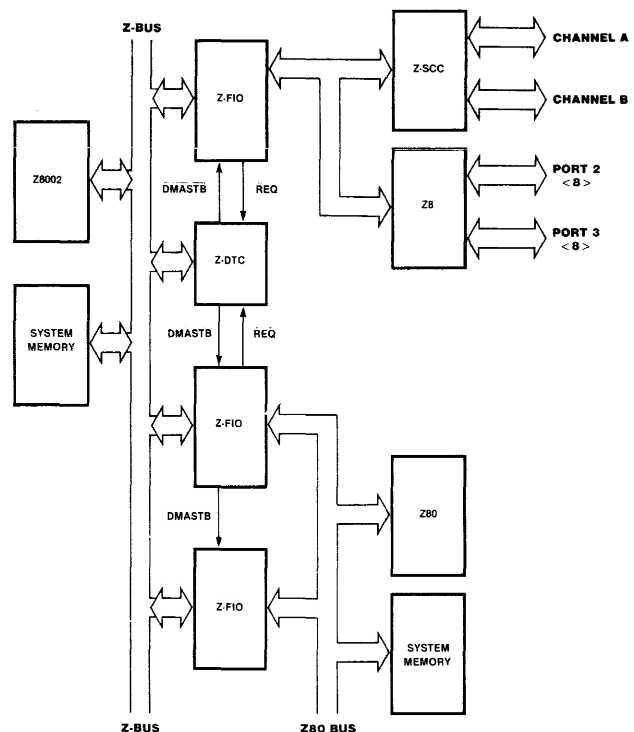


Figure 2—FIO links processors and cooperates in DMA transfers

1. CPU/MMU generation of physical addresses
2. Extended processing architecture
3. DTC/FIO "flyby" transfers

The common theme behind cooperative transfers is that each device has specific capabilities and that when a task requires a combination of capabilities, it is better to allow several devices to participate in the task than to replicate capabilities in several devices. Thus, for example, rather than equipping the FIO with DMA transfer capabilities, it was deemed more sensible to provide for joint DTC/FIO transfers.

Of the three examples of cooperative transfers listed above, CPU and MMU cooperation has already been discussed. The other two examples will now be described.

Extended Processing Architecture

An important goal of the Z8000 Family design was to accommodate additional processing capabilities (such as what would be provided by a floating point chip) with no redesign of the overall system or software. This goal was achieved with a scheme that allows certain CPU instructions either to cause traps (allowing simulation of an absent chip's function) or to be executed cooperatively by the CPU and an extended processing unit (EPU). With this cooperative approach, the CPU's addressing capabilities are used to fetch or store the arguments, and the EPU performs the operations. EPU operation can proceed in parallel with the execution of subsequent instructions by the CPU; synchronization is achieved by the EPU's assertion of the CPU's STOP line if the CPU fetches

another EPU instruction before the EPU is ready to execute it. Figure 3 illustrates the cooperation of the EPU and the CPU.

The Extended Processor Architecture gives designers a great deal of flexibility. For example, an EPU doing floating point operations could be used interchangeably with floating point software controlled by the same instruction stream; only a single bit in the CPU's Flag/Control Word (FCW) control register would need to change. Thus, a high-performance floating point chip could be an optional feature of a product that used floating point operations. The "slow" version would use software execution of the floating point instructions, and the "fast" version would use the chip to execute instructions. Both versions would have identical applications program code and circuitry.

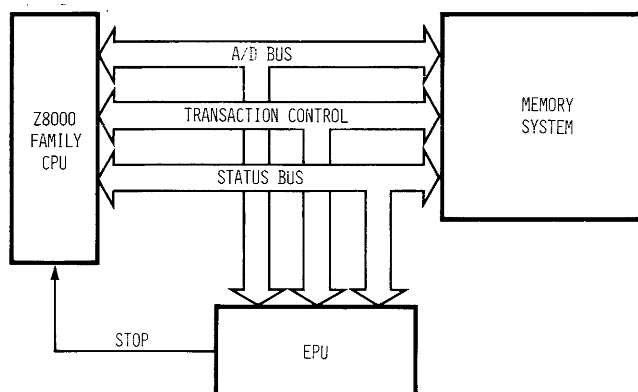


Figure 3—CPU and EPU cooperate to execute instructions

The EPU monitors the status lines, looking for "Instruction Fetch, First Word" status. When this occurs, it examines the instruction presented on the A/D bus. If the instruction is for that EPU, it either asserts STOP (if it is still busy executing a previous instruction) or initiates execution of the indicated instruction. The EPU instruction can be entirely internal to the EPU, or it can include one or more transfers of data between the EPU and CPU or EPU and memory. For each of these cases, the CPU generates the appropriate status signal (ST_3-ST_0) and transaction control (R/W, B/W, AS, MREQ, DS) lines, and the EPU takes or supplies data as appropriate.

Flyby Transfers

A "flyby" transfer is a DMA transfer in which the data never enters the DMA controller circuit. The DMA controller provides all necessary memory addressing, transfer counts, and bus control signals, but at the point in the transaction when data must pass from one component to another, an intelligent peripheral (like the FIO) supplies or takes the data. Flyby transfers are, therefore, approximately twice as fast as ordinary DMA transfers, in which one transaction is required to fetch the data from the source and to latch it in the DMA controller, and a second transaction is required to pass the data from the DMA controller to the destination.

SUPPORT FOR MESSAGE PASSING

The support for message passing in the Z8000 Family plan is predicated on the assumption that interprocess communication in Z8000 systems can be conducted effectively through messages. Other means of interprocess communication are not precluded, but message passing is the only interprocess communication method supported by special architectural features.

Since message passing is generally implemented through the movement of blocks of characters from one location to another, one of the principal means of supporting message passing in the Z8000 Family plan is the multi-level support of block data movement. The block I/O and memory transfer instructions of the CPU, the capabilities of the DTC, and the features of the FIO are all designed to complement each other in providing efficient, flexible block data movement throughout Z8000 systems.

Another instance of message passing occurs in the communication protocol defined between the Z8000 CPUs and the Universal Peripheral Controller (UPC). The UPC is a Z8-based single-chip microcomputer designed for use in device controllers. It functions as a slave processor to the CPU, and because it is directly tied to the operation of a physical device, it is essential that a faulty CPU program not cause the UPC to fail.

The fail-safe protocol for CPU/UPC communication calls for designation by the UPC of specific blocks of its internal memory for use as shared message buffers. The CPU has direct access to the designated buffer area but cannot access any other portion of the UPC's memory until the UPC designates that portion as the message buffer. The CPU always sees a single address in its I/O address space as "UPC message buffer," but the UPC maps this address internally into the desired area of its memory.

SUMMARY

Distributed processing with the Z8000 Family is not a special case. The distribution of function among CPU and extensively programmable VLSI components demands that the basic mechanisms of communication and synchronization be included in the design of the Z-BUS and all the Z8000 Family components. In addition, specific attention has been given to multi-CPU system problems through use of specific CPU instructions and bus protocols and through use of the First-In-First-Out Interface Unit (FIO) as a flexible buffer between asynchronously functioning systems. Cooperative transactions, in which the functions of several components must combine to carry out the desired action, bring distribution of function to the bus and component level. Finally, architectural features supporting message passing facilitate distributed processing at the software and application structuring level.

Distributed processing with iAPX 186 microprocessor systems

by TONY ZINGALE
Intel Corporation
Santa Clara, California

ABSTRACT

In most early computer systems, large central computers, minicomputers, or microcomputers were used to perform all the necessary data processing activities in the system. The overall performance of the entire system was limited by the ability of the central CPU to bring in data, process it, and output it in some usable format. This often resulted in large data input/output bottlenecks in I/O-intensive applications where the CPU time required to service I/O functions left little time for data processing. The obvious result is a slow, nonoptimum data processing system. Now many applications are moving in the direction of simpler and easier-to-use distributed systems, where a central CPU delegates some of the processing tasks to distributed processing subsystems. Not only are costs lower with the distributed system approach, but the time needed to implement such systems is substantially less.

For example, a network transaction processing system can now use numerous automatic tellers to process data at a variety of dispersed geographic locations. The tellers, or distributed nodes, can then collect, process, output, and eventually pass on necessary information to the central host computer, located at some detached location, without burdening the central computer with handling each simple transaction. The host becomes involved with an individual node only when the intelligent node requires the timely interaction. The heart of a distributed node itself must be an intelligent processing device capable of handling all the processing and I/O requirements needed by the node.

The iAPX 186 is a new highly integrated 16-bit microprocessor. It combines 10 of the most common microprocessor system components onto one. The 80186 is essentially a 16-bit CPU board integrated onto a single silicon chip. By combining a limited number of peripheral support components with memory together with an iAPX 186, one can achieve a condensed, cost-effective system on one board, making the 80186 an optimal microprocessor for distributed processing nodes. This high level of integration is accomplished through an advanced HMOS II silicon gate technology. For the first time it provides a system cost saving significantly greater than that of the previous 16-bit microprocessor design alternatives. The 80186, an upgrade from the industry standard iAPX 86 and 88, offers two to three times the system throughput of a standard iAPX 86. The iAPX 186 adds 10 new instruction types to optimize existing iAPX 86 or 88 application code or streamline new iAPX 186 application code. All these hardware and software attributes make distributed processing with iAPX 186 systems a cost-effective, easy 16-bit microprocessor solution.

CLASSICAL DISTRIBUTED PROCESSING

The concept allows the use of dispersed processing sites or nodes to offload a sophisticated central computer, minicomputer, or microcomputer. The real payoff from the distributed processing approach is the increased responsiveness to the user's needs of the data processing function, achieved by providing an effective, fast, powerful processing mechanism at the lower levels. The declining costs of microcomputers and memories have provided the economic justification for distributed computing. The approach now is to let microcomputers located near the data do much of the real-time processing and send only a summary to the host computer (Figure 1).

Distributed Processing Node Requirements

The data processing node must be

1. More cost-effective than similar approaches
2. Easy to implement, thus making possible a fast end-product time to market
3. Compatible with existing software, if any
4. Capable of high-speed execution rates

In addition to the general requirements stated above, there are a set of hardware requirements to be satisfied.

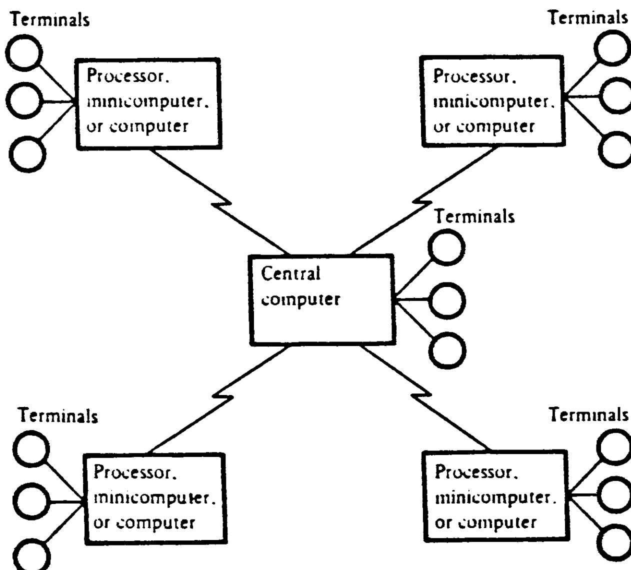


Figure 1—Distributed STAR network

1. High-speed, flexible DMA is needed by any I/O subsystem to accomplish data transfers between I/O devices connected to the distributed node (i.e., keyboards, disks, printers, modems) and local system memory or vice versa. This is a key requirement for moving blocks of data in and out of a distributed node that can improve system performance and execution time.
2. Flexible hardware timers are always required to time external events occurring in the system. Timed external events usually correspond to some sychronized system activity. For example, the number of words that have been printed may signify to the CPU that it needs to start a new page or generate some type of interrupt to the CPU to stop printing.
3. In a time-sensitive distributed system there is a definite need for the handling of a large number of real-time interrupts. For example, if several intelligent terminals are connected to a single distributed node in addition to the standard I/O devices, multiple interrupts will appear at the node simultaneously. These interrupts must be acknowledged, prioritized, and handled cleanly and rapidly.
4. Address decoding hardware is needed to provide the system with a systematic convention for selecting memory spaces and peripheral devices; wait-state generating circuitry is required to insure timing compatibility with memories and peripherals at the proper speeds. This hardware can require an appreciable portion of the board space of the distributed node.

This feature set is optimal in that it provides all the basic requirements of a distributed processing node. The iAPX 186 integrates these common system functions into a single silicon chip.

Cost-Effective, Optimal Integrated Feature Set

A block diagram of the iAPX 186 integrated hardware feature set is shown in Figure 2, followed by a summary of each on-chip feature.

Clock generator: The 80186 provides an internal clock oscillator, which requires a single external crystal or TTL-level frequency source. The system clock output is a standard 8-MHz, 50% duty cycle clock at half the crystal frequency, or 16 MHz. This output can be used to drive the clock inputs of other system components and hence make additional clock generation devices unnecessary. Synchronous and asynchronous ready inputs are supplied for flexible peripheral-device synchronization.

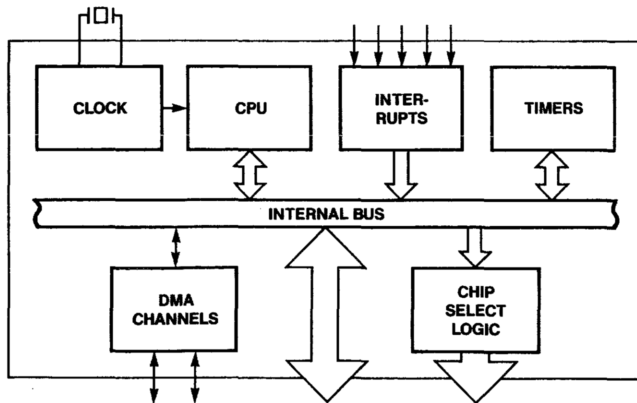


Figure 2—iAPX 186 CPU (80186) block diagram

Timers: Two independent 16-bit programmable timer/counters are provided to count time external events, external events, and generate nonrepetitive waveforms. A third 16-bit programmable timer, not connected externally, is useful for implementing time delays and as a prescaler for the two externally connected timers. The iAPX 186 integrated timers are very flexible and can be configured to time/count a variety of distributed I/O types of activities.

Each of the three timers is equipped with a 16-bit timer register that contains the current value of the timer. It can be read or written at any time, independent of whether the timer is running. Each timer is also equipped with a 16-bit max count register containing the maximum value the timer will reach. In addition, the two externally connected timers each have a second 16-bit max count register, which enables the timers to alternate their count between two different max count values as programmed by the user. When a terminal count is reached, an interrupt may be generated, and the timer value is reset to zero.

The timers have several flexible programmable options in their mode of operation. All three timers can be set to halt or continue on a terminal count value, so no external event or device need wait for a timer reset. The two externally connected timers can select between internal and external clocks, alternate between max count registers or use only one, and be set to retrigger on external events.

DMA channels: The on-chip DMA controller unit in the iAPX 186 contains two independent high-speed DMA channels. DMA transfers can occur between memory and I/O spaces (i.e. M–I/O) or within the same space (i.e. M–M, I/O–I/O). The latter feature allows I/O devices and memory buffers to be freely located anywhere in the distributed system. For example, memory-mapped I/O can be handled without any external decode logic to select the required I/O space or device. Each DMA channel maintains two 20-bit source and destination pointers that can be incremented, decremented, or left unchanged after each transfer. Data transfers are programmed by the user to be either byte or word transfers and can occur anywhere in the 1 megabyte of directly addressable memory space. This allows a maximum transfer rate of 1 MWord/second or 2 MBytes/second. The user can specify several different modes of DMA operation via the on-chip 16-bit DMA channel control word.

By using the 80186 DMA facilities, data can be input onto local system memory, processed, passed on to the host computer (if needed), and output to another I/O device, all by the use of the two independent, high-speed, on-chip DMA channels.

Interrupt Controller: The 80186 interrupt controller resolves priority among interrupt requests that arrive simultaneously. It can accept interrupts from up to five external hardware sources (NMI + 4) and internal sources as well (timers, DMA channels). Each interrupt source has a programmable priority level and a preassigned interrupt vector type, used in deriving an address to a table in memory where interrupt service routine addresses are located. This enhancement of predefined vector types makes the interrupt response time about 1.5 times faster than the typical iAPX 86 response time. The 8259A programmable interrupt controller (PIC) interrupt modes, like fully nested and specially fully nested, are provided by the 80186 as well. In addition, multiple 8259As can be cascaded to provide the system with up to 128 external interrupts. There is also an RMX-86 real-time operating system mode of operation for maximum user flexibility that provides many of the same interrupt features described here.

Chip select/ready generation: The iAPX 186 contains programmable chip select logic to provide chip select signals for memory components, peripheral components, and programmable ready (wait states) generation logic. The result of this integrated logic is a lower system part count, since as many as 11 TTL packs will be saved. In addition to a lower system cost, the speed/timing performance of the system will improve as a result of the elimination of external propagation delays. Another advantage involves flexibility in the choice of memory component size and speed. Three memory ranges (lower, middle, upper) can be programmed to variable lengths (1K, 2K, 4K, . . . , 256K) so that a variety of memory chip sizes can be used. Further, anywhere from zero to three wait states can be programmed so either high-speed or low-cost, slower memories can be used. With respect to the peripheral chip selects, as many as seven different peripheral components can be addressed via I/O or memory space. Again, programmable wait states may be injected to synchronize slower peripherals with the 80186 itself or memory.

The chip select/ready logic contributes heavily to making the iAPX 186 an optimum, low-cost choice for a distributed processing node. In the past, this necessary logic had to be designed, debugged, and programmed. Now, with the 80186, the design, debug, and programming are done by initializing the associated 16-bit on-chip control registers.

CPU internal registers: The added functionality of the iAPX 186 (i.e., timers, DMA, interrupt controller, and chip selects) uses on-chip 16-bit control registers for each integrated device. They are contained in a 256-byte control block (see Figure 3) included in the 80186 CPU register architecture. The control register block may be either I/O or memory-mapped, based on initialization for a new control block pointer in the CPU. Except for these additions, the register architecture of the iAPX 186 is identical to the iAPX 86.

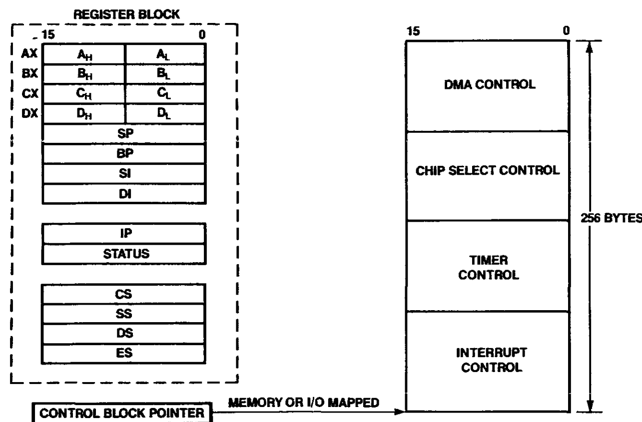


Figure 3—iAPX 186 register architecture

TABLE I—Relative execution comparisons: iAPX 186 (8MHz clock rate) vs. iAPX 186

Instruction	8086 (5MHz)	8086-2 (8MHz)	8086-1 (10MHz)
MOV REG TO MEM	2.0–2.9X	1.2–1.8X	1.0–1.4X
ADD MEM TO REG	2.0–2.9X	1.2–1.8X	1.0–1.4X
MUL REG 16	>5.4X	>3.4X	>2.7X
DIV REG 16	>6.1X	>3.8X	>3.0X
MULTIPLE (4-BITS) SHIFT/ROTATE MEMORY	3.1–3.7X	1.95–2.3X	1.6–1.8X
CONDITIONAL JUMP	1.9X	1.2X	1.0X
BLOCK MOVE (100 BYTES)	3.4X	2.1X	1.7X

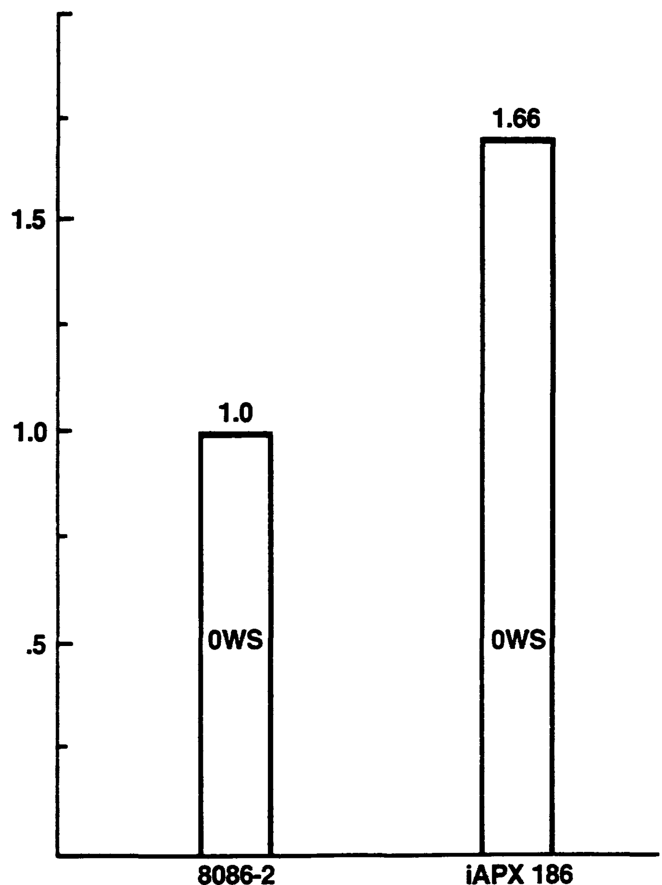
Software Compatibility

Since software costs are influencing most microcomputer decisions today, system designers must take this enormous investment seriously when choosing microcomputers for future product upgrades. This is especially true in the cost-sensitive distributed processing area, where virtually hundreds of nodes will be designed and programmed to interface to a central host computer. Software compatibility between the nodes and the central host makes the overall system easier to use and will shorten the design cycle considerably. For future product upgrades, software compatibility must be a decision variable in today's product. If not, when bringing a new product to market, engineers may spend all of their time rewriting hundreds of lines of general-purpose software rather than writing new streamlined application code. All this can be saved by using the iAPX 186. Since the 80186 is completely object-code-compatible with the iAPX 86 or 88 or 286, software investments are intact for future product offerings. Not only is the 80186 totally software-compatible with the 8086 or 8088; it adds 10 new instruction types as well. Instructions like block move (running at bus bandwidth or 2MBytes/sec), push or pop all the registers (push/pop all), and multiply immediate are all new to the basic iAPX 86, 88 instruction set. These instructions help enhance existing iAPX 86 or 88 application code, if needed, or produce optimum, high-speed iAPX 186 code.

iAPX 186 Performance Comparisons

The iAPX 186 overall performance speed is two to three times faster than the 5MHz iAPX 86 and 30% faster than the 8MHz iAPX 86. Many instructions, specifically those for integer arithmetic (i.e., multiply and divide), execute 5 to 6 times faster than on a 5MHz iAPX 86 (see Table I). In benchmarks based on Intel standard applications, operations like block translation, bubble sort, and automated parts inspection show that the iAPX 186 yields a 1.66 times performance increase over the 8MHz iAPX 86 (see Table II). These benchmarks were selected to evaluate the performance of 16-bit microprocessors and demonstrate the capabilities necessary for intensive I/O operations, general integer arithmetic, and data manipulation operations necessary for real-time business and EDP applications. Naturally the most likely environment for finding a distributed processing system lies in these application areas. The iAPX 186 satisfies the high-speed execution requirement for a distributed node by surpassing the existing high-performance standards set by the iAPX 86 and at the same time is totally software-compatible to the iAPX 86, 88, and 286.

TABLE II—Relative throughput benchmark, iAPX 186 vs. 8MHz iAPX (based on Intel standard application benchmarks)



TYPICAL DISTRIBUTED SYSTEM CONFIGURATION

A sophisticated central host computer capable of handling multiple users in a real-time environment is obviously a *major* need for an effective distributed processing system. This device is responsible for controlling all the distributed nodes in the system. This requires an extremely large memory space to handle the multiple-nodes memory and I/O space requirements and also requires some form of system integrity mechanism that would insure that each node executes independent of the others. The microcomputer that fits this requirement best is the iAPX 286 (see Figure 4). Not only is the 80286 software compatible with the iAPX 86, 88, 186, providing six times the performance of an 5 MHz iAPX 86; it also offers on-chip memory management and memory protection. The iAPX 286 is capable of directly managing up to 16 megabytes of real memory and up to 1 gigabyte (2^{30} bytes) of virtual memory. It can provide memory protection for each distributed node by verifying each specific task's address range and access rights for every memory access. These integrated features of the 80286 satisfy the requirements of a central host and of controlling the distributed nodes in a system, since each will require some independent memory space and also some form of protection from the other nodes in the system.

As Figure 4 shows, communications between the iAPX 286 host computer and the iAPX 186 distributed nodes takes place by passing messages and data through a dual-port RAM. The dual port is used to isolate the iAPX 186 systems or nodes from the protected bus structure of the iAPX 286, maintaining full system integrity.

One design variable to consider in a distributed node scheme is error detection and correction in and out of the dual-port RAM. The Intel 8206 Error Detection and Correction unit performs this function with one device. The 8206 serves as an interface between large memory systems (i.e., iAPX 286 systems) and the system bus of the iAPX 186. The EDC unit will internally detect all one-bit errors and most multiple-bit errors and automatically make corrections. Obviously, errors can occur in any system configuration when data are written incorrectly to memory, a memory cell loses data, or a complete memory component is missing or dead. These errors can be carried throughout the system and affect end results unless detected and corrected. In Figure 4 a dual-port RAM scheme is used to interface the iAPX 286 protected system bus to the iAPX 186 local bus. The 8207 Advanced Dynamic RAM Controller is capable of controlling two memory ports at the 8-MHz speed for both microcomputers and supporting a megabyte of address space. The 8207 provides the necessary control and timing signals to interface memory to the 8206 EDC component as well (see Figure 5). Previously this mechanism, the combined 8206 and 8207 components, took as many as 50 TTL components. Together the two peripheral devices provide a cost-effective, error-free, highly reliable memory subsystem for a distributed processing node.

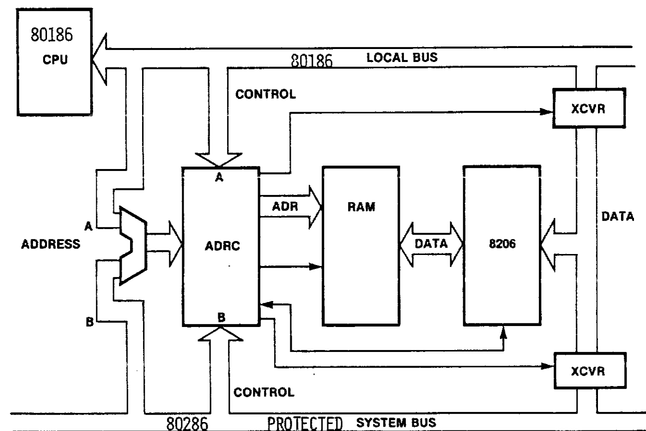


Figure 5—Dual-port RAM control with EDC

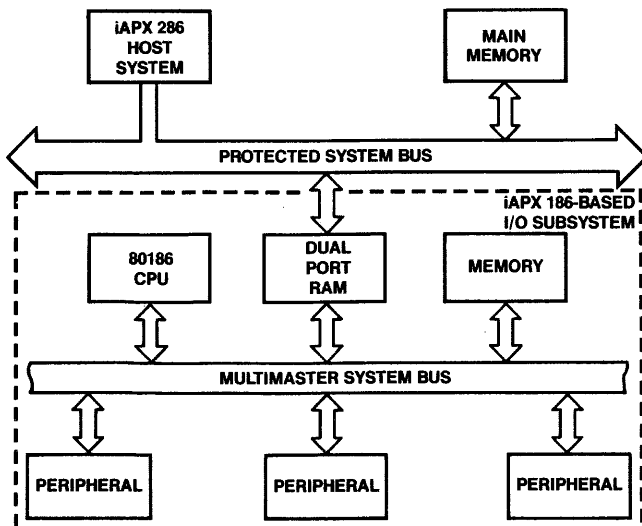


Figure 4—Distributed iAPX 286—iAPX 186 system

CONCLUSIONS

The iAPX 186 exceeds all the stated requirements for use as an effective distributed processing node. This optimal integrated feature set of the 80186 is streamlined to manage the necessary I/O hardware and real-time/high-speed software needs of a distributed system. It is very cost-effective, easy to use, high-performance, and compatible with any iAPX 86 or 88 existing software; and it can also be tightly coupled with an iAPX 286 central host and provide highly reliable memory subsystems through the use of the 8206 EDC and the 8207 peripheral devices.

REFERENCES

1. Thierauf, Robert J. "Distributed Processing Systems." Englewood Cliffs, New Jersey: Prentice-Hall, 1978.
2. Down, P. J., and F. E. Taylor. "Why Distributed Computing?" Rochelle Park, New Jersey: Hayden Book Company, 1977.

3. Moore, W. G. "Going Distributed." *Mini-Micro Systems*, 10 (1977), pp. 41, 44, 46, 48.
4. Klovstad, J., and S. Kopec. "iAPX 186 Target Specification Revision 2." Intel preliminary design document. October 16, 1980.
5. Klovstad, J., and S. Kopec. "iAPX 186 Architectural Overview Revised May 1981." Intel overview document, available from Intel Corporation, Santa Clara, California.
6. Heller, P. "The Intel iAPX 286 Microprocessor." *IEEE Wescon Trade Show Proceedings,* San Francisco, 1981.
7. Kop, H. "16-Bit Microprocessor Benchmark Report: iAPX 86/10, Z8000, MC68000." Intel Corporation, 1981.
8. Intel Corporation. *Perperal Design Handbook,* Available from Intel Corporation, Santa Clara, California. 1981.

High-performance, high-capacity single-chip microcomputers

by ED PEATROWSKY
Motorola Inc.
Austin, Texas

ABSTRACT

The MC6801 Single-Chip Microcomputer has long been recognized as a high-performance microcomputer. This paper provides a brief look at the complete M6801 family and then discusses the enhancements made to the Timer and Serial Communications Interface circuitry of the basic MC6801 to develop the new MC6801U4 microcomputer.

The MC6801U4 strengthens the M6801 family position in the high-performance single-chip microcomputer marketplace.

INTRODUCTION

The past several years have brought about expanded markets for Single-Chip Microcomputers (MCUs). Some of these new markets are demanding higher-performance MCUs for future products. Higher performance does not mean merely an internal memory map expansion; it also means improved features and functions, along with versatility in application.

Requirements in industrial control, communications, automotive, and many other such applications are constantly demanding higher-performance MCUs.

The M6801 family has met this high-performance and versatility need and continues to improve as its product portfolio grows. The M6801 family follows the compatible evolutionary expansion that was established throughout the development of the M6800-based microprocessor family.

Table I shows the products in the current M6801 family and the basic features associated with each member.

VERSATILITY

The M6801 family has the ability to operate in two worlds—as a microcomputer or as a microprocessor. The fundamental operating modes of the members in the M6801 family products are these:

1. Single-chip
2. Expanded nonmultiplexed
3. Expanded multiplexed

Within these fundamental operating modes the resources of the microcomputer are briefly summarized in the following paragraphs and allocated as shown in Figure 1.

Single-Chip

In the single-chip operating mode the MC6801 operates with all internal memory resources. This operating mode makes maximum use of the input/output capabilities with no address or data buses.

Expanded Nonmultiplexed

The expanded nonmultiplexed operating mode uses internal memory resources and allows the modest increase of 256 bytes of read/write locations. This mode uses separate data and address buses, thereby reducing the number of input/output functions available.

Expanded Multiplexed

The expanded multiplexed operating mode removes some or all of the internal memory resources and allows the MC6801 to function as a high-performance microprocessor. In this mode the external address space can be expanded up to 64K bytes for external resources.

THE ENHANCED FAMILY ANSWER

The M6801 family is continuously growing. The latest member is the MC6801U4, which is an enhanced MC6801 that is pin- and object-code-compatible. All addressing modes and features of the MC6801 remain intact. The enhancements are increased ROM, increased RAM, and improved Timer and Serial Communications Interface circuitry.

Where the additional features of the MC6801U4 require additional input/output, more of the port pins have been made multifunctional, as shown in Figure 2.

The internal ROM of the MC6801U4 has been doubled in size, from 2048 bytes to 4096 bytes. The interrupt vector locations are maintained as in the MC6801 for compatibility.

The internal RAM has been increased from 128 bytes to 192 bytes. The standby RAM portion of this memory has been decreased from 64 bytes to 32 bytes. This decreases the amount of standby current required to maintain the memory contents during power down.

TABLE I—The M6801 family

Single-Chip Microcomputers				Microprocessors			
Feature	6801	68701	6801U4	Feature	6803	6803E	6803U4
ROM size	2K bytes	—	4K bytes	RAM size	128 bytes	128 bytes	192 bytes
EPROM size	—	2K bytes	—	Stdby RAM size	64 bytes	64 bytes	32 bytes
RAM size	128 bytes	128 bytes	192 bytes	I/O lines	29 I/0 2 ctrl	29 I/O 2 ctrl	29 I/O 2 ctrl
Stdby RAM size	64 bytes	64 bytes	32 bytes	Timer	16-bit/3 funct	16-bit/3 funct	16-bit/6 funct
I/O lines	29 I/O 2 ctrl	29 I/O 3 ctrl	29 I/O 2 ctrl	SCI/baud rates	Full/4 selec	Full/4 selec	Full/8 selec
Timer	16-bit/3 funct	16-bit/3 funct	16-bit/6 funct				
SCI/baud rates	Full/4 selec	Full/4 selec	Full/8 selec				

```
Single-Chip (Mode 7)
  128 bytes of RAM; 2048 bytes of ROM
  Port 3 is a parallel I/O port with two control lines
  Port 4 is a parallel I/O port
Expanded Non-Multiplexed (Mode 5)
  128 bytes of RAM; 2048 bytes of ROM
  256 bytes of external memory space
  Port 3 is 8-bit data bus
  Port 4 is an input port/address bus
Expanded Multiplexed (Modes 0,1,2,3,6)
  Four memory space options, (total 64K address space)
     (1) Internal RAM and ROM (Mode 1)
     (2) Internal RAM no ROM (Mode 2)
     (3) No internal RAM or ROM (Mode 3)
     (4) Internal RAM, ROM with partial address bus (Mode 6)
  Port 3 is multiplexed address/data bus
  Port 4 is address bus (inputs/address in Mode 6)
  Test Mode (Mode 0):
     May be used to test internal RAM and ROM
     May be used to test Ports 3 and 4 as I/O ports
     Any mode can be irreversibly entered from Mode 0

  Resources Common to all Modes:
    Reserved Register Area
    Port 1 Input/Output Operation
    Port 2 Input/Output Operation
    Timer Operation
    Serial Communications Interface Operation
```

Figure 1—Summary of M6801 fundamental operating mode resources

TIMER

The timer features and registers of the MC6801 have been maintained and expanded. Three additional registers have been added, along with an additional input capture register and two additional output compare registers. Figure 3 is a basic block diagram of the MC6801U4 timer circuitry.

Dual Counter Register

The MC6801U4 has a duplicate timer control register. This Dual Counter Register allows software to examine the counter without the resetting of the Timer Overflow Flag in the Timer Control and Status Register.

Timer Control Register 1

A second counter register has been added, Timer Control 1, which allows the MC6801U4 to control the states of the pins associated with the output compare and input capture registers.

Timer Control Register 2

Timer Control 2 has been added for handling timer interrupts from the output compare and input capture registers. This allows software testing of the timer counter without clearing any of the associated status bits.

Input Capture Registers

A second input capture register has been added. The two input capture registers can be programmed independently to take a "snapshot" of the timer counter register at an appropriate transition on their associated input pin.

Output Compare Registers

The output compare feature has been extended by adding two additional output compare registers. These three registers can be programmed independently to respond to a match in the counter register and cause an appropriate transition on the associated output pin.

Serial Communications Interface

All the serial communications interface functions remain identical to those of the MC6801, and four more baud rates have been added. Table II shows the baud rates available for three given crystal frequencies.

SUMMARY

The MC6801 has been a leader among the available high-performance microcomputers in the marketplace for several years. The MC6801 continues to gain momentum in control and processing applications.

The enhancements added to the newest member of the family, the MC6801U4, allow the momentum already established by the existing MC6801 family of products to continue.

Diverse applications will continue to demand more and more powerful microcomputers. The MC6801 family products demonstrate that they are able to meet the challenge.

Figure 2—MC6801U4 8-bit microcomputer-block diagram

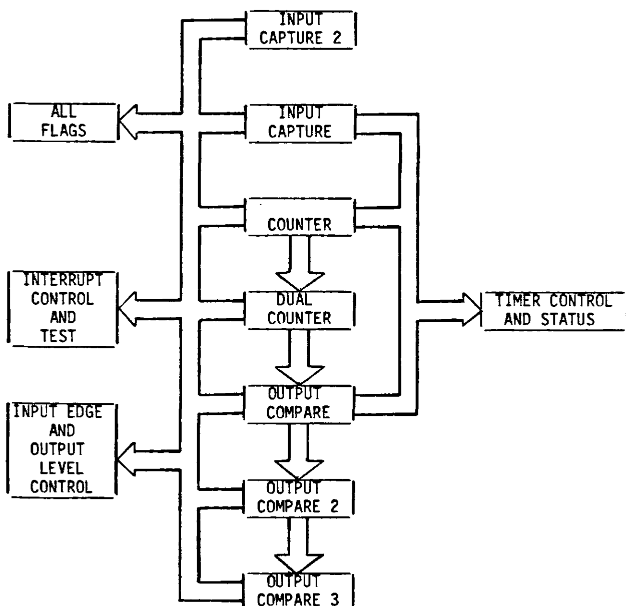


Figure 3—MC6801U4 timer-block diagram

TABLE II—Sci bit times and rates

		$4f_0 \rightarrow$	2.4576 MHz	4.0 MHz	4.9152 MHz
			614.4 kHz	1.0 MHz	1.2288 MHz
EBE	SS1:SS0	E	Baud	Baud	Baud
0	0 0	÷16	38400.0	62500.0	76800.0
0	0 1	÷128	4800.0	7812.5	9600.0
0	1 0	÷1024	600.0	976.6	1200.0
0	1 1	÷4096	150.0	244.1	300.0
1	0 0	÷64	9600.0	15625.0	19200.0
1	0 1	÷256	2400.0	3906.3	4800.0
1	1 0	÷512	1200.0	1953.1	2400.0
1	1 1	÷2048	300.0	488.3	600.0
External (P22)*			76800.0	125000.0	153600.0

*Using maximum clock rate

Expanded single-chip principles in practical application

by RANDY M. DUMSE
Rockwell International

ABSTRACT

For the past two decades the semiconductor industry has been in a headlong rush to pack more and more features on a single piece of silicon. The creation of the microprocessor as a single LSI device naturally gave inspiration for further advances. The microcomputer on a chip followed quickly and was again a technological stepping point rather than a final goal. New generations and process development variations made possible larger, faster, and more powerful systems on a chip. There is, however, a limit on the amount of CPU, ROM, RAM, and special-purpose devices that can be placed on a single, easily manufactured silicon die with current technology. In order to give the cost-reducing features of a one-chip computer with the flexibility of a multichip set, the expanded single-chip computer was developed. This paper will explain the theory behind that development, and then explore its application in a specific example.

INTRODUCTION

For the past two decades, the semiconductor industry has been in a headlong rush to pack more and more features on a single piece of silicon. The creation of the microprocessor as a single LSI device naturally gave inspiration for further advances. The microcomputer on a chip followed quickly and was again a technological stepping point rather than a final goal. New generations and process development variations gave larger, faster, and more powerful systems on a chip.

Along the way, many applications already designed in multichip systems were redesigned using advanced generation single chippers to take advantage of substantial systems cost reduction. Other designs which were not cost-effective previously in multichip versions were plausibly marketable, with one-chip computers providing substantial reductions. Of course, there are some applications where only a one chipper will suffice due to size, weight requirements, etc. The solutions in these areas using single-chip computers have grown in number and complexity as the more sophisticated parts have become available.

Still, between the realm of what has been and what could be, current applications of microcomputers to the world in which we live have barely scratched the surface. The future will bring new and exciting designs. These designs will make possible consumer products that will challenge the imagination of man while easing his burdens.

Limiting factors

A closer look at the reason we are no further along in that endeavor will show three major facets that regulate advances. The first is time. Time moderates progress in several ways. Most obviously, as new microelectronic devices are perfected by the semiconductor manufacturers, there will be an appreciable delay before ideas on their use come into hand. Investors, engineers, and entrepreneurs will come together within the business world and move their dreams from design to production and distribution. The span between concept and product is time. It is less apparent, however, that time not only modulates the activity of these people but also their numbers. Educational systems cannot keep pace with the production of industry. There are more positions needing design engineers than there are design engineers. It is important to remember the reason for this phenomenon. Industry has found more ways of condensing features and functions on a piece of silicon than educational facilities have found to cram equal amounts of understanding of the use of these features into a single human head.

The second factor controlling progress is the level of advancement of the currently available microcomputer hardware. Along the scale of what can be implemented (in at least some form of computerized electronic system) and what cannot, single-chip computer systems fall far short of center. The reasons are obvious. There are only so many CPU, ROM, RAM and special-purpose devices that can be placed on a single, easily manufactured, silicon die with current technology. Certainly, technology will increase production capabilities, but it is probably unreasonable to expect a single-chip microcomputer with over a kilobyte of RAM in the next two years, for example.

The last controlling factor to be mentioned is, of course, cost. The principle of anything which costs nothing and does everything will make the inventor a millionaire applies here. Overall system cost has limited many ideas from becoming realities. Certainly, if electronic calculators were still being done in costly multiple LSI sets, there would be several orders of magnitude fewer of them in the world today. Many applications which will become commonplace are unknown today because of cost.

Stretching the limits

Although time is an uncontrollable factor, system sophistication and cost factors are not. A closer examination of both is warranted. First, it should be pointed out that, to date, these two items have been counter points. System sophistication could not be improved substantially while independently reducing cost (at least while remaining at a given technological level).

Sophistication is generally improved by the addition of features. These may include new instruction sets or even revised architectures in the CPUs, more RAM and/or ROM, more input/output lines, addition or expansion of special purpose devices such as counter/timers, edge sensitive lines, latches, PLA's and the like. Almost all of these added features require their own portion of silicon. The more silicon per chip, the greater the likelihood of a small imperfection ruining that entire chip, resulting in lower numbers of good parts (from both less die per wafer and a higher degree of failure) and increased cost per chip.

Costs are generally held down with several techniques. The cost of the single-chip computer itself may be insignificant compared to that of the overall system. The amount of support hardware surrounding the microcomputer will to some degree be determined by the complexity of the applications. It is not always as obvious that the microcomputer itself may determine the cost and complexity of the support devices.

Internalizing more functions in implemented hardware or programmed software will reduce production costs. Of course, the programming required by such an approach will probably increase the engineering effort, but this added cost can be amortized over the production run.

Ideally it would seem every possible combination of ROM, RAM, I/O and special purpose devices like A to D converters, etc., should be included on a one-chip if maximum cost savings are to be realized. The assumption is based on a false economy, since such a device would be too large to manufacture with current technology or unique enough to have only one possible user. Remember, the main reason for using a microcomputer over discrete logic is the cost savings found in doing a custom programming of an existing part over a custom layout of a new logic design. It is the case then that an optimization between the device manufacturer and user must occur if both are to realize maximum profit (from reduced cost). The manufacturer should offer only a few options of microcomputers, the range of which combines the most often desired features in the best proportion for most users. This will ensure high volumes and low prices for the parts. There will, of course, be applications where these high-volume-oriented designs simply do not have the resources to handle the job. Now the cost balance between a custom-chip or a multiple-chip set must be made.

EXPANDED SINGLE-CHIP PRINCIPLES

The above discussion highlights the need for a compromise between single-chip and multiple-chip sets. A scheme is needed to give the cost-reducing features of a consumer one-chip computer with the flexibility of a multichip set. If an external bus structure were available on a single-chip computer, the problem would be solved. When the microcomputer did not have sufficient internal ROM, RAM, I/O and/or special function devices, they could be added externally. Cost would be held down by virtue of the fact that only the extra devices needed would be added externally, reducing chip counts.

This is exactly the principle of expanded single-chip computers. Designs already exist that not only incorporate a good deal of computing power on a chip with the support devices for most common applications included internally, but also allow flexible expansion externally. Two such microcomputers are the Rockwell R6500/1-11 and the R6500/1-41. A detailed look at these devices is in order.

The Rockwell R6500/1-11

The R6500/1-11 (called the R-11 hereafter for simplicity), is one of the most advanced multifeature one-chip microcomputers available commercially. Based on an enhanced version of the R6502, the part has an extremely powerful 8-bit CPU with four new instruction set groups added. These groups are Set Memory Bit (SMB), Reset Memory Bit (RMB), Branch on Bit Set (BBS), and Branch on Bit Reset (BBR). These new instructions, coupled with the parts, high level of throughput (one μs minimum instruction cycle time), give an I/O intensive and very powerful general purpose microcomputer. A generous portion of 3K bytes of ROM is designed into the chip Also, 192 bytes of RAM are provided. In the 64 pin QUIP up to seven I/O ports are available, each with 8 individual lines for a total of 56 lines. Four of these lines can act as edge sensitive inputs. A complete, double buffered, full duplex, advanced feature serial channel is incorporated in the part. It will operate either synchronously or asynchronously. The inclusion of two 16-bit timers, one with a 16-bit latch and one with two 16-bit latches with multiple modes, gives the device many real-time signal processing and generation capabilities. This brief listing does not mention all the features of the R-11 but does point out that the designers included as much capability on a single chip as is feasible. To accommodate applications where these features are not sufficient to meet the product designers' needs, they also included two external bus modes that are program selectable so that external parts could augment a one-chip microcomputer.

The first of these modes, the Abbreviated Mode, provides an external data bus and six address lines, as well as the control signals required to affect data transfers. This mode supports 64 external locations and is most suitable for the addition of memory mapped I/O or special function devices. The second mode, the Multiplexed Mode, provides fourteen addressing lines, eight of which must be latched, as they time-share with the data bus. This mode gives a 16K contiguous memory map external to the part. Any type of device such as ROM, RAM or special function I/O device could be accommodated singly or in combination.

The Rockwell R6500/1-41

The R6500/1-41 (called the R-41 hereafter for simplicity) is an interesting device which can be characterized as an Intelligent Peripheral Controller (IPC). Designed to reside on a host processor's memory or input/output busses, this device can be programmed to control a given set of preassigned real world tasks. Although it contains an enhanced 6502 central processor of its own, it would appear to the host as a special purpose input/output or control device, as would any other LSI controller device such as a floppy disk or CRT controller. Based on the control and data words written into the R-41, it could execute commands and sequences programmed in its 1.5K internal ROM. Also available are 64 bytes of RAM. Besides the three state port on the host bus, the R-41 can host up to 6 input/output ports or 48 individual I/O lines in the 64 pin QUIP version. Two of these lines have edge detect circuitry. Much like the previous generation R6500/1 single-chip computer, the R-41 also hosts a multifunction, multi-mode 16-bit counter/timer with full 16-bit latches. Like the R-11, the R-41 has two external bus modes of its own and can support other LSI controllers or memory in its own memory map. Although the R-41's external bus modes have the same names as those of the R-11, there is a slight variance in function between the parts. The Abbreviated Mode of the R-41 has four address lines and two control signals. This provides for 16 contiguous external memory address. The Multiplexed Bus Mode provides an additional eight data lines time multiplexed

on with the data bus. This provides a full 4K of external memory map for RAM, ROM or devices.

EXPANDED SINGLE-CHIP APPLICATIONS

To highlight these expanded single-chip computer principles, a specific example of possible application will now be explored. Consider the current market state of electronic typewriters. Most are still largely mechanical with servo enhancement of the operators keystrokes. A great deal of mechanical complexity could be replaced by microprocessor logic and a cost savings realized. In all likelihood, improved features could be added with little additional effort. A specification will be formulated in the following paragraphs to make good use of the R6500/1-11 and R6500/1-41 features in this application.

Application specifics

The actual printer mechanism to be considered will be a daisy wheel type. The most basic of features will require scanning of the keyboard and control of the printer servo mechanical devices. The wheel motor and position timing, hammer timing and control, carriage positioning (left and right), and platen control (paper advance or positioning) are included. A typing speed of 10 characters per second more than covers the speed at which an above average typist could enter keystrokes. This would be the equivalent of about 120 words per minute, so this will be the basis for all timing specifications.

The print wheel timing and positioning could be accomplished in a number of different ways. Almost all of these combinations, however, fall into two categories, i.e., either a stepper motor to give a character change per step or a D.C. servo. Both would require a start point reference input.

Hammer control would be used once the daisy wheel was in position for the impact. Since different size letters would print with a different tonal intensity if the impact was not calibrated, the force used to strike the letter must be modulated. This might require combinations of different coils or energizing a single coil with different width pulses for the different characters.

The common type spacings are pica or elite, which place a character every twelfth or tenth of an inch. The common denominator between the two type sizes is 120th of an inch. Assuming a stepper motor was used to position the carriage, it would make 10 to 12 steps to move between character positions (1/60 of an inch is also possible).

Even if subscript or superscript positioning were required, the mechanics of the paper feed could be fairly straightforward and done with a single stepper motor.

To meet requirements of printing 10 characters per second, all the functions of paper movement, carriage positioning, wheel positioning, and hammer impact would have to be accomplished within 100 milliseconds. Of course, other functions would be going on concurrently. The keyboard must be scanned every 20 milliseconds or so in order not to miss any key closures.

Design details

Beyond the most basic requirements, many other features are possible when microprocessing power is added to the system. A single-line display, correction and editing of a line prior to printing, page-at-a-time memory, interfaces to mass-storage devices, and even computer interfaces are possible at very little cost difference over the basic typewriters. These additional features will make good examples of the expanded microcomputer principles and will, therefore, be included in this example specification.

This design will include, therefore, a single-line, 80-character display unit. It will allow an entire line to be entered on the display before it is printed. It will also be memory expanded and will "remember" an entire document, up to four pages of typed material. Once the document is in this memory, the typist will be able to review and make any corrections needed prior to reprinting. An RS232 channel will also be included to allow communications with a host computer or RS232 compatible mass-storage devices. It will, therefore, be useful not only as a typewriter but also as a computer terminal, a data recorder, and a limited-application stand-alone word processor.

Now that the requirements are stated, the details of the implementation can be revealed. Although there are many possible combinations of R6500/1-11's and R6500/1-41's that could meet these needs, the design selected here represents only one. It is offered only as a reasonable example. A single R-11 would host the entire system (see Figure 1). Support devices, as needed, reside on this part's external bus. In this particular implementation, the multiplex bus mode will be

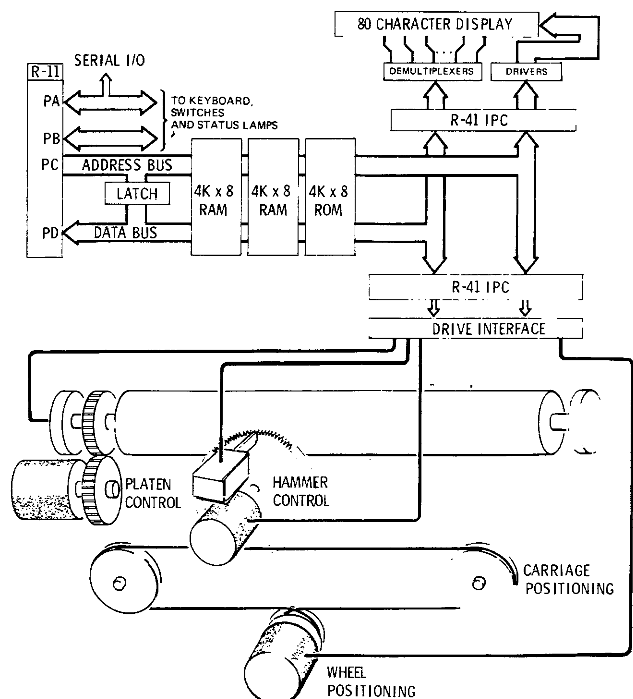


Figure 1—Typewriter block diagram

selected on the R-11 to allow a full 16K bytes of external memory to be addressed. In that address range of the system host will be 8K bytes of RAM, one ROM chip, and two R-41's.

System description

The tasks assigned to the host include the scanning and processing of all keyboard and panel switches, management of the RS232 serial port, maintenance of the entered document in RAM, performing the word processing functions, and commanding the actions of the two R6500/1-41's. One of these R41's is assigned to control all the stepping motor functions. The other is dedicated to the display. The tasks are organized in this manner to reduce the impact of small changes in the mechanics and display units on the overall system. This will allow future models to use more elaborate features in these areas without requiring any modification of the host system. Only the R41 involved with that portion would need reprogramming. As such, a great cost savings in new development could be realized.

The keyboard is a matrix of 47 alphanumeric key caps arranged in standard QWERT format with typewriter placement of the shifted characters (as opposed to teletype). Ten additional keys are required, comprised of the BACK SPACE, LINE FEED, RETURN, DEL, ESC, TAB, CTRL, LOCK and two SHIFT keys. A numeric key pad area and several selection buttons for system control are not required, but desirable for typewriter operations. Some of these additional keys can be in the matrix while others should occupy individual input positions. The CTRL and SHIFT keys are examples of the latter, since they will be closed simultaneously with other keys in the matrix.

Although there is some controversy about the type of rollover processing that is really required in a keyboard operated by a high-speed typist, N-key rollover is still the most popular and the reigning standard. Some terminal manufacturers are beginning to turn away from the concept, using 2-key rollover or lockout instead. N-key rollover programming requirements are considerably more complex for little or questionable performance improvement. Still, because it is the highest standard, the N-key design will be used in the example.

The RS232 port will be very easily implemented by using the serial port of the R-11. All the features of this channel are programmable to meet almost all common applications (including parity as required). When the typewriter LOCAL switch is active, the RS232 port could be connected to a selected RS232 compatible mass-storage device. Many such devices are available, the most suitable for this application probably being data cassette types. On command from the keyboard, the document contained in memory could be stored on the data tape. If it were desirable to review or edit it, the saved document could be retrieved later from storage. If not in the local mode, the key functions and printing would be independent. Keys depressed would be passed from the typewriter to an external device. The external device returns would be printed as received. This is exactly the essence of a full duplex terminal. Instead of a video screen for display, however, the output would be letter quality print.

Since internal RAM is limited to 192 bytes, it is necessary to expand the RAM with external parts. The internal RAM will be used for the processor stack and system constants such as tab settings, margins, etc.; and system variables will be used for calculations, keymask patterns, and the N-key stack. In order to provide N-key rollover, two images of the keyboard must be maintained. The differences from one scan to the next represent the new key information. As each new key is depressed, it is added to the N-key stack. A keyboard matrix can be rather large, so the two images will be stored in external RAM. In order to enter a line at a time and also do editing functions, a current line buffer will also be maintained in external RAM. A page of typed information requires 2000 bytes of storage or less, so 8K bytes are necessary external to the R-11 host.

The 3K bytes of internal ROM in the R-11 will be sufficient for the management of all features and communications with the possible exception of the keyboard key cap assignments for the SHIFT and CTRL combination, if the pattern is nonstandard, and perhaps some of the more complex text editing features that might be added. These features' programs could be maintained in an external ROM. It is doubtful that anything larger than a 4K byte ROM would be needed even if the features included centering commands and formatting with pagination functions.

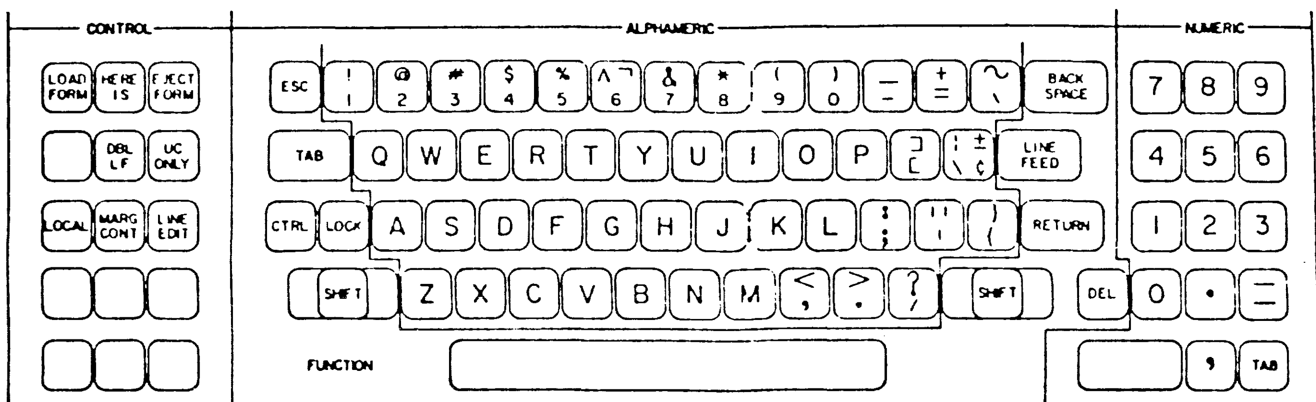


Figure 2—Example keyboard layout

All the features of the host have been described. Now attention will be turned to the slave processors functions. The two Intelligent Peripheral Controllers (R-41's) manage the output functions of the system. The first to be discussed controls the printer mechanism.

This R-41 would receive characters and commands through its data port from the host R-11's processor bus. The distinction between commands and characters would be made by the previously written control registers port. In this manner, the host would send characters in a stream to the R-41. The R-41 will determine spacing, paper feed, and print wheel and hammer control (in short, all the functions necessary to put the character to the paper). If special carriage control were required (line feed, carriage return, back space, superscript positioning, etc.), the host would send the specific associated control word instead.

The functions of the display could probably be processed by either the host R-11 or the printer R-41, but in order to give a flexible system design for future expansion as previously described, the second R-41 will be used to control the display. Such a design would also be advantageous by virtue of the fact that no additional I/O or timing burdens would be placed on the host or printer subsystems. After all, the simpler the modules, the quicker the system can be completed at a lower development cost.

The interface between the host R-11 and the display R-41 would be nearly identical to that of the printer slave processor. Buffered with transistors, some of the R-41's output lines could drive patterns for a long vacuum fluorescent display tube. Other port lines could be demultiplexed to give the select for a particular character position. Beyond these parts, only a power supply for the required V.F. display voltage would be needed to complete this module.

SUMMARY

The specification and design details are now complete, at least to the scope of this paper. Only six LSI MOS chips would be required for this entire system (an R-11, two R-41's, two 4K × 8 quasi static RAM and one ROM). The cost of the parts in OEM quantities for this application is under $50. The entire electronics assembly with display could probably be made for under $90, meaning it is reasonable to conceive of a high-quality typewriter, terminal text processor that could be marketed for under a $400 retail price tag.

The principle of using expanded single-chip computers to reduce costs is, therefore, proven. Development of a custom processor chip to handle all the features described could approach one million, if possible at all in current technology.

A multichip set approach would at least double the chip count and probably the cost of electronics, while offering no additional features. The application of expanded single-chip computers fits the needs of today's market.

Making the most of VLSI in microcomputers

by JERRY L. CORBIN
Texas Instruments Incorporated
Houston, Texas

ABSTRACT

An introduction to the innovative SCAT design philosophy for VLSI microcomputers of Texas Instruments (TI) is presented. The recently announced 8-bit TMS7000 Microcomputer family is used as an example of a SCAT design. TMS7000 benefits resulting from SCAT include a very dense bar for lower chip costs and microcomputer prices; a unique microprogrammability feature that will allow a user to modify the instruction set for the few applications that require it; and the architectural flexibility that will allow TI to bring many new microcomputer devices to the marketplace quickly and easily.

THE MICROCOMPUTER LAYOUT PROBLEM

Design techniques for large-scale integrated microcomputer circuits have traditionally followed those of printed circuits. Separate design teams typically pack desired functional performance into separate functional blocks. The job of interconnecting the functional blocks is left as the last step.

Thus, in comparison with memory chips, microcomputer designs tend to sprawl over large areas of silicon. As the complexity of microcomputers has increased, the interconnections between the various subunits can consume a significant portion of the available silicon. If random logic is used, its irregularity makes the problem worse.

SCAT ARCHITECTURE

Texas Instruments made an important step toward moving microcomputer design into the VLSI era with the introduction of the Strip Chip Architectural Topology (SCAT). SCAT integrates architecture and layout into a dense, memorylike, array-structured chip. SCAT replaces as much random logic as possible with regular structures such as read-only memories (ROM) and transistor arrays. TI's recently announced TMS7000 family of single chip, 8-bit microcomputers represents the culmination of the SCAT design philosophy.

With SCAT, the chip's layout is not left until the end of the design process, but is an integral part of it. For example, the TMS7000's registers for the timer, I/O control interrupt handling, and arithmetic logic unit are arranged in a strip. The chip appears to be a tightly stacked set of 8-bit-wide bricks that are interconnected through a data bus (see Figure 1).

Since the memory-intensive subunits are aligned in vertical strips, practically all the interconnection paths run over silicon that has already been used for active devices. The polysilicon and metal interconnections are made with an absolute minimum of signal path length, which also lessens the required size for the line drivers.

The net result of TI's SCAT is a very powerful microcomputer packed into a small chip size. The 2K ROM TMS7020 microcomputer, for example, has a chip area of 35,000 square mil using conservative 4.5 micrometer design rules that can easily be shrunk to 3.0 μM rules.

The costs of fabricating a microcomputer chip are exponentially related to chip size. For example, a microcomputer chip with only a 10% increase in silicon area (with the same design rules) could cost up to twice as much to manufacture! Small microcomputer chips equate to lower chip costs and thus to lower pricing to microcomputer customers.

MICROPROGRAMMABILITY

To take advantage of the silicon efficiency of ROM over random logic, TI replaced the traditional programmed logic array and associated random logic with a Control ROM to implement internal control of the TMS7000 microcomputer. The Control ROM stores the microcode that determines the instruction execution sequence.

Microcoding of the TMS7000 is extremely simple because of the general technique of instruction decode. The CPU has no microprogram counter; instead, the present Control ROM state supplies the address of the next state. With microprogramming, all the necessary control signals are contained in a single microinstruction lying lengthwise down the Control ROM. No complex routing or combinational logic is required. Most instructions executed by the TMS7000 share microstates with other instructions. This simple microarchitecture and microcode-sharing technique result in a reduced chip size while increasing tremendously the flexibility of the TMS7000.

Probably the single most unusual feature of the TMS7000 is the flexibility the microprogramming feature offers the customer. The already powerful standard TMS7000 instruction set can be altered or customized for applications that require unique performance, memory, or I/O features. These user-defined instructions are substituted for standard TMS7000 instructions on the Control ROM.

In some user applications, microprogramming will enhance TMS7000 performance. By combining or modifying the existing microinstruction execution sequence to perform critical tasks or subroutines in less instruction clock cycles, the throughput or "speed" of the TMS7000 in the user's application is enhanced.

Another advantage to microprogramming is that in specific applications it can allow more efficient use of the limited on-chip program memory. By combining or modifying the standard microinstruction execution sequence for unique repetitive tasks or subroutines, the total overall application program may require fewer steps and less on-chip program memory.

In effect, microprogramming can be also thought of as a safety net for the design engineer should he/she overestimate his/her software capability or underestimate the application system requirements.

Microprogramming could also be useful in providing increased system security for TMS7000 customers competing in very competitive business environments. Reverse engineering of a system implemented on a TMS7000 microcomputer with a unique user-defined instruction set would be difficult.

ARCHITECTURAL FLEXIBILITY

Because of the unique structure of the SCAT design philosophy, the orthogonal control and data paths are readily available to modify or enhance the TMS7000 chip.

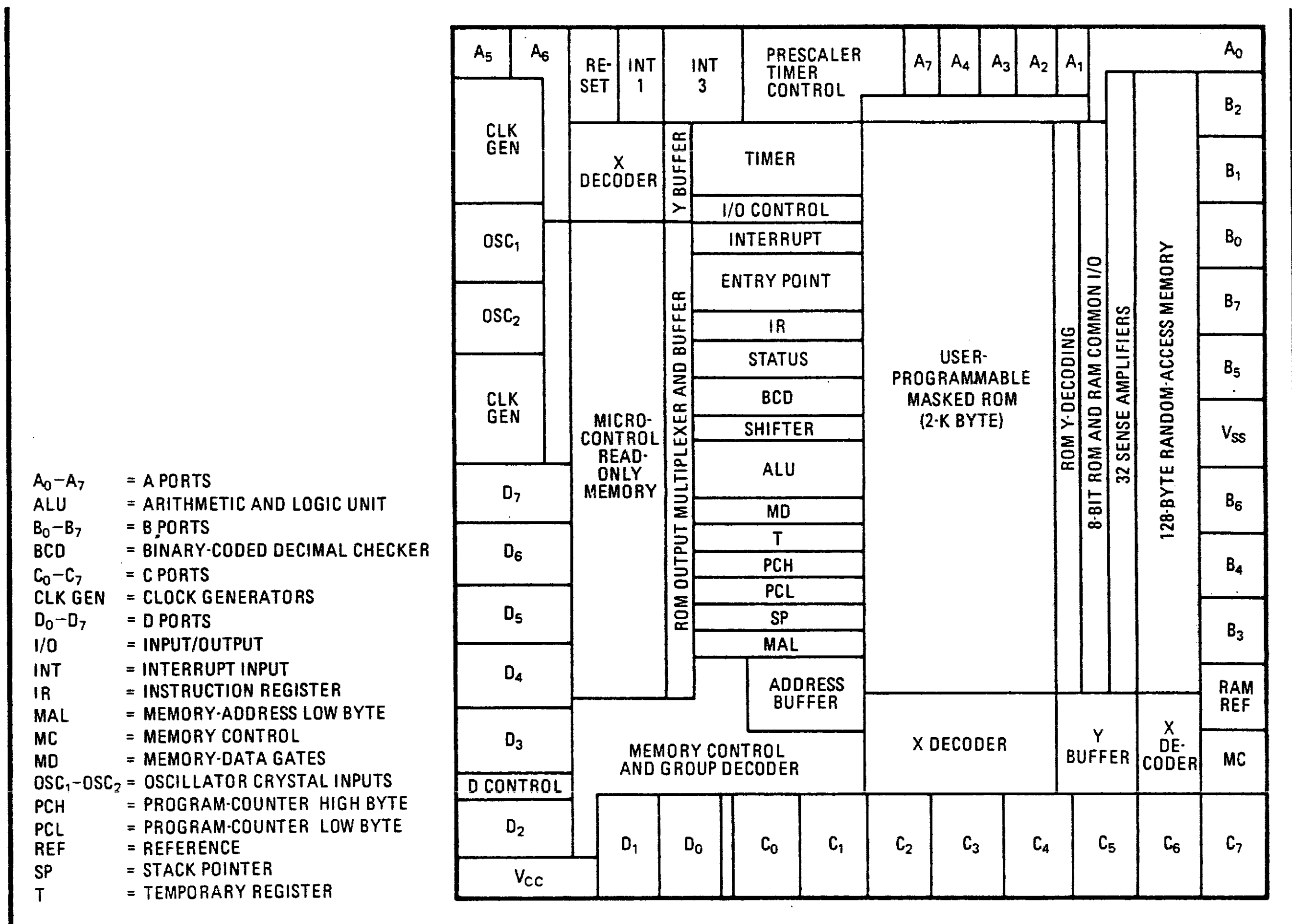


Figure 1—TMS7020 chip layout

For example, TI created the TMS7040 4K ROM version from the 2K TMS7020 2K ROM version without redesigning the chip. The chip design was separated at the memory border, and the additional 2K of memory was singly inserted by the design computer. Likewise, additional features such as more ROM, RAM, or different I/O structures can be added with a minimum of design resources and time.

TI plans to take advantage of SCAT by adding many device members to the TMS7000 family in the near future. EPROM, CMOS, communications devices, and more are in design.

Single-chip microcomputers can be easy to program

by BILL HUSTON
Motorola Inc.
Austin, Texas

ABSTRACT

Most single-chip microcomputers (MCUs) use the split-memory Harvard architecture. A few single-chips trace their architectural heritage to large computers due to the common-memory Von Neumann organization. The major differences are that a Harvard-based MCU costs less in its undistorted form, and a Von Neumann-based MCU is more expandable and easier to program.

Since the traits of Harvard-based single-chips are quite well known, though perhaps not by that name, the focus is placed on the programming benefits of a Von Neumann MCU. Programming costs can be lowered while increasing program reliability. Data organizations can be more flexible in both RAM and ROM. Program changes can be incorporated more quickly. The generalized instruction set is easier to understand. The M6805 family of MCUs is used to illustrate these benefits.

ARCHITECTURAL COMPARISONS

Like most major products, the single-chip microcomputer has evolved in a series of stages rather than being the inspired creation of a genius. All of the popular 4-bit single-chip microcomputers (MCUs) and many of the 8-bit MCUs are derived from the evolution of the calculator. Some 8-bit MCUs have instead evolved down from larger computers. These two diverse evolutionary paths are identified by comparing the two architectures that have resulted.

Harvard Architecture

The unique trait of the architecture shown in Figure 1 is the separate memory organization for programs (ROM) and data (RAM). Each type of memory has a dedicated address register. The ROM address register is the program counter, but the RAM address register has various names. Separate address

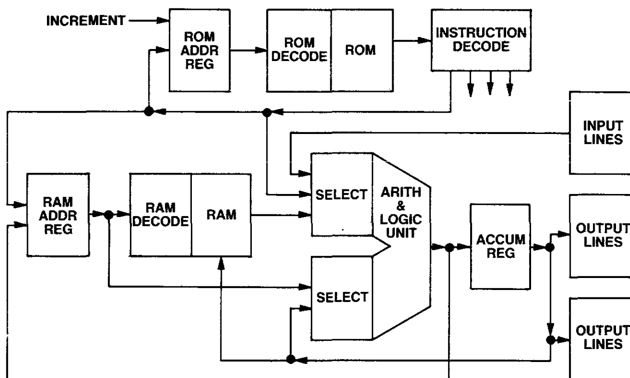


Figure 1. Harvard architecture single-chip MCU

registers permit register lengths and interconnections to be optimized. For example, a 6-bit RAM address can be used with a 10-bit ROM address.

With the separate memory architecture, data read from ROM are fed directly to the instruction decoder. Similarly, the RAM output goes only to the ALU. Thus, data widths of 4 bits in the RAM and ALU are not incompatible with an 8-bit ROM instruction size.

The split program and data memory architecture is sometimes called the Harvard architecture (or Aiken architecture). This designation contrasts it to the Von Neumann (or Princeton) architecture of all large computers today. The Harvard architecture was used in some of the very first electromechanical and electronic computers, built under the direction of Professor Howard Aiken at Harvard. Memory technology was of course very rudimentary in the 1940s. Since separate storage techniques were used for programs (paper tape) and data (telephone 10-step relays), the separate memory architecture was a natural. As with MCUs today, the hardware components and the interconnections are fewer with split dedicated memories.

The Harvard Mark I computer was used for over 10 years as a high-precision calculator of mathematical reference data such as navigation and ballistics tables. When the processor usage is straightforward, the Harvard architecture is fine, even superior. The problems arise when the needs become more complex.

For example, to allow a subroutine, a program counter save register is placed beside the PC. This does not dramatically disturb the interconnect efficiency of the Harvard architecture. The Harvard benefits dissipate quickly when three or more PC save registers are cascaded together into a costly amount of silicon. Sometimes the program is permitted to read and write into the PC save register, which adds more dedicated interconnects to Figure 1, as well as encountering the problem of unequal word sizes. Sometimes an MCU includes a stack pointer and saves the PC in RAM, which doubles the RAM read/write paths in Figure 1.

Address calculations are another example of Harvard architecture difficulties. Figure 1 shows that all MCU implementations have a path from the RAM address register to the ALU to permit calculations. RAM data structure sizes are limited when a 4-bit ALU is used with a 6-bit RAM address. Harvard MCUs use one or more instructions to calculate the content of the RAM address register. Then one or more instructions are used to obtain and operate on the RAM content. There are no single instructions that calculate the RAM address and then operate on the RAM content.

Some MCUs have no provision for calculating ROM addresses. The ROM address register is not available to the ALU, so relative addressing is not possible. In such cases it is not possible to read the content of a data table in ROM. Thus, a straightforward BCD-to-7-segment conversion has to be implemented in an I/O PLA. In some cases the Harvard architecture is further distorted to allow a program to read and write to a ROM address register. In such cases there are now two inputs to the ROM decoder in Figure 1, the program counter and a program-accessible ROM address register.

As the computer pioneers of the late 1940s and early 1950s discovered, the Harvard architecture has severe limits when it comes to generalized uses. Thus the Harvard architecture in today's more advanced single-chip MCUs includes numerous distortions. As a result, the economic motivation for the Harvard architecture in a calculator is lost in a general-purpose MCU. Extra dedicated registers and ALU data paths are added to the silicon area of an MCU, which increases the price. The Harvard architecture is also more difficult (expensive) to program.

It has been successfully shown with the M6805 family that a Von Neumann architecture MCU can be both lower in cost (less silicon die area) and easier to program.

Von Neumann Architecture

Figure 2 shows the fundamental architectural difference to be a common addressable area for RAM and ROM, and I/O as well. Rather than use point-to-point interconnecting as in Figure 1, Figure 2 shows common data and address busses. The program registers are also more generalized.

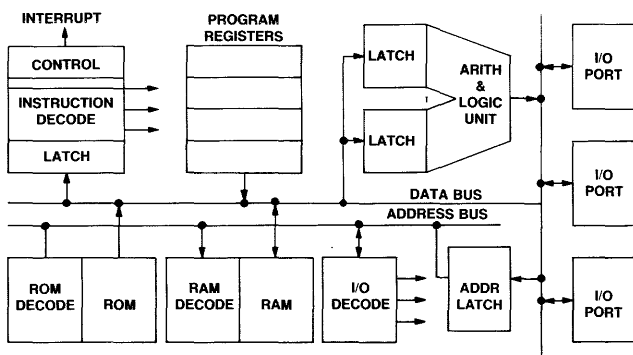


Figure 2. Von Neumann architecture single-chip MCU

Professor John Von Neumann at Princeton first documented the concept of a program stored in a common memory space with data. The chief benefit is the inherent ability to operate upon addresses as easily as data. Program and data table pointers can be saved in RAM. Indexing the other address calculations can be included.

The Von Neumann architecture has some shortcomings. The common bus saves interconnect area only when there are enough points tapping onto the bus to justify the three-state control needed to manage the use of the bidirectional bus. All address and data elements must be standardized to the bus width.

In current implementations, 8-bit busses, registers, and ALU are used, which means that some elements are larger than in 4-bit MCUs. Elements larger than the bus—addresses, for example—occupy more than 1 bus cycle. With an 8-bit bus, expansion to 16 bits of addressability is as easy as handling a 10-bit address.

The remainder of this paper focuses on the program benefits of the Von Neumann architecture, particularly as applied to the M6805 MCU family.

PROGRAM AND PROGRAMMER EFFICIENCY

It was once considered sufficient simply to have a very low-cost programmable IC. The programs written were short, and the programming effort was to be amortized over a large number of units. This view is obsolete today in many applications. The applications are more complex than the microwave ovens of a few years ago. Programs are not just written once and forgotten; they are changed, in some cases many times. Program changeability costs should also be considered when amortizing program costs.

The Von Neumann type of MCU architecture also permits greater program design flexibility. Memory use tradeoffs are more easily made. System hardware functions can be taken over by the program. The tools are available to allow programs to be more reliable. The most important efficiency factor for MCU programs is efficiency of ROM use—fitting the most features into a given ROM size.

Program Changeability

Only unsuccessful programs are never changed. Since a project is seldom started that is planned to be unsuccessful, all projects need to plan for program changeability. Field testing of a prototype points up faults in the original program as well as desirable improvements. The sources of program requirements (customers and marketers, for example) frequently conclude that what they asked for is not exactly what is needed. Similarly, the managers, marketers, and customers always come up with new features that would be desirable. These are just some of the sources of changes to the original product.

There are also changes to the program that generate derivative products. It is difficult to hide the fact that the single-chip is programmable. Everyone wants to take advantage of the programmable IC to suggest derivative products. Changeability must be designed in from the beginning.

Programming costs thus include the cost of incorporating program changes as well as the initial programming effort. Frequently the changes are incorporated by a different programmer. Program changeability costs thus also include the time it takes a new programmer to figure out what the original programmer did.

The MCU architecture can limit future extensions of the program to include additional functions. In such cases the program changeability costs include reprogramming for a new MCU. Specialized programming techniques that take advantage of odd MCU features or use unused memory in odd ways also limit future changeability. Major reprogramming costs can be avoided by using generalized MCU architectures, which do not tempt the programmer to use odd quirks in the inevitable attempts to get seven pounds of functions into a five-pound ROM sack. The features of the end product can be so tightly interwoven with each other and with the given memory organization that changes, even some apparently simple ones, can send the programmer back to Square 1.

The architecture of a single-chip MCU has more impact on the cost of program changes than at first suspected. The Von Neumann architecture allows programs to be written faster initially, understood more quickly by a different programmer, and changed more rapidly.

Fewer Lines of Code

"The programming time is directly proportional to the number of program statements."

This axiom has been widely accepted for programming projects, from compiler-language business-data-processing programs to assembly-language microprocessor applications. The axiom is also applicable to single-chips.

The functional definition, functional flow chart, and user documentation effort are rather independent of the MCU chosen. However, the detail flow charts, coding, program

checkout, and program documentation phases are proportional to the number of lines of code. In typical projects, coding and checkout represent the bulk of the programming effort.

If an MCU architecture permits the program to be written with fewer lines of code, it saves programming expense. Benchmarks have shown that that M6805 family programs need about half as many lines of code to accomplish a given task as a typical 4-bit MCU. The benchmarks include full applications as well as typical comparison subroutines. Thus 50% of the program coding and checkout time can be saved.

More details of the M6805 family architecture are included later, but a few of the features that contribute to the program savings are listed here. Address calculations, including table look-up indexing, are a part of the instruction, not separate instructions that must precede the operation. In two-operand instructions such as add, AND, and compare, one operand is an addressable memory byte, which saves frequent register loading. Memory bits and bytes can be modified directly, without disturbing any registers, in a single instruction such as set a bit and increment a byte. All I/O pins may be set, cleared, or tested with one instruction. Interrupts automatically save and restore all registers.

As applications become more complex, programming time is becoming a larger part of the end product cost. A larger benefit in many cases is that the end product will be available sooner. Many products using MCUs go into competitive marketplaces where saving a few months can measurably increase market share. When changes can be incorporated faster, the new product variations can also reach the market ahead of the competition.

ROM Versus RAM Tradeoffs

MCU programmers frequently get caught with not enough memory. Product cost targets can block switching to an MCU with more memory. So effort must be expended in redesigning the program until it fits.

When only ROM or RAM is overloaded, tradeoff techniques can be used to decrease the use of one at the expense of the other. The common memory field of the Von Neumann architecture is again shown to be an advantage. ROM and RAM are equally accessible, so functions can more easily be moved back and forth.

The flexibility of having any number of subroutine levels gives the user considerable control over the mix of ROM and RAM used. The more subroutine levels needed, the more RAM used for subroutine return addresses. So when spare RAM is available, the code can be shortened with more subroutines. When RAM is overfilled, fewer subroutine levels can be used by increasing ROM usage.

Efficient bit and byte handling instructions, such as that of the M6805 family processors, allow RAM data to be packed, multiple elements per byte.

I/O Versus ROM Tradeoffs

The increased instruction and addressing mode sophistication of a Von Neumann MCU sometimes allows previous hardware functions to be taken over by the software. Since hardware-versus-software tradeoffs are application-dependent, only generalized examples are cited.

Some MCU applications use an off-chip A/D converter. There are a series of alternative approaches that can be considered. One approach is to use an MCU that includes an on-chip A/D. Second, the analog value can also be converted to a variable frequency or pulse width, which is measured either with a timer on the MCU or with a program. A third method is to use an interrupt program to count the cycles it takes for an external ramp to match on an external comparator. Perhaps money can also be saved in the analog sensor or in the accuracy of the A/D conversion. A lower-cost sensor might produce nonlinear outputs, but the program could compensate for the nonlinearity by using an indexed conversion table or a smoothing formula.

The goal is the lowest total system cost, not the lowest MCU cost. There are frequently opportunities to consider doing by program functions that require external hardware with other MCUs.

Program Errors

Program reliability should be considered in relation to single-chip MCUs. It may seem improbable for an error to go undetected that is serious enough to require scrapping end products, but it has occurred. Such scrappage is part of the cost of programming. Software costs are treated as amortizable costs. The exception is program errors that turn into recurring costs. Program errors occur as a result of insufficient program checkout, which frequently is due to hurriedly incorporated changes.

Rather than initiating end product scrappage, program errors more often cause a quirk to show up in the end product. Such errors cause a series of recurring costs (costs proportional to the quantities in use, not one-time costs). Instruction manuals are expanded to explain the quirk. The service people are trained not to interpret the quirk as a failure. Time is taken to explain the quirk to complaining customers. These are direct, measurable costs of program errors.

An indirect cost of program errors is loss of good will. Customers who have to live with a recognized quirk are irritated. Some will take their business to a competitor the next time. These are not one-time programming costs.

Program unreliabilities also bring in the risk of legal liability. Some program errors could be construed as causing loss of life, limb, or property.

The use of sound programming techniques is clearly the best way to reduce the risk of program unreliabilities. The architecture of the MCU can contribute to encouraging good programming techniques.

Errors are inclined to be proportional to the number of lines of code it takes to write a given program. A processor that uses fewer statements to perform a function, is also easier to keep clear in the mind of the programmer. As implied earlier, orderly change incorporation presents the best opportunity to reduce the error risk. In this case, the otherwise unmeasurable factors of an easy-to-understand, consistent instruction set with few oddities has major value. When the application functions are tightly interlinked with memory and I/O traits, changes can be extensive and thus error-prone.

The watchword is to be sensitive to program reliability and to put some value on an MCU architecture that encourages better programming.

ROM Usage Efficiency

Using the least ROM area is one of the more important criteria used to select single-chip MCUs. The number of single-byte instructions in the repertoire is not a good measure of ROM efficiency. The question is not whether one thousand instructions fit into a 1K ROM, but rather the number of system functions that can be programmed into a 1K ROM. This brings up the subject of benchmarks.

It is tempting to gather or devise half a dozen routines that are felt to be typical of the intended application and implement them in two or three competing instruction sets. Such a tradeoff is vulnerable to human bias, perhaps unintentional, on two major fronts. First, the programmer is likely to be more experienced in one processor and thus less likely to produce optimal code on the alternate processors. Second, the choice of the benchmark routines is clearly a simplification of the application and likely to be slanted to the programming techniques used on one or a few processors.

In spite of the risks, comparisons obviously need to be made. Steps can be taken to reduce, as far as possible, these biases. But why not go one more step?

The initial writing of an MCU program tends to be short compared to programs on larger computers. Many single-chips have been programmed in a month or two. So if two MCUs are in contention, program them both for the complete application. Then the comparison benchmark is not just a few isolated routines, but also all the overhead that it takes to use those routines in a practical application. Small benchmarks can serve to evaluate speed-critical program paths in response-time-sensitive applications. But MCU users are usually more concerned with ROM efficiency than with throughput. ROM usage efficiency is not as easily judged from small benchmarks.

THE M6805 FAMILY ARCHITECTURE

In covering the benefits and shortcomings of Von Neumann-based single-chip microcomputer architectures, some of the architectural traits of the M6805 family of MCUs have been alluded to. This report is thus concluded with some details of the M6805 family architecture. How well have these MCUs capitalized on the shortcomings of the popular Harvard architecture MCUs? Is the M6805 family really easier to program, and does programming ease have monetary value? The result is an MCU architecture which is more economic (has a smaller die area) than the popular 8-bit Harvard architecture MCUs and at the same time includes the big-computer features that are usable in a single-chip.

Such programming tools as indexed look-up tables, many subroutine nesting levels, single-instruction memory modification, single-instruction bit test and modify, and common access methods for all addressable locations, are direct user benefits of the computer heritage as opposed to the calculator heritage. With these tools, programs are written easier and faster and are easier to modify and more reliable.

One Address Map

A striking feature of a Von Neumann architecture is the common memory space for the ROM and RAM. The M6805 family extends the advantage by allocating space in the address map for I/O registers. The common address map is shown in Figure 3. The instructions include short addressing modes for more ROM-efficient access to the first 256 addressable locations. The most frequently accessed data elements are thus concentrated in the quick-access 256-byte page zero. Present implementations include 64 bytes and 112 bytes of RAM in various versions, but future versions could easily include more or less RAM.

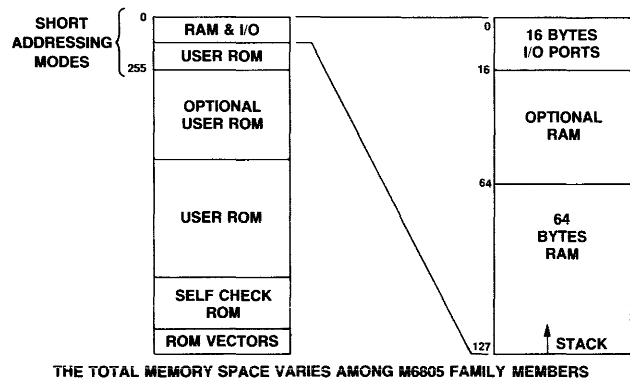


Figure 3. Common address map

ROM Areas

A portion of the user ROM is included in the first 256 locations to allow quick access to frequently used subroutines and to allow quick access to look-up tables.

In addition to the user ROM, all M6805 family ROM-based MCUs include self-check ROM. A small program is included for factory wafer-level testing and is available for user testing if desired. The self-check ROM area is not counted as user ROM and does not in any way reduce factory final testing to data sheet specifications. Some users are using the callable self-check subroutines implemented in most versions for functional confirmation when coming out of reset. Some are using a low-cost self-check tester for functional screening of parts before PC board assembly. The EPROM versions do not use the small mask ROM for self-checking, but rather for bootstrap self-programming of the user EPROM.

The highest memory addresses are user ROM for the interrupt and reset vectors. The vectors are 16-bits (2 ROM bytes) designating the interrupt program starting address. Separate vectors are included for the external interrupt; the timer interrupt; the software interrupt; the power-up reset program; and, in the CMOS versions, the stand-by recovery (Wait mode) program.

Addressable I/O

The first 16 addressable locations are reserved for the on-chip I/O registers. I/O is thus accessible to all instructions using the ROM efficient short addressing modes. I/O data

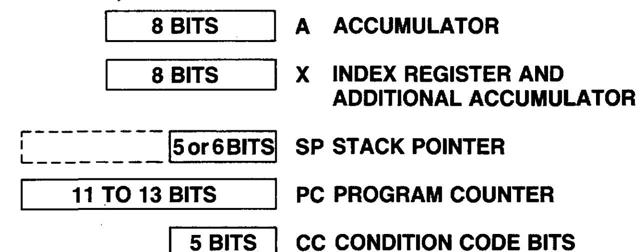
may be read or written (load and store) as bits or bytes. But I/O bytes may also be operated upon (AND, add, compare, etc.).

Current MCUs include up to four 8-bit ports. Each port read/write register occupies 1 memory byte. The ports include a second byte, the data direction register, which determines whether each I/O pin is an input or a driven output.

The 4 ports thus occupy 8 addressable bytes. The timer accounts for 2 more bytes, one for the 8-bit counter and the other for timer control. The second external interrupt available on some versions occupies 1 byte. The A/D converter on some versions uses 1 byte for the digitized result and 1 byte for A/D control. The EPROM versions include a register to control the self-programming of the EPROM. One family version includes an on-chip phase-locked loop for frequency synthesis that uses 2 I/O bytes for the variable divider.

The Register Set

Figure 4 shows that when a generalized address map is used, only five program registers are needed to provide a powerful instruction set. The specialized registers of the Harvard-type architecture are not needed.



S AND PC LENGTHS VARY WITH THE AMOUNT OF MEMORY IMPLEMENTED

Figure 4. Register

The accumulator is used for arithmetic and logical operations. The program counter is from 11 to 13 bits long, depending on the amount of memory implemented.

The index register has two uses. The three indexed addressing modes use X to contain a variable that is added to a value provided within the instruction. The X register is also an auxiliary accumulator. Many of the register manipulation instructions that operate on A also are used with X.

Additional general-purpose registers are not needed, since instructions are available to modify RAM locations directly without disturbing A or X. Examples are increment a byte, set or clear a bit, and test a bit or byte.

The stack pointer is initialized to the highest RAM address. The variable portion is 5 or 6 bits to limit the maximum stack length to 31 or 63 bytes. A subroutine call uses 2 stack bytes to save the return address. The automatic interrupts use 5 stack bytes to save the A, X, PC, and CC registers. The 5-bit stack pointer thus permits up to 13 nested subroutines, assuming 1 interrupt level, $(31-5)/2=13$. The 6-bit stack pointer allows for 29 subroutine levels. Both subroutine nesting levels are safely beyond that which could normally be used in a single-chip program. It is convenient, however, to let the programmer determine the needed subroutine levels rather than have the limit established by the architecture.

The condition code register is five individual status bits that are treated as a register when an interrupt save occurs. Four of the CC bits represent the results of the last data byte accessed or register operation performed. These permit subsequent testing with conditional branch instructions. The four result conditions are carry (or borrow), half carry (for BCD adds), all zeros byte, and negative (bit 7 set). The fifth CC bit is the interrupt mask, which enables all on-chip interrupts.

Future Expandability

A frequent restriction of Harvard architecture MCUs is a limit on expanding the memory or I/O size in future versions. In most cases the maximum RAM size is limited within the op-code field of instructions that load the RAM address register. There are a number of popular architectures that cannot use more than 64 bytes of addressable RAM.

A Von Neumann architecture has few restrictions on the mix of ROM and RAM. The only address limit imposed by the M6805 family architecture is that the maximum addressability is 64K, though no current versions include a full 16-bit address. The program counter, all the long addressing mode instructions, and the subroutine and interrupt save space all accommodate a 16-bit address field with no architectural changes.

Numerous System Configurations

A major benefit of architectural expandability is that many family versions can be introduced in a short time. Eleven versions of the M6805 family are already available, and more are on the way.

Three technologies are presently represented: HMOS, CMOS, and EPROM. ROM sizes range from 1K to 4K, with RAMs from 64 to 112 bytes. The 28- and 40-pin packages typically permit 20 and 32 I/O pins respectively. For evaluation, prototyping, and smaller production runs, both EPROM and ROM-less versions are offered. Some versions include an on-chip 8-bit A/D converter. Another includes a frequency synthesizer for RF applications. Standby RAM capability is included in some versions. Most include high-current output drivers.

Automatic Interrupts

Interrupts are the primary tool allowing a program to synchronize to real-time I/O events. Single-chip MCU applications have become I/O-intensive. Inputs and outputs of diverse natures must be accepted and generated. Frequently, tight timing relationships must be measured or maintained. Multiple timing relationships must be coordinated, sometimes at higher speeds.

Some Harvard-architecture-based MCUs have no interrupt facilities because there is no place to store the return address. The modernized Harvard MCUs have added an interrupt, which is frequently only a fixed subroutine call. Fully automatic interrupts save all program registers, not just the pro-

gram counter. The interrupt program thus need not waste ROM bytes and time storing all of the registers.

Efficient interrupt tools make complex real-time MCU interfaces possible.

Ten Addressing Modes

Another benefit of the Von Neumann architecture is that the common address map allows the instruction set to be enhanced by providing more addressing modes.

Figure 5 shows that the M6805 family has added four addressing modes to the M6800 instruction set while dropping only one 16-bit mode. The new bit manipulation capability is particularly appropriate to the controller environments that use single-chip MCUs. The extra indexing modes ease the table look-up task, the most useful indexing function in controllers, as well as permitting better ROM use.

ADDRESSING MODE		MC6800 MC6801	M6805 FAMILY
INHERENT (OPERAND IN OPCODE)		✓	✓
IMMEDIATE (OPERAND FOLLOWS OPCODE)	8 BITS	✓	✓
	16 BITS	✓	—
ABSOLUTE (OPERAND ADDRESS FOLLOWS OPCODE)	256 LOCATIONS (DIRECT)	✓	✓
	64K LOCATIONS (EXTENDED)	✓	✓
RELATIVE	PC ±128 (BRANCHES)	✓	✓
INDEXED (FOR TABLE ACCESSES)	EA = X	—	✓
	EA = X + 8-BIT VALUE	✓	✓
	EA = X + 16-BIT VALUE	—	✓
BIT MANIPULATION	BIT SET/CLEAR	—	✓
	BRANCH ON BIT	—	✓
		7	10

EA = EFFECTIVE ADDRESS
X = INDEX REGISTER CONTENT

Figure 5. Ten addressing modes

The inherent addressing mode includes the single-byte register reference and control instructions, which do not reference memory. Immediate addressing is the inclusion of an 8-bit data value in the second byte of a 2-byte instruction.

Short and long absolute addressing, called the direct and extended modes, includes the memory address in the instruction. The first 256 most frequently accessed bytes, the RAM, I/O, and part of the ROM, are accessed with a 2-byte instruction. A 3-byte extended instruction accesses any byte in the address map.

Relative addressing allows the conditional branch instructions to reach a program within the range of -127 to $+129$ of the instruction. An absolute jump can then reach anywhere else in memory.

The three indexed addressing modes add flexibility in the organization of the data in memory. In a single-byte indexed instruction, the effective address is the contents of the index register. The index register thus contains an 8-bit pointer to the data byte to be accessed. As such, the X pointer can reference any RAM byte, any I/O byte, or a portion of the ROM. This no-offset indexing is similar to the only available RAM access method on typical Harvard-architecture-based MCUs. The program calculates an address, puts it in a RAM address register, and then accesses the data. No-offset indexing is most frequently used in the M6805 family processor to scan down a data table looking at each entry.

The second and third indexed addressing modes are short and long table look-up indexing. The 8-bit contents of the index register is added to an 8-bit or a 16-bit value contained in the instruction to determine the effective address of the data to be accessed. In table look-up use, the instruction contains the address of the beginning of the table, and X contains a displacement into the table. Short offset indexing includes an 8-bit address within a 2-byte instruction; long indexing uses a 3-byte instruction to include a 16-bit table address. With short indexing the table must begin in the first 256 locations, but the displacement may create an effective address up to 255 locations beyond page zero.

Most microprocessors and 8-bit single-chip microcomputers have been good at byte manipulation. To be controller efficient, the M6805 family has added single-instruction bit manipulation and test capability. Any bit of any byte within the first 256 addressable bytes may be set or cleared. All the I/O pin and all the on-chip RAM bits may thus be individually changed. The addressed byte is read, the designated bit is changed, and the modified byte is written back into memory, all in one instruction. The two addresses—the direct (page zero) byte address and the bit address—are both contained in a 2-byte instruction. The read-modify-write cycle does not disturb the A or X program registers.

The second bit addressing mode is the single-instruction bit test capability. These are 3-byte instructions that include three addresses. First is the 8-bit direct address of any byte within the first 256 bytes. Second is a 3-bit address of the bit within the byte that is to be tested. Third is an 8-bit relative conditional branch displacement. One instruction is used to branch anywhere within the range of -126 to $+130$ locations of the instruction, depending on whether the designated bit is set or clear.

Instruction Set

The 10 addressing modes presented above bring much of the power to the M6805 family instruction set. The addressing mode flexibility allows many specialized instructions to be avoided. The instructions themselves are generalized; this feature, when combined with the addressing modes, produces a remarkably powerful processor in a small silicon area.

Except for a few miscellaneous instructions, all instructions are combined with one of the addressing modes to access memory. The 10 addressing modes combine with 59 basic instructions (61 instructions in the CMOS versions) to produce 207 total instructions (209 in CMOS). The programmer gets the power of 207 (209) instructions while having to learn only 59 (61) instructions plus 10 addressing modes.

The most frequently used M6805 family instructions are the memory reference instructions. Included are four move instructions, four arithmetic instructions, three logical instructions, three compare instructions, and two jump instructions. Except for the jumps, these are all two-operand instructions. One operand is taken from memory via the addressing mode, and the other operand is the A or X register. The result of the arithmetic and logical instructions is put into the A accumulator. The compare instructions perform a subtract (for magnitude compare) or an AND (bit compare) of the two values without modifying the registers or memory. Six of the major

addressing modes apply to each of the 16 memory reference instructions. Both short and long absolute addressing allows the memory operand (or jump address) to be anywhere in the address map and to be more efficiently accessed if within the first 256 locations. All three indexing modes are applied to all 16 instructions. An indexed table retrieval need not simply load a byte; it may also add, AND, compare, etc., a table byte with A. Immediate addressing is also usable with all the memory reference instructions, except the jumps.

Programming time is saved in several ways. Operations are performed during the same instruction as a memory retrieval (load). Magnitude and logical compares are accomplished without first saving the state of a register. Diverse memory data organizations can be used, since retrievals can use absolute addressing, register pointer indexing, or table look-up indexing.

The next class of instructions are the register and memory modification instructions. Included are the typical register manipulation functions of increment, decrement, complement, clear, shift, and rotate. A test without modifying is also included in this set. The unusual thing about these instructions is that they may be used to operate on memory data as well as both the A and X registers. An instruction like the memory increment can displace up to five instructions in another processor: Save the content of A, load memory byte, increment A, store incremented byte, restore previously saved content of A. All three short addressing modes are applicable to the memory modification instructions. Short absolute and both short indexing methods are included. Since ROM bytes are not modifiable, the long addressing modes have little use with these instructions.

The bit manipulation and test capability has already been covered. The four instructions are bit set, bit clear, branch on bit set, and branch on bit clear.

Ten of the 14 conditional branches test the condition code bits for the result of the last data operation. This set includes tests for zero, negative, carry, half carry, and above zero. The states of the interrupt mask bit and the interrupt pin are also testable. All these conditional branches allow branching on the true or false state. It is convenient that the branch is a relative arithmetic displacement (+ or −128 nominally), which has no page boundaries. In many MCUs the branch is permitted only within a fixed page.

The list of 13 miscellaneous instructions is short so that few specialized instructions need be learned. A regular (generalized, not specialized) register set and instruction set leave very few specialized functions to be performed. Six instructions are register reference functions: interregister transfers and the CC bit manipulations. There are four stack manipulation instructions associated with the interrupts and subroutines: return from subroutine and interrupt, call software interrupt, and reset stack pointer. The M6805 family versions implemented in CMOS include the Stop and Wait instructions.

Since CMOS ICs use dramatically less power when not operating, two program-initiated standby modes are included. The differences in the two modes are the conditions that cause the processor to resume execution. In the Stop mode the external interrupt pin causes the processor to restart. In the Wait mode either the external interrupt or the timer interrupt causes execution to restart. The timer interrupt permits the processor to be restarted at regular intervals. The timer interrupt can initiate a cycle consisting of scanning all inputs, processing the inputs, saving needed results, and generating needed outputs. When this cycle is complete, the processor can be put back into the Wait state. The battery drain is thus the average of the operating current and the stand-by current for the operating-to-stand-by duty cycle.

FULL PROGRAM PERFORMANCE

As single-chip microcomputer applications are becoming more complex, the real-time program needs typical of larger computers are becoming necessary.

Program costs must be kept down. The programs must be capable of being easily changed for future products, and easily documented to allow a different programmer to incorporate changes. MCU architectures can permit efficient ROM use. The classic computer types of architectures offer more tools for memory optimization. RAM usage and I/O features can be traded off with ROM use.

Generalized instructions with many addressing modes allow large-computer performance for an 8-bit MCU. Single instruction table manipulations are included in the M6805 family of MCUs. Single instruction memory bit and byte manipulations are included. Memory bits and bytes can be tested without disturbing the program registers. A common address map is used to allow ROM and I/O space to be accessed with as much flexibility and ease as RAM. The address map is designed for instruction-efficient access to the most frequently used data elements without making any memory inaccessible. There are no architectural restrictions on the amount of memory or on the implemented mixture of ROM and RAM.

The programmer's single-chips are Von Neumann architectures like the M6805 family.

Speak software and carry a strip chip

by MICHAEL SHAPIRO
Texas Instruments
Houston, Texas

ABSTRACT

A short description of TI's innovative Strip Chip Architectural Topology is given. The key features of the TMS7000 8-bit Microlanguage Processor are listed, and each of the current family members is discussed briefly. The architecture of the 7000 family is reviewed with emphasis placed on those aspects which enhance its programming power. Addressing modes and other software highlights are discussed in some detail, followed by an overview of microprogramming.

INTRODUCTION

In the 1970's the Texas Instruments team hit high and low, scoring points with both the budget-cutting TMS1000 4-bit microcomputer family and the cerebral TMS9900 16-bit microprocessor. While churning out yards of silicon in 4-bit slices (more than 70 million chips), we also introduced the industry's first 16-bit single–chip microcomputer—the TMS9940. Now, to center our offensive line, we have plunged into the 1980's with the innovative TMS7000 Microlanguage Processor family, our new 8-bit star.

TI had no intention of being a look-alike in a marketplace which already accepted several 8-bit architectures. Rather, by using a unique design approach to lower chip costs, and by implementing a rich instruction set to raise programming efficiency, we embarked on a third-generation design which is expanding into a powerful line of microcomputer products. This paper will touch first on the design concept and hardware features, concentrating later attention on the instruction set highlights and other software considerations.

SCAT—STRIP CHIP ARCHITECTURE TOPOLOGY

SCAT is TI's term for the design philosophy that incorporates the nonmemory elements of the microcomputer (the CPU registers, the ALU, the control logic) into a strip of vertical blocks in the logic design. Traditional design schemes have attacked the individual functional blocks first, leaving the problem of interconnect for last. Unfortunately, in the final layout, the interconnect often squanders the real estate prudently conserved in the early stages of design. To combat this profligate process, TI planned both architecture and layout from the beginning.

Figure 1 shows the layout of the TMS7020, the 2K ROM version of the TMS7000 family. By placing most of the random logic in the "strip," we were able to use control and data paths that interconnect the active elements but take up almost no additional silicon area. The logic of the elements in the strip is implemented on a low level of the silicon bar, whereas the data and address busses are constructed in metal over the silicon. This avoids the wasteful dedication of bar area to interconnect alone.

An additional space-saving feature of the SCAT design is the use of transistor arrays and ROM elements to replace random logic. Not only are these structures more compact, but the use of the micro-control ROM in place of the commonly used programmable logic array for the instruction decode allows the necessary control signals to be fed horizontally out of the control ROM right across to the strip. Torturous routing problems are avoided, and no additional combinatorial logic is required. A valuable by-product of this approach is microprogrammability, which will be discussed later in this paper.

TMS 7000 MICROLANGUAGE PROCESSOR FAMILY
TMS 7000/7020 MICROCOMPUTER DEVICE BAR PLAN

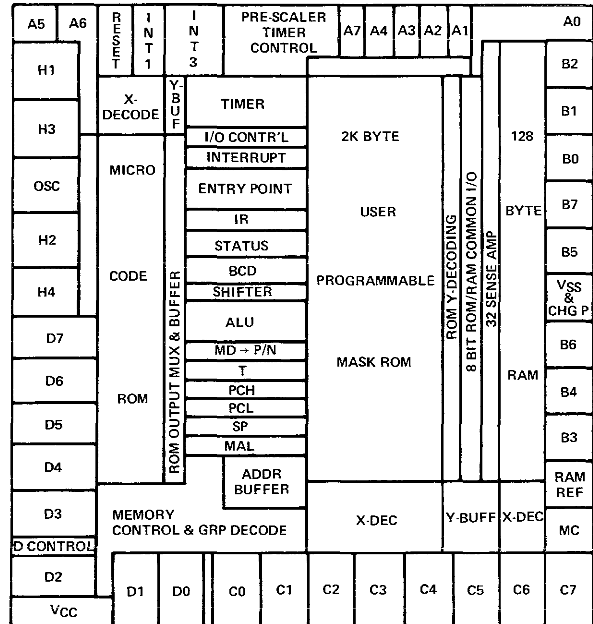


Figure 1—TMS7000/7020 microcomputer device bar plan

KEY ELEMENTS OF THE TMS 7000 FAMILY

The most attractive components of the TMS7000 family include the microprogrammed 8-bit CPU, addressing capability for up to 64K bytes of onboard and offboard memory, 32 individual I/O lines, multiple operating modes, unrestricted stack for control and data storage, 8-bit timer with presettable 5-bit prescaler, and four levels of vectored interrupt. The first family members have been implemented in high-density NMOS technology. CMOS and LMOS versions will follow in the months to come.

Family Overview

The TMS7000 family offers a variety of on-chip RAM and ROM configurations plus packaging and technology options to support the full scope of application requirements. The current family members include the TMS7000, 7020, 7040, 70L22, and the soon to be released 70E40.

The TMS7000 is a ROM-less device with 128 bytes of RAM. It functions as a powerful 8-bit microprocessor with on-chip RAM, interfacing to as much as 64K bytes of external memory on an 8-bit data system bus. The TMS 7000 provides eight input and four output I/O pins on the chip, each of which may be set, reset, and tested individually. Utilizing the 8-bit data bus, any of the common 8-bit I/O peripherals can be easily interfaced to the TMS7000 in order to expand its I/O capability.

The TMS7020 and 7040 are similar to the TMS7000 and contain the same CPU, RAM, and on-chip I/O when operating in the Microprocessor Mode. Moreover, these devices contain 2K and 4K respectively of on-chip ROM for application programming. The 7020 and 7040 may be configured in several memory expansion modes where memory interface pins are traded off for I/O pins. Besides the Microprocessor Mode, the other choices are as follows:

1. Single-Chip Mode providing 32 I/O lines
2. Peripheral Expansion Mode for interfacing to 8-bit peripherals
3. Full Expansion Mode to address 64K bytes of memory
4. System Emulator Mode for aiding program development

The most pertinent features of the TMS7020 and 7040 microcomputers are as follows:

1. Microprogrammed 8-bit CPU
2. 2048 bytes of on-chip ROM—TMS7020
3. 4096 bytes of on-chip ROM—TMS7040
4. 128 Memory-mapped registers (register file)
5. Multilevel program/data stack
6. 32 bits of general purpose I/O
7. On-chip 13-bit timer/event counter with interrupt and capture latch
8. Three maskable interrupts

The TMS70E40 is functionally identical to the 7040 except that the System Emulator Mode has been deleted and the on-chip mask ROM has been replaced by a programmable EPROM. One change has also been made in the instruction set to allow the 70E40 to program its own internal EPROM. This device is ideally suited for prototype fabrication or initial field testing of a new application prior to masked ROM volume production.

The TMS70L22 is a lower-cost alternative to the 7020, which retains most essential features, but gives up nine I/O pins to accommodate the smaller (and cheaper) 28-pin package. Processed in our power-saving LMOS technology, the 70L22 also works a trade on the clock frequency, operating at 1 MHz versus 5 MHz, achieving a tenfold reduction in power consumption. A new feature on the 70L22 is a slowdown mode that allows the user to further reduce current to accommodate applications in which power must be conserved.

Architecture

All members of the TMS7000 family incorporate features that take the best from both memory- and register-based architectures. The first byte in the RAM register file, Register A (R0), functions just like a dedicated accumulator to allow for faster access times and the 1-byte instructions that are inherent in a register type of machine. Similarly, the second byte, Register B (R1), can perform the task of a dedicated index register. However, the flexibility of the 7000 enables any one of the on-chip RAM bytes to assume the accumulator function by the addition of one byte to the instruction. True register-to-register operations can be accomplished throughout the 128-byte register space when a third byte is used in the instruction to specify the second operand.

Registers

The 7000 family has three hard-wired CPU registers accessible to the user. The 16-bit program counter (PC) contains the address of the next instruction to be executed. The status register (ST) contains three status bits that are used for conditional jump instructions. Also present in this register is the interrupt enable bit (I). The 8-bit stack pointer (SP) points to the top (last) entry in the data stack, and it facilitates multilevel subroutining and interrupts. The register file (RF) consists of 128 bytes of on-chip RAM.

Peripheral File

Beyond the memory address space devoted to the register file, there is a 256-byte region for memory-mapped peripheral input/output control, called the peripheral file (PF). The 32 bits of general purpose I/O, available in the Single-Chip Mode, are broken out into four 8-bit ports (see Figure 2) that can be manipulated via six dedicated peripheral instructions. Any of these bits may be individually set or cleared, or tested in conjunction with an appropriate bit-test-jump instruction.

Not only can the dedicated input (A Port) and output (B Port) ports be read from and output to, but the individual bits of the bi-directional ports (C Port and D Port) can be configured selectively as input or output by accessing their data direction registers (DDR), which also reside in the peripheral file.

To simplify use of the peripheral file, a special peripheral file-addressing mode was established to reference all 256 locations. Inputs and outputs on the I/O lines are accomplished by reading or writing to the appropriate port. For example, the B Port is implemented as port P6 in the periph-

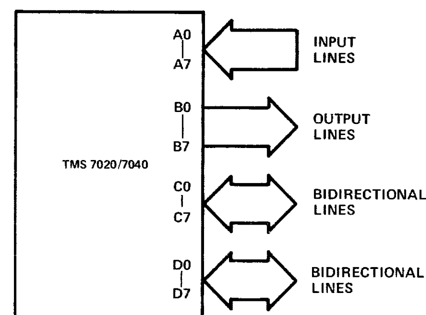


Figure 2—I/O ports in single-chip mode

eral file. Thus, writing to this port is handled by the instruction

```
MOVP    A,P6
```

which takes the value in the Register A (R0) and stores it on the B Port outputs.

In the Peripheral Expansion Mode, the peripheral instructions can be used to communicate with off-chip devices. When a memory address not corresponding to an on-chip port is used, the 7000 family device performs an external memory reference enabling an 8-bit peripheral chip to respond.

Timer/Event Counter

The 7000 family is equipped to handle real-time control applications by using a programmable 8-bit timer with a presettable prescaler value of from 1 to 32. As shown in Figure 3, the timer may use an internal clock source divided down or an external signal. On each positive edge transition of the clock input, the prescaler register is decremented. When the prescaler reaches zero, the decrement is performed on the 8-bit timer, and the prescaler is reloaded from the control latch.

As with the prescaler, the timer register will decrement until it reaches zero. The succeeding decrement will generate an interrupt (INT2), and the timer register will be reloaded from the timer latch. Since these registers reside in the peripheral file, the prescale latch value and the timer latch value may be written to, and the current timer value may be read using peripheral file instructions. Likewise, the timer on/off and the clock source bit are under program control in the peripheral file.

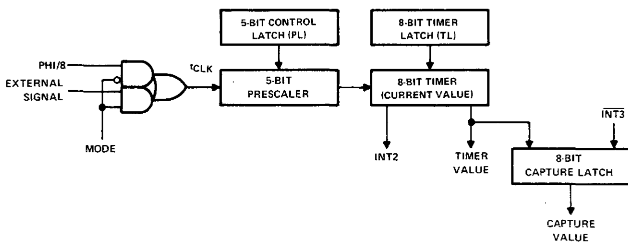


Figure 3—Programmable timer/event counter

In the event counter mode, the counter will function as described above, but the decrementer clock source will now be line A7 of the A Port. This timer mode can also serve the purpose of a real-time clock when an appropriate source is fed to A7. The A7 input can also be used as a positive edge-triggered interrupt by loading the prescaler and timer latches with 0.

A unique feature of the 7000 timer is the 8-bit capture latch, which saves the current value of the timer when an external (INT3) interrupt occurs. This allows the processor to determine precisely when the external event took place by comparing the captured value to the value that is now current. This capability can be essential if the external interrupt occurs while the processor is servicing a higher-order interrupt.

Interrupts

There are three levels of maskable interrupts: the INT2 associated with the timer and INT1 and INT3, which are externally triggered. The system reset cannot be masked, but the other three interrupts can each be enabled separately by bits in the I/O control register, and as a group by the interrupt enable bit (I) in the status register. When an interrupt is recognized, the contents of the status register and the program counter are pushed onto the stack. The processor then branches to the location stored in the corresponding interrupt vector location and starts execution of the interrupt routine.

Interrupts may be tested without actually recognizing them, allowing for greater user flexibility. Interrupts may be edge– or level–triggered, and no external synchronization is required. The signals are latched internally to catch short interrupt pulses.

The TRAP instruction can be used to create a "software" interrupt. There are 24 TRAP opcodes corresponding to 24 trap vector locations in the highest addresses of memory. As in an interrupt, the trap vector will provide a branch address at which a subroutine begins execution. Limitation on nesting in subroutines or interrupts is only a function of the overall stack capacity.

PROGRAMMING THE TMS7000

From the outset, the TMS7000 family was designed to optimize programming efficiency by virtue of its architecture and instruction set. The ease of access to the RAM, ROM, and I/O is achieved by mapping all of these into a single address space. Figure 4 illustrates the memory address scheme for the 7020/7040. This structure can be fully exploited by means of

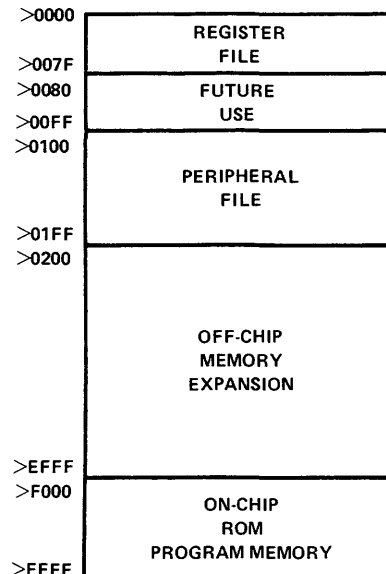


Figure 4—TMS7040 memory map

nine separate addressing modes. Add to this a full complement of standard instructions (the usual byte-oriented instructions plus multiplication, single- and multiple-bit tests, double precision arithmetic), and the design engineer has the upper hand in dealing with almost every application.

Addressing Modes

The nine different addressing modes for the TMS7000 family are listed below. The terms Register A and Register B are synonymous with the first two bytes in the register file, R0 and R1, respectively.

1. Register File—The byte(s) following the opcode specify any byte in the register file as the operand location(s). This includes single operand instructions such as

INC	R56	Increment the contents of R56
CLR	R99	Clear R99

   and dual operand instructions such as

ADD	R68,R45	Add R68 to R45 and store in R45

2. Register A—The operand location is implied, and R0 is fetched from the register file. This is a special case of register-file addressing, since Register A can be referenced implicitly as A or explicitly as R0; however, the implied mode saves a byte in the instruction. For example, the instruction

MOV	R20, R30	Move R20 to R30

   is three bytes versus two for the instruction

   MOV R20,A

3. Register B—The operand location is implied, and R1 is fetched from the register file. This is identical to Register A addressing except now B is the implied register.
4. Peripheral File—The byte following the opcode specifies a port in the peripheral file which contains the operand. These instruction mnemonics are identified by a P suffix. Each is a dual operand instruction with a peripheral file as the second or destination operand. Examples of these are

XORP	A,P3	Exclusive OR A with P3 and place the result in P3 (the timer control register)
MOVP	% > 60, DDDR	Setup bits 1,2 of D PORT as inputs

5. Direct—The two bytes after the opcode contain the address of the byte in memory that contains the operand. The notation for the direct memory address is the expression preceded by the @ sign. For example

LDA	@ > E34D	Copy the contents of memory location > E34D to Register A

6. Indirect—The byte following the opcode specifies the second of a RAM register pair which contains the address of the operand. This addressing mode is indicated by the * before the register as in the following instruction:

STA	*R19	Copy the contents of A into the memory location specified by R18 and R19

7. Indexed—The 16 bits following the opcode are added to the B register contents to form the effective address of the operand. The format for this instruction is given below.

BR	@HERE(B)	Branch to the address specified by the contents of B and the value of the symbol HERE

8. PC Relative—The byte following the opcode is used as a signed offset to the current PC to produce the effective address. This is the addressing mode used for all jump instructions, and it eliminates the designer's concern about where in ROM his program is jumping to, since the offset may lie anywhere in ROM.
9. Immediate—The byte following the instruction is the operand. For example, the instruction

ANDP	%COUNT, P10	Logically AND the value of COUNT and the contents of P10 and copy results to P10.

   illustrates the use of immediate addressing.

Because of the memory-mapped architecture, many modes can apply universally to any 16-bit address in the TMS7000 memory space. Thus ROM, RAM, or peripherals can be referenced with similar instructions possibly using common routines. The need for dedicated instructions in each category is now eliminated.

A very flexible feature of the 7000 is the capability of freely specifying two operands, the source and destination, within the dual operand addressing modes. While most microcomputers would restrict one of the operands to a particular register, the 7000 allows any RAM location to be named the source or the destination.

Instruction Set Highlights

As mentioned before, the TMS7000 family provides the full range of standard instructions. Rather than list the entire set, we will discuss some of the more unique members.

The MPY (Multiply) instruction takes the product of a general source and destination operand and places the 16-bit result in either A or B. The 7000 can perform this 8-by-8-bit

unsigned multiply in just 17.2 microseconds, assuming a 5 MHz clock.

The MOVD (Move Double) instruction is used to move a 16-bit value to a specified register pair destination. The source for this move can be an immediate constant, another register pair, or an indexed address.

The DAC (Decimal Add with Carry) and DSB (Decimal Subtract with Borrow) instructions provide the unique feature of performing fully corrected decimal addition or subtraction on two packed binary coded decimal (BCD) bytes.

The DECD (Decrement Double) instruction allows a 16-bit address to be easily decremented. This instruction can be especially useful for referencing tabular information in memory.

There are several jump instructions with especially useful test conditions to dictate transfer of program control. The BTJO (Bit Test and Jump if One) instruction looks at those bits which are 1's in the source operand and compares the corresponding bits in the destination operand. If any of these bits are also 1's, the relative jump is taken. There is a similar instruction BTJZ which does the comparison on bits which are 0's. These instructions allow for single- or multiple-bit tests.

Instructions as powerful as these are usually only available on more expensive high-end microcomputers (if at all). However, in the case where the designer has underscoped the task or runs up against a particular application intricacy, microprogrammability provides a possible out.

Microprogrammability

When TI implemented the TMS7000 instructions using a control ROM rather than random logic, it opened up the possibilities for user-defined "personalized" instruction sets, because the control ROM can be altered and then mask-programmed for production. Although the standard instruction set is very efficient for most applications, the user may find a repetitive program sequence of several instructions that could be reduced to a single command through microcoding. This would both increase throughput and reduce memory usage. Approximately 75% of the standard instructions are designated as core instructions and must be maintained. The remainder may be swapped out for user-created instructions which are customized to best serve that particular application. Software will soon be available to aid users in the design of microcode for a custom instruction set.

SUMMARY

This paper has attempted to give a broad overview of the TMS7000 family. We have given the reader only a brief taste (with software seasoning) of the capabilities available in the 7000 larder. In addition to the stock of products now available, we will soon be introducing a CMOS implementation and enhanced feature versions. To the hungry design engineer in search of a satisfying microcomputer—bon appetit!

A distributed operating system for a powerful system with dynamic architecture

by STEVEN I. KARTASHEV
Dynamic Computer Architecture, Inc.
Lincoln, Nebraska
and
SVETLANA P. KARTASHEV
University of Nebraska, Lincoln

ABSTRACT

The paper discusses the organization of a distributed operating system for dynamic architecture. It shows that the operating system must feature two types of distribution: (a) *functional* or vertical, whereby it is distributed among functional units in accordance with the types of conflicts that should be resolved; and (b) *modular* or horizontal, whereby it is distributed among modules performing the same functions.

In a dynamic architecture there are three types of conflicts: memory, reconfiguration, and I/O. This leads to the division of OS into three subsystems: (1) a processor OS that resolves memory conflicts, (2) a monitor OS that resolves reconfiguration conflicts, and (c) an I/O OS that resolves all types of I/O conflicts. The paper presents a detailed organization for the processor operating system.

1. INTRODUCTION

The architecture of powerful parallel systems (Supersystems) for fast real-time algorithms should take into account major peculiarities of these algorithms, as follows:

1. As a rule, each such algorithm is characterized by a large and variable number of concurrent instruction streams in the range of hundreds and thousands. Severe time restrictions imposed on some portions of these algorithms disallow their computation in an interrupted mode of operation. This leads to the necessity of having one computer dedicated to computing one instruction stream.[1] As a result, the Supersystem must incorporate hundreds and even thousands of computers; i.e., it must have an enormous complexity in order to be adequate for computations.
2. Typically, extensive data exchanges are required between information streams; i.e., the Supersystem must possess a very flexible, very fast interconnection network between its computers.

These two requirements are contradictory because of the reasons stated below.

To interconnect hundreds or even thousands of computers, the network must be multistaged, because the use of a single staged network (crossbar) becomes cost prohibitive as a result of the n^2 growth in the number of its connecting elements. However, a multistaged network becomes very slow as a result of the following undesirable characteristics:

1. It introduces long delays into signal propagation from one computer to another, since each data path will take $\log_2 n$ connecting elements. Since n is in the range of thousands, this delay becomes a significant factor in slowing down information broadcast between computers.
2. Since for a multistaged network one or more connecting elements may belong to several data paths, there arises the problem of blockages (conflicts) when several data exchanges use the same connecting element(s).

Therefore, multistaged interconnection networks create new types of conflicts in addition to conventional ones created in conventional multiprocessor systems during program competition for the processor and memory resources. As it turns out, multistaged networks require conflict resolution in data propagation through the connecting elements. A conventional way to solve these conflicts is in repeated data broadcasts of exchanges that are allowed. Therefore, not only is each data exchange slowed down by $\log_2 n$ connecting elements in its path, but it must be broadcast in several passes to eliminate blockages. In addition, extensive control overhead is created during each blockage, since the OS must analyze the priority of each data broadcast to find out whether it is allowed or prohibited.

These two characteristics of multistaged interconnection networks lead to a significant reduction in the Supersystem throughput, making it unsuitable for computing a number of fast real-time algorithms.

Another adverse factor is that the complexity of some real-time algorithms constantly grows as a result of technological progress. Therefore, in the future the problems discussed above will have an even greater effect on the performance of Supersystems.

The way out of these contradictions between the Supersystem throughput and its complexity lies in adopting the following design strategies:

1. To overcome significant throughput loss introduced by multistaged interconnection networks, it is desirable to partition the entire Supersystem into several subsystems, each of which uses one staged (crossbar) connection between its computers. This will eliminate long delays in both data broadcasts and blockages, since each data path will go through a single and dedicated connecting element. On the other hand, since each subsystem will have a much smaller number of computers (tens), the problem of complexity caused by the necessity of having n^2 connecting elements will also be significantly relieved.
2. To increase computational concurrency in each subsystem without augmenting it with additional computers, each subsystem must be provided with dynamic architecture. Indeed, as was shown by Kartashev and Kartashev and by Vick, Kartashev, and Kartashev[3,4,5] a dynamic architecture can maximize the number of instruction streams computed by the available resources. This means that a dynamic architecture creates an additional concurrency using available resources, or a required concurrency may be obtained on a less complex subsystem.[6] Therefore, a dynamic architecture allows a less complex subsystem to become suitable for computing more and more complex real-time algorithms during the short periods these algorithms may need.

A dynamic architecture is assembled from building blocks called Dynamic Computer (DC) groups. Each DC group is capable of partitioning its resources into a selectable number of dynamic computers with changeable word signs.

Each DC group contains n h-bit computer elements, CE, where each CE includes an h-bit processor element, PE; an h-bit memory element, ME; and an h-bit I/O element, GE

(Figure 1). In Figure 1 the DC group includes 4CE, i.e., n = 4. DC group may form dynamic computers $C_i(k)$. Each dynamic computer handles h·k-bit words; it is assembled from k consecutive $CE(CE_i, CE_{i+1}, \ldots, CE_{i+k-1})$, and i subscript shows the position code of its most significant CE. Kartashev and Kartashev[4] showed that the most expedient h = 16 bits. Then the word sizes formed are multiples of 16 (16, 32, 48, 64, etc.), whereas the number n of CE may be 4, 8, and 16. Thus, the DC group may be conceived as a subsystem of a Supersystem.

The following major characteristics of one DC group should be mentioned here:

1. A DC group may assume a large number of different architectural states. Each state is characterized by the number and sizes of concurrent computers and arrays. Transition from one state to another is performed via software in several microseconds.
2. The same hardware resource of one DC group may assume different types of architectures: multicomputer, multiprocessors, array, and mixed. (The mixed architecture is characterized by coresidence of several types of architecture in the same system, whereby a portion of the resource functions as a multicomputer/multiprocessor, another one acts as an array, etc.)
3. A DC group provides for very fast data exchanges between any pair of resource units belonging to the same or different computers. It performs high-speed parallel word exchanges with 16·k-bit words (where k = 1, 2, . . . , n) using 16-bit size of a communication bus.

One DC group assembled from n computer elements, CE, may execute in parallel up to n concurrent programs. A Supersystem assembled from DC groups is conceived as a distributed parallel system in which each DC group is connected with the others via one of the fast interconnection busses described in the literature (crossbar, etc.).

Since, in a distributed Supersystem, the number of DC groups is in the order of tens, the complexity of the communication bus is much smaller than that of the alternative one-staged interconnection network. Therefore, the significant reduction in delays of data exchanges afforded by a one-staged interconnection network is accomplished without paying the price of excessive complexity.

On the other hand, equipping each subsystem (DC group) with dynamic architecture significantly widens a level of concurrency of a complex real-time algorithm or algorithms that can be computed in a single subsystem.

This paper discusses the organization of fast data exchanges between dynamic computers in a single DC group. This exchange is organized as follows:

If Computer B needs a data array stored in a memory page of Computer A, then this page is connected with Computer B. By loaning its page(s), Computer A does not interrupt its program, whereas Computer B begins to work with a loaned page as if it belonged to its own memory.

Loaning the memory resource for temporary use by another computer requires interference of the local operating system, which resolves conflicts arising each time two or more computers request the same memory page of another computer.

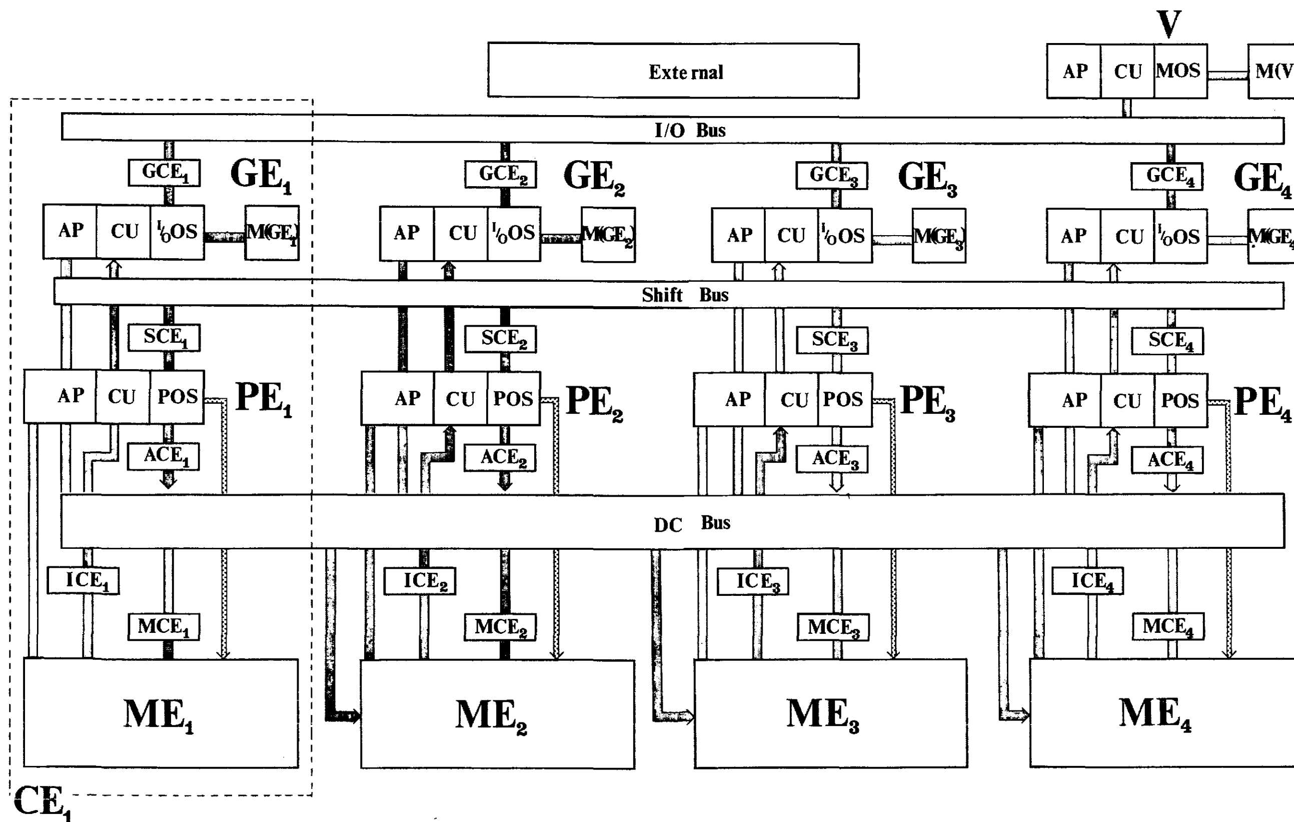


Figure 1—Block diagram of one DC group containing four computer elements

Memory conflicts are not the only type of conflicts that must be resolved by the operating system for dynamic architecture. Another type of conflict is that occurring during reconfiguration when two computers existing concurrently in a current architectural state N request transition into two different next states N′ and N″ (N′ ≠ N″).

A third type of conflict is associated with the use of I/O resources. The I/O conflicts may arise if memory-memory data exchange is performed via an I/O bus or if there are conflicts for the I/O terminals, or several I/Os make concurrent communications with the system monitor.

To solve the three types of conflicts outlined above (memory, reconfiguration, and I/O), the OS must be functionally or *vertically* distributed; i.e., it must include three subsystems:

1. The processor operating system, POS, which resolves memory conflicts
2. The I/O operating system, I/O OS, which resolves all types of I/O conflicts
3. The monitor operating system, VOS, which resolves conflicts during system reconfiguration

To be most efficient, these three OSs must reside in functionally oriented units with matching dedication; i.e., the POS must reside in the processor of a dynamic computer, the I/O OS must reside in its I/O unit, and the VOS must reside in the system monitor.

In addition to vertical or functional distribution, the operating system must feature horizontal distribution among separate CEs of the dynamic computer. Indeed, since dynamic computer, $C_i(k)$, consists of k CE and minimal k = 1, then the same POS and I/O OS must reside in every CE. Therefore, not only is the entire OS vertically distributed because of the three types of conflicts that should be resolved; it becomes horizontally distributed to resolve memory and I/O conflicts because of the modular structure of a dynamic computer.

This paper discusses the organization of the processor operating system, POS. The results presented are implemented in the POS designed for the system with dynamic architecture, which is now under construction for the Ballistic Missile Defense System Command of the U.S. Army (Contract DASG60-80-C-0058).

2. DATA EXCHANGES BETWEEN DYNAMIC COMPUTERS

Since each dynamic computer $C_i(k)$ is assembled from k CEs, to organize a parallel data exchange between two dynamic computers it is necessary to organize concurrent exchanges between respective pairs $CE_A \rightarrow CE_B$, where CE_A belongs to Computer A and CE_B belongs to Computer B. Assume that in each type of A-B exchange, the exchange is requested by B computer, whereas A loans its equipment; therefore the direction of exchange is A→B.

Further, the DC group is provided with three system busses (Figure 1): (1) a DC bus that connects all the PEs with all the MEs via separate instruction and data paths; (2) a P bus that connects all PEs; and (3) the I/O bus, which connects all I/O elements, GE. Therefore, the four types of exchanges between CE_A and CE_B which are possible are as follows (Figure 2):

1. Memory-processor exchange, $ME_A \rightarrow PE_B$, performed via DC bus
2. DC memory-memory exchange, $ME_A \rightarrow ME_B$, performed via DC bus
3. Processor-processor exchange, $PE_A \rightarrow PE_B$, performed via P bus
4. I/O memory-memory exchange, $ME_A \rightarrow ME_B$, performed via I/O bus

To increase a system's throughput, it is essential to provide maximal concurrency in all the possible data exchanges listed above. Further, since some programs require minimal time of execution without interrupts, it is necessary to discuss such organizations that do not interrupt currently executing programs.

Since there are three dedicated busses in the DC group, any two dynamic computers may perform up to three concurrent data broadcasts.

Indeed, it is possible to organize the following types of exchange concurrency in a system:

Type 1 Exchange Concurrency: (a) $ME_A \rightarrow PE_B$ (DC) via DC bus; (b) $PE_A \rightarrow PE_B$; and (c) $ME_A \rightarrow ME_B$ (I/O) via I/O bus.

Type 2 Exchange Concurrency: (a) $ME_A \rightarrow ME_B$ (DC) via DC bus; (b) $PE_A \rightarrow PE_B$; and (c) $ME_A \rightarrow ME_B$ (I/O) via I/O bus.

Since each exchange is requested by Computer B, Computer A does not interrupt its program for all exchanges but

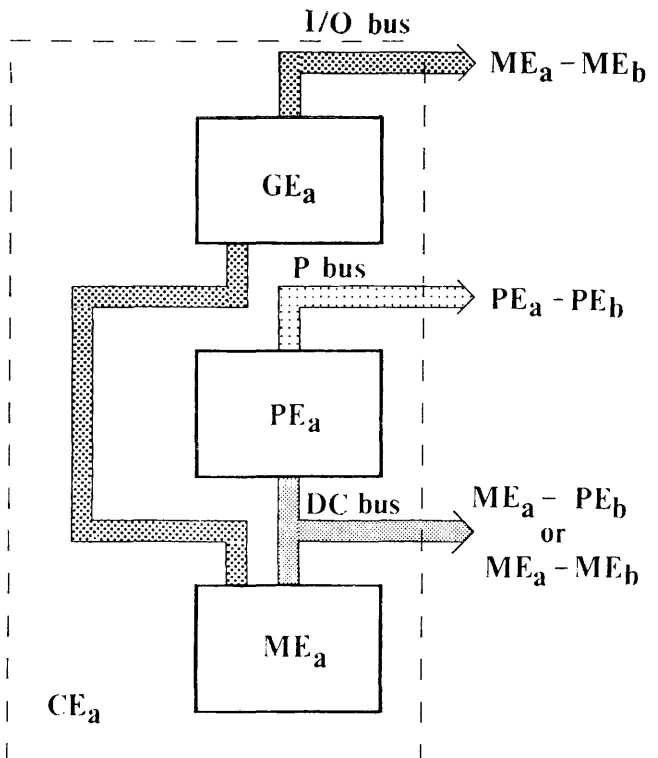


Figure 2—Possible data exchanges with one CE

$PE_A \rightarrow PE_B$. For the latter case, the A computer computes an A operand for the B computer, whereas a B operand is computed by B computer. In addition, one modification of the $ME_A \rightarrow PE_B$ exchange provides that the program computed in B computer fetch one of the operands from the memory of A computer. This fetch takes the same time as the fetch of an operand from the B memory. The result of the computation can be written to the memory of Computer A, Computer B, or both (Figure 3).

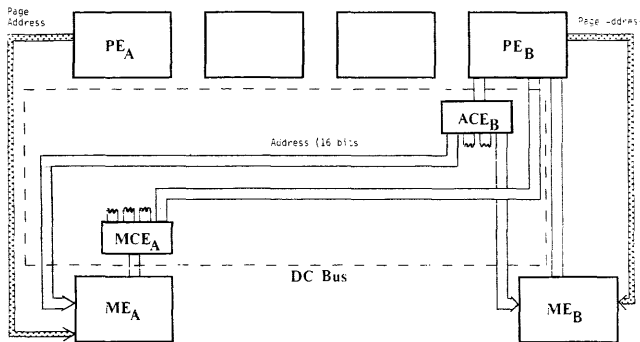


Figure 3—$ME_A \rightarrow PE_B$ exchange

Realization of concurrent data exchanges from Dynamic Computer A to Dynamic Computer B without interrupting the programs that are computed by these two computers requires implementation of the following features:

1. For a dynamic computer, every memory element, ME, of its primary memory must be multiport and reconfigurable; i.e., ME should be provided with four information ports and be capable of connecting all its pages to the four ports (Figure 4). These ports are as follows:
   a. A local data port that provides fetch of the local data word for the program computed by the dynamic computer A.
   b. An instruction port that fetches instructions to all PEs of the dynamic computer. Since instructions may be stored in any ME of a dynamic computer, every ME must have a separate instruction port.
   c. A DC port that fetches a data word stored in this ME to another dynamic computer.
   d. An I/O port that transfers a data word stored in this ME to another dynamic computer, using an I/O bus.
2. Every computer element, CE_A, of the dynamic computer A must be provided with the two operating systems: POS and I/O OS. Further, to speed up conflict resolution, it is essential to implement these two OSs via hardware. The POS will resolve memory conflicts for the pages of this ME_A, and I/O OS will find what dynamic computer B will participate in the memory-memory exchange $ME_A \rightarrow ME_B$ with ME_A via the I/O bus.

2.1 Multiport and Reconfigurable Memory ME

As was indicated above, each CE is equipped with the reconfigurable multiport ME that can access up to four 16-bit

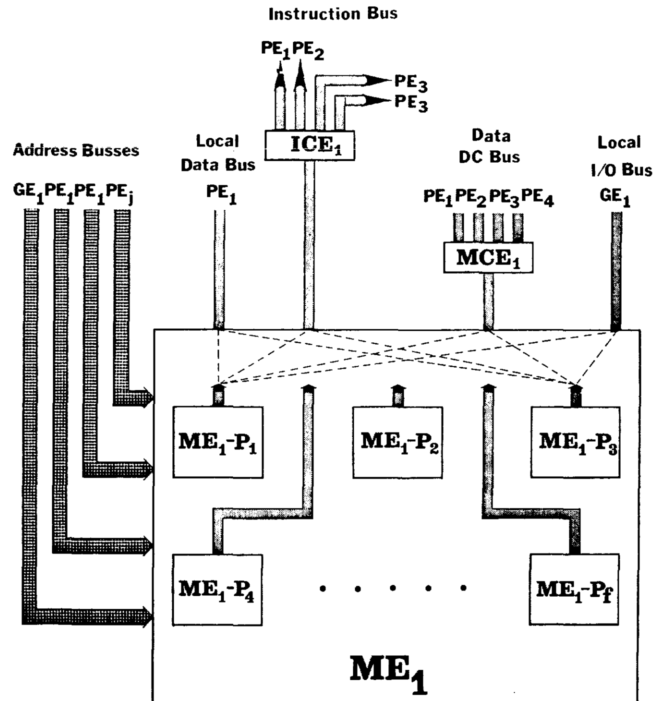


Figure 4—Multiport reconfigurable ME with four information busses

data words concurrently. One ME has f pages: $ME - P_1$, $ME - P_2$, ... , $ME - P_f$ (Figure 4). In the system, DCA-2, that is now under implementation, $f = 32$; i.e., each ME has 32 pages, each of which has 64K words. The memory, ME, is provided with four ports, each of which consists of a 24-bit address bus and a 16-bit data bus. There are four data busses.

Let us discuss the designation of each of them.

1. The local data bus provides 16-bit word exchange between the local ME and PE. A dynamic computer handles 16·k-bit words in parallel. Since a 16·k-bit word is stored in a parallel cell of k ME specified with the same address, A_p, an access to this cell (read or write) is specified by concurrent broadcast of the same address, A_p, by k PEs of the dynamic computer. This allows a concurrent fetch of k 16-bit bytes of the same data words via the respective local data buses to k PEs of the dynamic computer.
2. The instruction bus provides fetch of 16-bit words of one instruction that is stored in one ME. One instruction may be two-word (32-bit instruction) or three-word (48-bit instruction). Further, it must be fetched to all k PEs of the dynamic computer. The instruction broadcast to k PEs of the computer is performed via a connecting element, ICE.
3. The DC bus provides broadcast of a 16-bit word from the given ME to any PE or ME of the DC group. For ME_A, belonging to Computer A, any of its pages may be connected with any PE_B or ME_B belonging to Computer B (Figure 5).

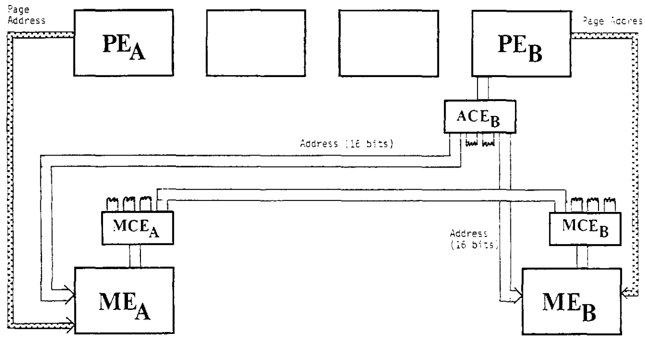


Figure 5—$ME_A \rightarrow ME_B$ exchange via DC bus

4. The local I/O bus provides broadcast of 16-bit data between local ME and I/O (GE), belonging to the same CE.

Therefore, multiport reconfigurable memory, ME, provides for concurrent connection of any combination of its four pages with the four busses mentioned above. It may be fed with up to four addresses leading to concurrent accesses of up to four data words transferred to local PE and GE and nonlocal transfer to PE_B or ME_B of another computer element CE_B.

As was mentioned above, each address bus is a 24-bit. It is formed of two parts: an 8-bit page address and a 16-bit relative address within one page.

Each ME_A receives its addresses from the following sources. All four 8-bit page addresses are received from the local POS_A belonging to the local PE_A; 16-bit relative addresses for local data words and instructions that must be fetched to local PE_A are broadcast from the local control unit CU(PE) of this PE_A. A 16-bit data word that must be broadcast to a nonlocal PE_B of another computer is defined via a 16-bit relative address broadcast from PE_B.

Local GE_A broadcasts a 16-bit address for the data to be transferred via the local I/O bus (Figure 6). The same 8-bit

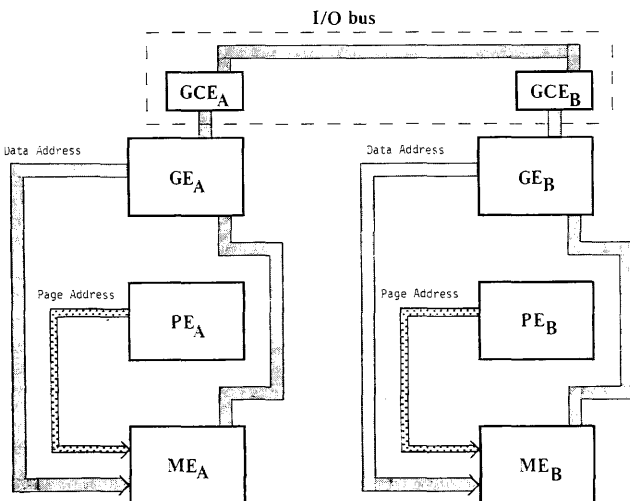


Figure 6—$ME_A \rightarrow ME_B$ exchange via I/O bus

page addresses are fed continuously during program computation, whereas 16-bit relative addresses are available only during fetch clock period. A change in an 8-bit page address is performed either via special instruction or via POS_A.

2.2 *Memory-Processor Data Exchange $ME_A \rightarrow PE_B$*

Let us discuss the memory-processor data exchange of the data word stored in ME_A and fetched to PE_B. Suppose that a PE_B of the B computer needs a data array stored in the page ME_A-P_ℓ of the ME_A contained in the A computer. In this case Computer A connects page ME_A-P_ℓ with the DC bus. The 8-bit page address, P_l, is generated by the operating system POS_A in PE_A of the A computer and continuously fed to ME_A during the entire exchange (Figure 3).

The B computer sends a 16-bit relative address to the ME_A via its connecting element ACE_B and the address portion of the DC bus. A data word fetched via an effective 24-bit address is sent via connecting element MCE_A to the data portion of the DC bus that connects MCE_A with PE_B.

It should be noted that the delay introduced by the DC bus in transferring addresses and data words is very insignificant and equivalent to two gates delay. In addition, these delays are permanent and independent of the location of ME_A and PE_B in the resource. This allows organization of new types of instructions. Each such instruction executed in Computer B may fetch one operand from ME_A and the second operand from ME_B and write the result either to ME_B or ME_A. This instruction is organized as follows: Computer B sends concurrently two 16-bit addresses; one via DC (address) bus is fed to ME_A, and another via local (address) bus is fed to ME_B. Thus, PE_B receives two operands concurrently: one fetched via DC (data) bus, another via local (data) bus. The page address for ME_B is generated by PE_B. The result of the operation can be written either to ME_A or to ME_B.

Note that the fact that Computer A loans its page, ME_A-P_ℓ, to Computer B does not prevent A from executing its program because a loaned page, ME_A-P_l, is connected with the DC bus, whereas program instructions computed in Computer A are stored in another page connected with the instruction bus. The data words for this program are stored in another page connected with the local data bus. Sequencing of instruction and data arrays stored in several pages is performed with special instructions that change the page addresses connected with instruction and local data busses.

Therefore, the ability of one dynamic computer to loan its memory pages for use by the B computer eliminates the necessity of using ME_A-ME_B data exchange. If there are no conflicts for the page, ME_A-P_ℓ of the A computer, then to transfer a data array made of d words from ME_A to PE_B takes $(d + t)$ clock periods, where t is a small number of clock periods required to generate the page address by POS_A; thereafter each word may be transferred during one clock period.

Another advantage of such organization is as follows: Since the DC bus is connected with all computer elements, it may be used by the B computer for fetching one of its operands from the local ME_B; the second operand may be fetched concurrently from another page of the same ME_B via a local data bus. This results in a concurrent fetch of two operands that leads to a significant speedup in data fetches.

2.3 Memory-Memory Exchange, $ME_A \rightarrow ME_B$

As was indicated above, there are two types of memory-memory exchange between A and B computers, organized via a DC bus and an I/O bus, respectively (Figures 5 and 6). Both types of exchanges do not interrupt programs run on A and B computers. The most typical use of both exchanges occurs when the data array stored in A computer is transferred to B computer before the program run in B computer actually needs this array.

2.3.1 $ME_A \rightarrow ME_B$ exchange via DC bus

Since the DC bus is connected with each element, MCE, the $MCE_A \rightarrow MCE_B$ connection will establish a data path for data words fetched from ME_A and to be written to ME_B (Figure 5). Since the program run on B computer requests this exchange, B computer generates 16-bit addresses for ME_A and ME_B. In addition, B computer generates the page address (8 bits) for ME_B. This address connects the respective page in ME_B to the DC bus. The page address for ME_A is generated by Computer A.

Two 16-bit addresses that define respectively the source and destination of a data word in ME_A and ME_B are generated concurrently, and one data word is transferred in one clock period from ME_A to ME_B. The same addresses are fed continuously to ME_A and ME_B during the entire broadcast of the data array from the same page.

2.3.2 $ME_A \rightarrow ME_B$ exchange via I/O bus

Since the DC bus is often occupied by $ME_A \rightarrow PE_B$ exchanges, and the program on the B computer may often need data words computed by A computer in the past, it is desirable to organize another type of $ME_A \rightarrow ME_B$ exchange via I/O bus.

Since each ME may connect its pages to the local I/O bus, one can organize $ME_A \rightarrow ME_B$ exchange via I/O elements GE of A and B computers (Figure 6). The page addresses in ME_A and ME_B are generated by PE_A and PE_B respectively. Therefore, the following data path is established: During the first clock period, $T_0^{(1)}$, a data word is fetched from ME_A to GE_A; during $T_0^{(2)}$ it is transferred via I/O bus to GE_B; during $T_0^{(3)}$ it is written to ME_B.

This transfer is overlapped, so that each clock period features one fetch of a new data word from ME_A and the entire transfer of a data array having d words from ME_A to ME_B takes $d+2$ clock periods.

This concludes the description of all data exchanges that involve the operating system.

It should be noted that for $PE_A \rightarrow PE_B$ exchange, the OS is not involved, since this exchange provides for concurrent reception of the same operand by all PEs, which is useful for array architecture. Thus, it will not be considered in this paper.

3. DISTRIBUTED OPERATING SYSTEM

If the same page of the A computer is requested by two or more other computers, the POS_A residing in PE_A has to decide which computer request for $ME_A \rightarrow PE_B$ or $ME_A \rightarrow ME_B$ exchanges should be granted. Similar conflict resolution must be provided by the I/O OS_A to decide which computer may use $ME_A \rightarrow ME_B$ (I/O) exchange via an I/O bus. To solve these conflicts, each program is assigned the priority code, PC, that shows the relative importance of this program among all others that are being computed by the system. Also, each request for a page is provided with another important characteristic—the tentative duration of a data exchange, TDE.

These two codes will provide the user with a much better quality of service, since the programs with low PCs and small TDE may be granted requests because the requested exchange will take a short time. Should a page request be characterized by the PC code alone, it would be impossible for a program of low priority or priorities to request data exchanges of short durations.

Thus, if a page of Computer A is requested, either POS_A or I/O OS_A receive two characteristics of each page request: the priority code, PC, of the requesting program and the tentative duration, TDE, of the exchange. The POS_A receives PC and TDE, if a requested data exchange will use the DC bus; the I/O OS_A receives PC and TDE if the exchange will use the I/O bus.

POS_A controls all four busses of ME_A; i.e., it alone connects memory pages to the busses. Therefore, if I/O OS_A has a request on $ME_A \rightarrow ME_B$ (I/O) exchange, it requests the POS_A on the possibility of connecting a requested page to the local I/O bus.

Other functions of I/O OS_A include finding the terminal that should be connected with GE_A, communicating with the V monitor, and similar functions. These functions of I/O OS_A will not be considered in this paper.

3.1 Communication Between Different POSs

Since a dynamic computer, A, includes k CEs, it has k POSs. Each POS_A makes decisions concerning the pages of the local ME_A only. As will be shown below, such horizontal distribution of functions allows parallel data exchanges both with full 16·k-bit data words and with 16·f-bit bytes, where $1 \leq f < k$. Indeed, if two computers, A and B, are assembled from the same number k of CEs, then parallel 16·k-bit data exchanges mean concurrent communication by each CE of the A computer with the respective CE of the B computer. In Figure 7 this communication is shown for 32-bit computers A and B, assembled from CE_1 and CE_2 for A and CE_3 and CE_4

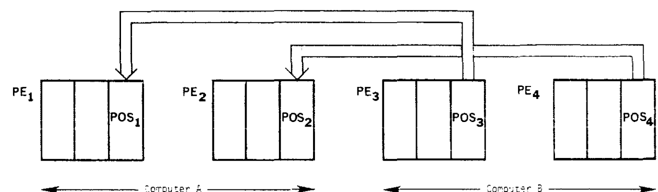


Figure 7—Concurrent communications between different POSs of two dynamic computers

for B. Thus POS_4 of PE_4 communicates with POS_2 of PE_2, and POS_3 of PE_3 communicates with POS_1 of PE_1.

This means that each CE_A of the A computer must generate the same page address in order that this page be connected with the DC bus in the local ME_A. This will lead to a concurrent fetch of a 16·k-bit word from the same page of k ME_A, so that each ME_A will produce one byte of this word. The decision on what page should be connected with the DC bus is performed by the local POS_A that receives the request from the respective POS_B of the B computer. Thus, if A and B computers have the same number of CEs, then k communicating pairs POS_A,POS_B are formed, where each POS_B sends the request to the respective POS_A. This request is organized with a special communication request instruction, CR instruction, whose organization is described in the next section.

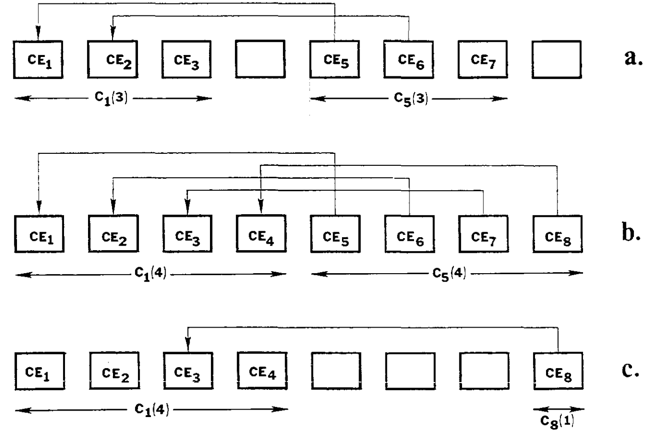


Figure 8—Establishment of POSs communication pairs for parallel 16·f-bit exchanges where f = 1, . . . , k

3.2 *Organization of Communication Request Instruction*

The CR instruction is fetched by all CEs of the B computer. It stores the following codes: w_B, y_A, and x_B, where

1. w_B is the position code of the most significant CE (CE_w) of the B computer that requests communication with its analog of the A computer.
2. y_A is the position code of the most significant CE (CE_y) of the A computer that receives requests from CE_w.
3. x_B is the position code of the least significant CE (CE_x) of the B computer that participates in communication.

Let i_B be a current position code of a CE_B of Computer B, where $w_B \le i_B \le x_B$. Then position code j_A of CE_A of Computer A, which communicates with this CE_B, is given as follows:

$$j_A = i_B + (y_A - w_B) \tag{1}$$

The execution of this instruction is organized as follows: Having received a communication request instruction, each CE_B of the B computer compares its position code i_B with w_B and x_B. If $w_B \le i_B \le x_B$, CE_B finds j_A of the CE_A via Equation 1. Otherwise, the CR instruction is not executed. It should be noted that this instruction can organize parallel byte exchanges as well as full data word exchanges. For byte exchanges w_B and x_B show positions of most and least significant bytes for data words to be exchanged. For full data word exchanges, there is no need to store x_B, since the instruction is received by all CEs of B computer and every CE_B of B computer will thus find the position code, j_A, of its communication pair in the A computer.

Example 1. First, consider parallel exchange with 16·f-bit bytes. Let a dynamic computer $B = C_5(3)$ assembled from CE_5, CE_6, and CE_7 require that CE_5 send a request to CE_1 and CE_6 send a request to CE_2, where CE_1 and CE_2 belong to Computer $A = C_1(3)$ that contains CE_1, CE_2, and CE_3 (Figure 8). The CR instruction stores the following codes: $w_B = 5$, since CE_5 is the most significant slice of B computer that needs exchange; $y_A = 1$, since CE_1 is the most significant slice of A computer; $x_B = 6$, since CE_6 is the least significant slice of the B computer that needs exchange.

Each CE_B of the B computer compares its position code, i_B, with w_B and x_B; for CE_5, $i_B = 5$; therefore, $w_B \le i_B \le x_B$ is true ($5 \le 5 \le 6$). For CE_6, $i_B = 6$, $w_B \le i_B \le x_B$ is also true ($5 \le 6 \le 6$). For CE_7, $i_B = 7$, therefore, $w_B \le i_B \le x_B$ is false ($5 \not\le 7 \not\le 6$). Therefore, $j_A = 5 + (1 - 5) = 5 - 4 = 1$; i.e., CE_5 sends a request to CE_1. For CE_6, $i_B = 6$. Therefore, $J_A = 6 + (1 - 5) = 2$; i.e., CE_6 sends a request to CE_2. CE_7 sends no communication request, since it did not pass via the conditional test.

For full data word exchanges, the CR instruction stores only w_B and y_A codes; i.e., Field x_B is empty and each CE_B of the B computer executes Equation 1.

In Figure 8(b) let the B computer, $C_5(4)$, assembled from CE_5, CE_6, CE_7, and CE_8, need 64-bit data words stored in a computer, $C_1(4)$, assembled from CE_1, CE_2, CE_3, and CE_4. The CR instruction stores $w_B = 5$, $y_A = 1$, and $x_B = 0$. In CE_5 the following j_A is obtained: $j_A = 5 + 1 - 5 = 1$; i.e., CE_5 communicates with CE_1. For CE_6, $j_A = 6 + (1 - 5) = 2$; i.e., CE_6 communicates with CE_2. For CE_7, $j_A = 7 + (1 - 5) = 3$; i.e., CE_7 communicates with CE_3. Similarly, CE_8 communicates with CE_4.

The power of the CR instruction is such that it can organize data exchanges between different-size computers, A and B, when the size of B may be either smaller or larger than the size of Computer A. If the size of B is smaller than that of A, then B may receive 16·f-bit bytes from Computer A that match the size of B. If the size of B is larger than that of A, only a portion of the CEs in the B computer will establish communication requests with their pairs from A computer.

Computer A, assembled from f CE, will send full 16·f-bit data words, which will be received only by f slices of Computer B assembled from k CE, where $f < k$. The data exchange between different-size computers is exemplified by Figure 8(c), in which 16-bit Computer $B = C_8(1)$, assembled from CE_8, requests an array of 16-bit bytes stored in CE_3; CE_3 belongs to the A computer, $C_1(4)$, assembled from CE_1 through CE_4.

The B computer fetches the CR instruction that stores $w_B = 8$, since CE_8 is the most significant in $C_8(1)$; $y_A = 3$, because CE_3 is the most significant CE in Computer A that receives communication requests, and $x_B = 0$. Thus $j_A = 8 + (3 - 8) = 3$, and CE_8 communicates with CE_3.

4. PROCESSOR OPERATING SYSTEM, POS

Every POS_A of the dynamic computer A may perform the following functions:

1. *Generation of the page addresses for all four busses of the local ME_A.* Local POS_A may generate concurrently up to four 8-bit page addresses that specify what pages of the local ME_A will be connected with four data busses that are available (local data, bus, instruction bus, DC data bus, and local I/O bus).
2. *Handling requests on the local memory resources.* The local POS_A receives and handles requests concerning the pages of the local ME_A from any other POS_B. Or POS_A may receive a request from the local I/O OS_A contained in the same CE_A. Having received each type of request (either from POS_B or from I/O OS_A), POS_A finds whether or not it is possible to connect a requested memory page(s) to the DC bus and/or the local I/O bus. In both cases the page requester is informed of the decision made.
3. *Handling denied requests.* If POS_A cannot give a requested memory page to its requester (either POS_B or I/O OS_A), each denied request is stored. When a requested page becomes free, a requester is informed of this occasion.
4. *Generation of page requests for the nonlocal memory resource.* If a local program needs a nonlocal memory page of ME_C, then the local POS_A sends a page request to another POS_C local with ME_C.
5. *Handling program interrupts.* If a requested memory page is not received, so that a local program cannot perform further computations, the local POS_A handles program interrupt(s) of the program computed by CE_A. To this end it evacuates all the related data words stored in data registers of PE_A to the local memory ME_A and initiates computation of another program that waits in queue.
6. *Resumption of interrupted program(s).* If the local POS_A is belatedly granted a page request for a program that was interrupted because this request was not satisfied in time, the POS_A resumes computation of the interrupted program, provided its priority is higher than those of all other interrupted programs waiting for computation. To this end, the POS_A completely restores a computation status of an interrupted program before the last interrupt.

Let us now give organizations on each of the functions introduced above except Functions V and VI, which have been considered in Kartashev and Kartashev.[4]

4.1. Generation of Page Addresses

In each CE a local computing program may use two busses: the local data bus, which receives data words from the local ME; and the instruction bus, which broadcasts instructions fetched from the local ME to all CEs of the dynamic computer, provided a current program segment is stored in this ME (Figure 4).

For convenience of programming, it is provided that the data words needed by a current program may be stored not in one but in four pages. The respective four-page addresses are stored in four 8-bit registers, R1 through R4 (Figure 9). Each data fetch instruction that organizes an operand fetch from the local ME stores a two-bit code m that specifies which of the registers R1 through R4 stores a current page address. This register is then connected with the 8-bit local page address for the local data bus. In Figure 9 the following m's are used: If m = 00, R1 stores a current page address; if m = 01, it is stored in R2; etc.

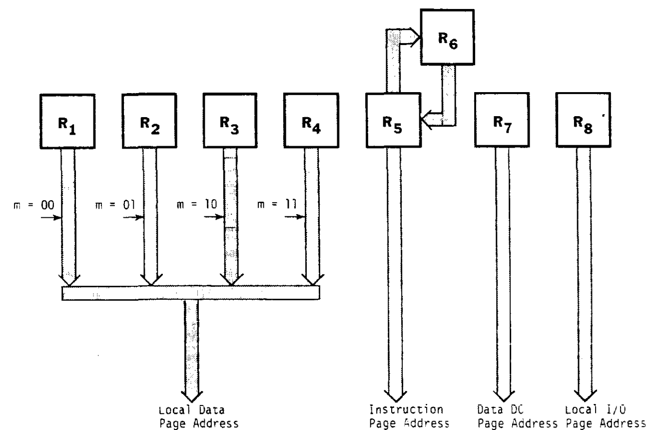


Figure 9—Page address registers

Similarly, it is provided that the instructions for a currently executed program segment be stored in two pages whose addresses are stored in R5 and R6, where R5 stores the page address that is connected to the instruction bus and R6 stores the next page address. If a program needs to jump to a page address stored in R6, the following transfers are performed: R5→R6 and R6→R5; i.e., a current address is saved in R6 and a new address is transferred to R5. This is done with a special instruction that transfers control to a new program segment whose page address was in R6.

For the DC bus and the local I/O bus, the respective page addresses are stored in R7 and R8; i.e., a programmer may work with one page for each of these busses. All these registers (R1 through R8) are included in the Page Set Registers of the POS (Figure 10).

4.2. Handling Page Requests

All POSs of the DC group may exchange with 16-bit messages. Each message may belong to one of the following categories: (1) it may be a *page request,* or (2) it may be a *Yes* or *No response* on a page request. There are other types of messages that are not discussed in this paper.

A page request to a given POS_A from all the POS_B's is broadcast via connecting element SCE_A local with CE_A. The DC group has n SCEs that are forming the P-bus considered above. In Figure 10, n = 4; therefore the SCE has four 16-bit channels, of which three are input channels for receiving page

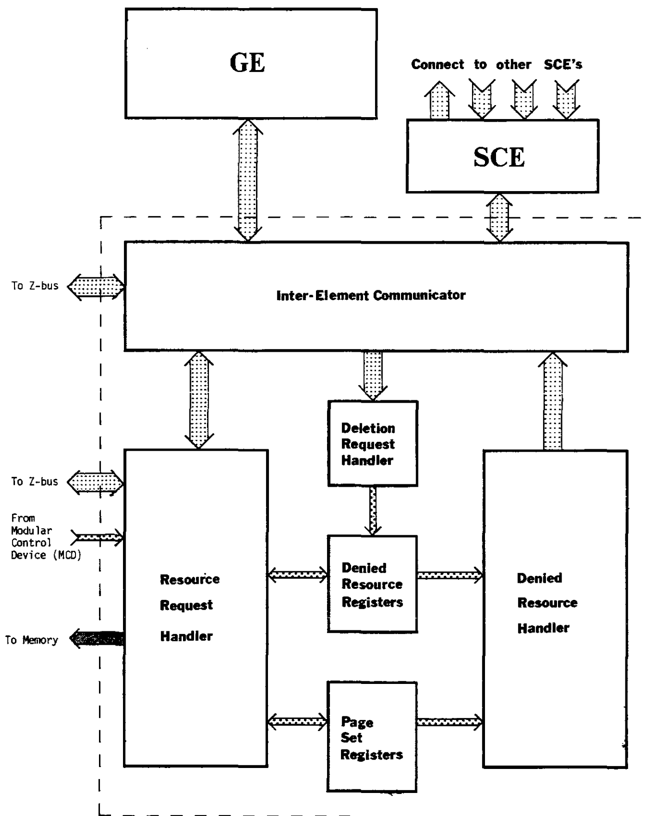


Figure 10—Block diagram of the processor operating system

requests from other CEs and one is the output channel for sending its own page request to other CEs. Further, as was indicated above, every POS_A may receive a 16-bit request from the local I/O OS_A via a 16-bit data bus that connects PE_A with the local GE_A.

Since each POS_A may receive up to $n - 1$ concurrent page requests from other POSs, these requests form a queue in the *Inter-element Communicator* that allows only one request to be handled at a time. After this request is finished, the next request in a queue is processed, etc. Having received a page request, each POS_A initiates a *Resource Request Handler* that executes the actions discussed in the next sections.

4.2.1. Resource Request Handler

A received page address is compared concurrently with all those stored in R1 through R8 to find out whether a requested page, RP, is busy.

If it is free, the POS_A performs the following functions:

1. Generates a "Yes" response to the external requester, POS_B.
2. Writes this page address to the R7 register that connects it to the DC bus.
3. Activates the data channel in the connecting element, MCE_A, that connects the local memory element, ME_A, with its destination processor element, PE_B, that requested this page (Figure 5). This establishes the $ME_A \rightarrow PE_B$ data path whereby a local page in ME_A was loaned to Computer B.

If a requested page, RP, is busy, its address is stored in one of the registers, R1 through R8. This means that this RP-page is used by one of the following programs (Figure 9):

1. If it is stored in the R1–R6 registers, it is used by the local program, AP, computed by Computer A.
2. If it is stored in the R7 register, it is used by another computer, C, which borrowed this page via the DC bus. Thus this page is used by the CP program.
3. If it is stored in the R8 register, it is used by the local program, GEP, computed in the local I/O element, GE_A.

To generate a Yes or No response to a page request, the operating system, POS_A, must compare the received PC_B and TDE_B parameters (where PC_B is the priority code and TDE_B is the tentative duration of the data exchange of the requester B) with those of the program that is currently using the requested page, RP. It is either AP, CP, or GEP. The priority code and tentative duration of each of these programs is stored in the Resource Request Handler.

4.2.2 Estimate Table

To find out which of the programs must use the requested memory page, RP, the POS_A performs a priority analysis that consists of the following: It analyzes a special *estimate table* (Table I) whose rows are marked by priority codes, PC, and columns are marked by TDE times.

TABLE I—Estimate table

Program priority	Tentative duration (TDE)				
	10^2	10^3	10^4	10^5	10^6
I	20	15	10	5	1
II	25	20	15	10	5
III	30	25	20	15	10
IV	35	30	25	20	15
V	40	35	30	25	20
VI	45	40	35	30	25
VII	50	45	40	35	30

In the system under implementation, there are seven levels of priorities; thus PC is a three-bit code and $PC_7 > PC_6 > \ldots > PC_1$.

Assume that the TDE code ranges from 10^2 clock periods to 10^6 clock periods. Thus the estimate table will have seven rows and five columns.

The intersection (PC, TDE) of the given PC row and the TDE column gives an integer called *program weight, PW,* that shows what actions should be taken by the POS_A. For instance, in Table I, if $TDE = 10^2$ and $PC = I$, then $PW = 20$. If $TDE = 10^4$ and $PC = IV$, $PW = 25$, etc.

It should be noted that the estimate table is made up on the

basis of statistical analysis of algorithms that are computed by the DC group. Each estimate table can be augmented with new columns and new rows to reflect given computational requirements. Further, since the estimate table is stored in the memory, its expansion requires no hardware changes. Row and column entries may also be changed to reflect a change in a set of programs that are under execution.

4.2.3 Decisions made by the POS_A

To make a decision concerning a requested page, RP, the POS_A must find two program weights, PW_U and PW_B, where PW_U is the program weight of the program that is currently using a requested page and PW_B is the program weight of the program that is requesting this page. If $PW_U \geq PW_B$, a requester, POS_B, receives a No response. If $PW_U < PE_B$, a requester, POS_B, receives a Yes response.

Handling the No response is considered in Section 4.3. As for the Yes response, the type of actions undertaken by the POS_A—following comparison of PW_U and PW_B—depends on what type of programs have been using a requested page, RP. As follows from the material above, there are four cases of this usage:

A requested page, RP, has been used by one of the following:

- Case (a): Local AP program using a local data bus
- Case (b): Local AP program using an instruction bus
- Case (c): External CP program using a DC bus
- Case (d): Local GEP program using a local I/O bus

Case (a): Local AP program using a local data bus. If a requested page, RP, is used by the AP program, for data fetches, its address is stored in one of the registers, R1–R4 (Figure 9). If this page is granted to the B computer, its address should be written to the R7 register connected with the DC bus, whereas the register that stored it before should be reset. However, the local AP program may continue its execution until it starts data fetches from the requested page. In this case the AP program will become interrupted only when it fetches a data fetch instruction with 2-bit code m that connects a requested page granted to B computer with the data page address. Such an organization allows elimination of unnecessary interrupts. Thus the AP becomes interrupted only when such interrupt is absolutely necessary.

Case (b): Local AP program using an instruction bus. If a requested page, RP, is used by the AP program for instruction fetches, its address is stored either in R5 or in R6 registers. If the request on this page to POS_B is granted and a requested page is stored in the R5 register, R5 is reset and the AP program is interrupted. If a requested page, RP, is stored in the R6 register, R6 is reset and the execution of Program AP proceeds until it fetches the instruction that transfers control from R6 to R5.

Case (c): External CP program using a DC bus. If a requested page, RP, is used by the external CP program, its address is stored in Register R7. If the POS_A decides to give the requested page to the POS_B, then the following actions are performed: (1) The Resource Request Handler of the POS_A sends a message to POS_C that a page in ME_A will be denied for further use by CP program; (2) having received this message, POS_C acknowledges its reception; (3) POS_A establishes a new path in the DC bus by connecting ME_A with PE_B; and (4) POS_A sends a Yes response message to POS_B, indicating that its page request is granted.

Case (d): Local GEP program using a local I/O bus. Actions similar to those in Case (c) are performed if a requested page has been used by the local GEP program.

4.3 Denied Resource Handler

If the POS_A denies a page request made by the POS_B, this request is remembered in the *denied resource table* (Table II) that is stored in the Denied Resource Handler unit (Figure 10).

TABLE II—Denied resource table

CE_1	5, VI, 3	6, III, 5	7, V, 10	22, II, 10
CE_2	6, III, 5	5, IV, 2	—	—
CE_3	—	—	—	—
CE_4	16, VI, 5	—	—	—
CE_3	5, IV, 4	—	—	—

4.3.1 Denied resource table

This table has n + 1 rows, where n rows are assigned for n CEs of the system and the last row is assigned for the local I/O element, GE_A.

The table contains d columns, where d specifies the number of denied requests from one POS_B that can be stored. Each row marked by CE_B stores denied page requests made by the POS_B residing in CE_B. Since local POS_A does not make page requests to itself, the row CE_A is empty. This is accomplished to preserve the circuit identity of all POSs. The last row, GE_A, stores denied page requests made by local I/O OS_A. For instance, if the DC group has four CE (n = 4) and it is selected that d = 4, then Table II has five rows and four columns. The entry (i, j) of the ith row and the jth columns stores the following *page request parameters:* the address RPA of requested page, RP; the priority code PC of the program that requests RP; and the tentative duration of an exchange, TDE.

Example 2. For the DC group with four CEs (n = 4), consider Table II stored in POS_3 local with CEs. This means that Row CE_3 is empty; in the first four rows this table will store all page requests on the pages of ME_3 made by POS_1, POS_2, and POS_4. In the last (fifth) row, the requests made by local I/O OS_3 will be stored. The row CE_1 stores all page requests made by POS_1. There are four such requests. Request 1 is on Page 5, for the program with PC = VI. The page is needed during 10^3 clock periods. Request 2 is on Page 6 for the program with PC = III during 10^5 clock periods; etc. The local GE_3 row stores only one page request on Page 5 for the program with PC = IV during 10^4 clock periods.

4.3.2 Handling denied requests

For each POS_A Table II stores two types of denied requests: external and internal. External denied requests are from all other POSs. Internal requests are from the local I/O OS_A.

To satisfy an external request requires that the DC bus be free and the requested page, RP, be free. To satisfy an internal page request requires that only the requested page be free. Consider handling external denied requests only, since handling internal denied requests is a simple extension of this more general procedure. Each time the DC bus is free and one page of ME_A is released—i.e., its address called *released page address, RPA,* is taken away from one of the registers, R1–R8—the RPA address is sent to the Denied Resource Handler.

Thereafter, RPA is compared with all page addresses stored in Table II.

If it is not stored in Table II, the next released page is analyzed.

If it is stored in Table II, assume that it is stored in Row B and Column 1—i.e., it is requested by POS_B for the first time. Upon fetching this request, the POS_A informs the POS_B that the requested page can be connected to the computer element CE_B. If the POS_B agrees to accept this page, the POS_A writes this RPA address to register R8—which is connected with the DC bus—and deletes this request from Table II. If POS_B does not agree to accept this page, then again this page request is deleted from Table II. If the same released page address, RPA, is stored in several entries of Table II, this means that the same page is requested by several programs. All such requests having the same RPA are fetched; and, using Table I, their program weights, PW, are found. Thereafter the request with the highest PW is satisfied and deleted from Table II. The remaining requests continue to be stored in Table II.

Example 3. Suppose that the DC bus is free and ME_3 releases Page 5 (i.e., RPA = 5). In Table II, the same RPA = 5 is stored in three requests: Request 1, made by POS_1; request 2, made by POS_2; and request 1, made by the local I/O OS_3. Using Table I, we find that the program weights of these requests are as follows:

Request #1 (POS_1)	$PW_1 = 40$ (row VI, col. 10^3)
Request #2 (POS_2)	$PW_2 = 35$ (row IV, col. 10^2)
Request #1 (I/O OS_3)	$PW_3 = 25$ (row IV, col. 10^4)

Since PW_1 is the highest program weight, Request 1 (POS_1) is granted. This means that POS_3 informs POS_1 that Page 5 can be connected with CE_1.

If POS_1 agrees to accept this page, POS_3 writes RPA = 5 to register R8 and deletes this request from Table II (Figure 9).

If POS_1 informs POS_3 that it does not currently need Page 5, Request 1 (POS_1) is deleted from Table II.

4.4 Generation of Page Requests

Consider the organization of page requests on a nonlocal memory resource. Let the program B computed in CE_B request a page, RP, belonging to ME_A. This page request is organized with the communication request instruction (CR instruction) introduced in Section 3.2. Whereas Section 3.2 discussed how the CR instruction finds the destination, POS_A, which controls all accesses to the requested page, RP, this section will discuss other actions of the CR instruction on generating a page request.

In all, the following information is stored in the CR instruction: requested page address, RPA; position codes w_B, y_A, and x_B, which allow finding destination position code j_A of the POS_A via Equation 1; and tentative duration of the exchange, TDE.

When fetched to the control unit, the CR instruction is transferred to the local POS_B, which begins its execution independent of Program B. Such organization leads to concurrency in establishing a needed data exchange with execution of the main program. This allows setting up a needed exchange before this exchange is needed in computation. This is achieved as follows: The program B will have two identical CR instructions. The first one, the CR_1 instruction, is stored somewhere in a program segment that goes far ahead of the instructions handling the data array stored in the requested page, RP. The second one, the CR_2 instruction, is immediately followed by the first instruction that handles data words from the requested page, RP.

If the CR_1 instruction receives the requested page, RP, the CR_2 instruction is ignored. For this case, the time of establishing a data exchange from Computer A to Computer B will be reduced to 0, since Computer B will receive the requested page, RP, before this page is needed in computation. If the CR_1 instruction is denied the requested page, RP, Program B continues execution until it reaches the CR_2 instruction.

If the CR_2 instruction is denied the requested page, RP, Program B is interrupted until the requested page, RP, becomes released. During this interrupt, Computer B may begin other executions.

Therefore, by allowing executional concurrency in computing CR instructions (CR_1 and CR_2) with the main program B, one can obtain a complete overlap in data exchange with execution of the main program, provided its program weight is high. Indeed, in this case, by assigning a high PW to the main program, it is possible for the requested page to be received even by the CR_1 instruction, i.e., long before it is actually needed in computations.

When the CR instruction is received by the local POS_B, it forms a 16-bit page request message that stores: (1) the requested page address, RPA; and TDE time taken from the CR instruction; and (2) the priority code PC, stored in a special priority register of POS_B. This request is formed in the Resource Request Handler (Figure 10). Thereafter it is transferred to the Inter-element Communicator and local connecting element SCE_B. Selection of the bus that connects SCE_B with SCE_A is made with Position Codes w_B, y_A, and x_B, discussed in Section 3.2.

This section concludes the introduction of the major organizations for a distributed operating system in a system with dynamic architecture.

CONCLUSIONS

This paper has introduced major concepts for the distributed operating system of dynamic architecture. This system is now

under implementation (Federal Contract DASG60-80-C-0058) for a system with dynamic architecture for ballistic missile defense applications.

To be most effective, the operating system incorporates two types of distribution:

1. Functional or vertical distribution whereby it is distributed, in accordance with three major functions it must perform (resolution of reconfiguration conflicts, I/O conflicts, and page conflicts)
2. Modular or horizontal distribution when the same operating system (POS and I/O OS) is distributed among different CEs of a dynamic computer as a result of the modularity concept implemented in this computer

It is shown in the paper that such duality in distributed functions leads to extreme effectiveness in the organization by the operating system of various data exchanges between various dynamic computers that are formed from the resources. Indeed, a portion of the memory resource of one computer can be very easily attached to that of the second computer. This computer now has the loaned memory units, with the needed data arrays, incorporated into its own primary memory. This eliminates most of the delays caused by data transfers from one memory to another that must be spent in conventional systems for organizing data exchanges between different computers.

Other advantages of dynamic architectures for very fast real-time applications are not discussed in this paper, since they were extensively treated by Kartashev and Kartashev, Vick and Kartashevs, and Baer.[3-6]

Summarizing what has just been said, one can state that a system with dynamic architecture provides a significantly higher throughput than a conventional system, provided that both types of systems exhibit the same complexity of resources (the number of processor and memory elements and the complexity of the interconnection network) and are built from the same types of components.

REFERENCES

1. Davis, Carl G. and Robert L. Couch. "Ballistic Missile Defense: A Supercomputer Challenge." *Computer,* 13 (1980), pp. 37–48.
2. Lincoln, Neil R. "Technology and Design Trade-offs in the Creation of a Modern Supercomputer." Accepted for publication in *IEEE Transactions on Computers,* Special Issue on Supersystems, May 1982.
3. Kartashev, S. I., and S. P. Kartashev. "Dynamic Architectures: Problems and Solutions." *Computer,* 2 (1978), pp. 26–40.
4. Kartashev, S. I., and S. P. Kartashev. "Multicomputer System with Dynamic Architecture." *IEEE Transactions on Computers,* C-28, No. 10 October 1979, pp. 704–721.
5. Vick, C. R., S. P. Kartashev, and S. I. Kartashev. "Adaptable Architectures for Supersystems." *IEEE Transactions on Computers,* C-29 (1980), pp. 1114–1132.
6. Baer, J. L. "Multiprocessing Systems." *IEEE Transactions on Computers,* C-25 (1976), pp. 1271–1277.

Software testing techniques for universal building blocks of multimicrosystems

by M. ANNARATONE and M. G. SAMI
Politecnico di Milano
Milan, Italy

ABSTRACT

VLSI components testing—in particular, concerning microprocessors—is an essential step during design and production of fault-tolerant complex systems. Actually, an efficient general method should adapt to such different phases as incoming acceptance, periodical testing, maintenance, and even design of self-testing and fault-tolerant units.

Most authors presenting this problem in recent papers advocated functional approaches as the most promising, or even the only possible, ones. In the present paper the problem is analyzed with the purpose of identifying a *general* criterion capable of leading to semiautomatic test pattern generators through formal definition of the test approach itself. To this end, microprogramming is adopted for creating the functional model of a VLSI programmable device, starting from user-available information.

The approach aims at identifying the presence of faulty behavior *error* rather than at localizing its physical source *fault*. This appears to be reasonable, given the testing criterion applications listed above. It will be seen that, although a degree of freedom exists in defining *device model* and *error model*, basic characteristics are independent of it and lead to *necessary* conditions for error coverage.

---

1. INTRODUCTION

Testing LSI and VLSI circuits is a basic problem both for manufacturers and users because of the intrinsic complexity of the devices and of the testing algorithms that could be derived by classical approaches. Users in particular are confronted with the problem of performing acceptance tests on complex devices without having adequate information about the devices' internal structures. Moreover, devices characterized as "identical" as far as external performances are concerned (typically, second-source products) may actually have completely different internal structures.

Several authors have already suggested the adoption of functional approaches to VLSI circuit testing; in fact, they advocate such approaches as the only possible ones. A widely known approach has been suggested by Thatte and Abraham.[1] They introduce a graph-theoretic model for microprocessor architectures, allowing the use of microprocessor organization and instruction set as parameters for test generation procedures. An oriented graph representing data flow among registers is derived for the instruction set, and a labeling procedure is introduced, subsequently allowing the construction of a rational test procedure going from lower to higher label values. Functional-level fault models are introduced for basic functions, and the graph model is used as a guide for generating test procedures covering such faults. An assumption already presented[2] identifies faults as related to operators rather than to structures. Although this does not exclude the introduction of structural, even physical, fault considerations, it also permits operation on a purely functional level.

The approach introduced by Courtois[3] can be considered somehow intermediate, in that, although functional faults are considered, a fairly detailed knowledge of microprocessor internal structure is required. In fact, the author foresees the possibility of gathering test information not simply "at the pins"—as Thatte and Abraham do and as it is discussed also in the present paper—but also by access to internal registers. The approach can thus be seen as oriented to manufacturers or to fairly sophisticated users capable of gaining such insight.

The problem presented by Sridhar and Hayes[4] can be considered as rather different, since the authors refer to bit-sliced microprocessors rather than to monolithic ones (as the previous ones do). This type of structure intrinsically allows far more detailed information as regards both microprocessor organization and test points availability. Actually, the case of bit-sliced microprocessors interests us because it leads to the use of an organization model based upon microprogramming concepts. Such a model can be used in cases where detailed architectural information is available, as well as in a mainly functional approach, as will be seen in the present paper.

An assumption consistently made in most papers is that the model should be derived simply from user-available information. In general, testing performed is of a static type, in the sense that timing problems are not considered; on the other hand, problems introduced by particular instructions and/or control signal sequencing are analyzed. The internal units considered in most cases are registers and functional units; i.e., faults considered are operator faults. This philosophy can be practiced even while testing the "fetch and decode" phase, since operators such as "instruction decode" and "register decode" can be used.[1]

The above assumptions are employed also in the present paper. Here it is suggested that a functional description of the microprocessor be derived, one that consists of a set of microprograms from user-available information such as instruction set, operational characteristics, and timing charts. Microprogramming as a means of abstractly representing a computer is a classical approach, and it does not necessarily reflect a physical implementation. In fact, it will be seen in the sequel that the definition of the microinstruction set strongly depends on the internal organization model derived for the microprocessor, so that different microinstruction sets can be associated with the same device; on the other hand, it will be seen that other microprogram characteristics (basic to test procedure definition) are *independent* of the model and typical of a given microprocessor.

Basically, the test problem will be seen as detection of *errors,* i.e., detection of faulty execution of microinstructions or of faulty sequencing, rather than faults. Again, this reduces to seeing faults as related to operators rather than to physical devices. When error modes are listed for microinstructions and/or micro-orders, it is possible to introduce modes deriving from physical considerations. Further, we assume a "well-defined" microprocessor; i.e., we assume that no errors arise from faulty architecture design.

It is obvious that a purely functional approach such as the one described in the present paper cannot lead to *sufficient* conditions for error coverage; but some *necessary* conditions for defining test procedures capable of fault coverage will be introduced. From these conditions, criteria permitting the definition of test procedures will be derived. Although at present it does not seem reasonable to configure a completely automated test procedure generation, a computer-assisted method is outlined.

The criterion here described is being used not only for writing incoming acceptance test programs of VLSI devices, but also for implementing periodical test routines and for designing a self-testing CPU in a high-reliability multimicroprocessor system.

2. DEFINITION OF THE MODEL

The information we consider in order to derive the microprocessor model is the conventional user-available information, defining the following:

1. The set of internal registers and functional units (control unit, ALU, and similar items) as they appear to the user through the operation of the microprocessor
2. The set of instructions and of asynchronous control signals (such as interrupts and DMA requests)
3. The behavior of signals at external pins and of internal operations in correspondence of each clock semicycle, as derived from timing charts

Information 1 makes possible the definition of a simple functional model of the microprocessor. Actually, it is usually possible to provide several models, more or less detailed; as will be seen in the sequel, above a minimum level corresponding to actual functionalities, further detail may lead to better error *localization,* not to higher error coverage.

Given this model, its operation answering the various instructions is described by means of a corresponding set of microprograms. Any given microinstruction may consist of a number of concurrent microorders; the detail of the model obviously reflects upon the choice of microorders. The testing problem becomes, in this context, the problem of identifying erroneous execution of microorders, microinstructions, or erroneous microinstruction sequencing. The microorders we consider belong to three classes: Class 1 consists of transfers either between internal registers or between an internal register and an external unit; Class 2 consists of commands to internal functional units (this includes also the control unit and associated decoding operators); Class 3 consists of branch and jump microorders. This reflects the usual classification adopted for *instructions*; obviously, at microorder level such operations as decoding also have to be taken into account.

In order to better explain how microprograms can be derived, we refer to a sample case: the Z80 microprocessor as a fairly complex and very widely known 8-bit device. The internal model is given in Figure 1. Consider first one of the simplest possible instructions, NOP; it only consists of "fetch" and "decode" phases. The corresponding microprogram can be derived in detail as follows:

1st clock semicycle	(PC) → address bus
2nd clock semicycle	request read operation from memory
3rd, 4th clock semicycle	above signals are kept stable (two s.c. delay of the control unit)
5th clock semicycle	(data bus) → data buffer; (refresh address) → address pins; refresh signal
6th to 8th semicycle	(data buffer) → control unit; (PC) + 1 → PC; instruction decode.

The incomplete knowledge available to the user, when internal transfers and operations timing are concerned, leads to the fact that we cannot introduce a one-to-one correspondence between microinstructions and clock semicycles without in-

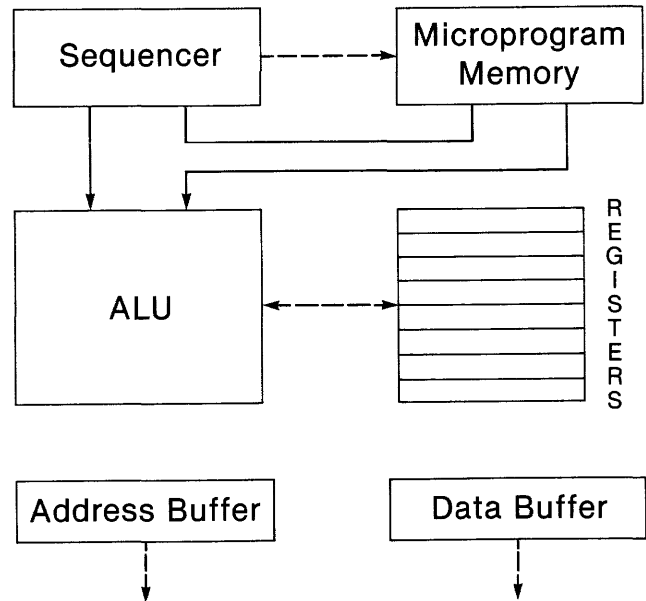


Figure 1—A general microprogrammed processor architecture

creasing the risk of creating false error localization cases. Therefore, in the sequel, we will refer to sequences of microinstructions rather than of semicycles. The "fetch and decode" phase repeats identically for all 1-byte instructions; for longer instructions, "fetch" is modified simply by the introduction of further read-and-transfer operations. Therefore, we do not here consider in detail other instances of "fetch." Consider now a simple "transfer" instruction, such as LD A,n (the value n is loaded into register A): its "execute" phase is translated by the microinstruction sequence:

1st microinstruction	(PC) → address bus
2nd microinstruction	request read operation from memory
3rd microinstruction	(data bus) → data buffer; (PC) + 1 → PC
4th microinstruction	(data buffer) → A

Microprograms for "manipulation" instructions—e.g., ADD, ROTATE—are derived in the same way. It is worthwhile to examine in detail the "execute" phase of a branch instruction, e.g.:

JP cc,nn (if condition code is set, jump to location nn)

1st microinstruction	test condition code
2nd microinstruction	conditional transfer of either (PC) or nn on address bus

When developing the set of microprograms for all the instructions, "execute" phases are obviously expanded into microinstructions sequences independently for each instruction (and addressing mode). As regards the "fetch" phase, we split it into two subphases. The first one is concerned with address generation and byte(s) fetching, and it is tested at the beginning of the test procedure, independently of results of "exe-

cute" phases; the other one concerns instruction decoding, and it can be validated only after the whole instruction set has been tested.

When defining a microprogram, any transfer operation or command must necessarily involve only registers and/or functional operators that can be (explicitly or implicitly) accessed or modified by at least one instruction; in other words, they must be derived from user-available descriptions. Therefore, while it is acceptable to represent any given register, if so desired, as the interconnection of two semiregisters operated on by parallel microorders, it is useless and actually unacceptable to represent it as the cascade of a buffer (transparent to external purposes) and a register. As a consequence, the *number* of microprogram steps (i.e., microinstructions) describing any given instruction is independent of the detail of the model, whereas the "width" of the microinstructions (i.e., the number of concurrent microorders) depends on it.

We distinguish, in a given instruction set, *directly observable* and *not directly observable* instructions. Again, this distinction is independent of the model, and it only depends on the function(s) performed by the instruction.

We say that instruction I* is *completely observable* if, at the end of its "execute" phase, signal sequences read at the pins transfer outside the result of all functions performed by I*. (This holds also for the instruction-sequencing function of the control unit, since at the end of any given "execute" phase, excepting HALT, the next instruction address is available.)

We say that instruction I* is *partially observable* if, at the end of the "execute" phase, at least one of the functions performed has been made observable at the pins. Unless I* is an instruction operating only on the program counter, we require that at least one function apart from instruction sequencing be made observable.

Finally, we say that instruction I* is *not directly observable* if, at the end of its "execute" phase, none of its functions apart from instruction sequencing has been made observable to external pins.

Obviously, in any well-designed microprocessor, for any partially observable or not directly observable instruction there exists an instruction sequence allowing the various functions to be observable at the pins. For instance, consider an ADD instruction: it falls, for most microprocessors, in the not directly observable class. By means of a "write to memory" instruction, the main result is made observable, while the Carry can also be made observable either by a conditional Branch on Carry (if available) or by a sequence "rotate-write to memory." The same observability definitions can be extended to microinstructions and microorders. For this last purpose, a microorder that cannot be made observable by means of any sequence of instructions, however complex, corresponds to an internal function totally "transparent" to the user, and therefore it is meaningless in a functional approach to testing. Should any such microorder be introduced when defining the model, it can subsequently be deleted.

We now introduce the concept of *instruction cardinality*. Cardinality is evaluated with reference to the "execute" phase. The "fetch and decode" phase is conventionally given cardinality 0, whatever the number of words constituting the instruction itself. Instructions consisting only of "fetch and decode," without any further functions (typically, NOP) are defined as having cardinality 0. *Cardinality* is defined as the number of independent accesses (possibly through functional transforms) to registers. Two clarifying statements are in order:

1. Any functional transform of a register's content must be interpreted as two subsequent accesses; that is, "read" followed by "write" after functional transform.
2. By "independent" accesses we mean actions that are *not* performed through parallel commands upon parallel units logically interpretable as single entities. Thus, an ADD operation accesses in the "write" cycle two registers (Accumulator and Carry) that, in this context, may be interpreted as one.

Now, from the set of microprograms describing "execute" phases for all instructions, we derive each instruction's cardinality. While the complete set of cardinalities for Z80 is reproduced in the Appendix, we consider here in detail some meaningful examples:

1. A "load immediate" instruction consists of the simple transfer from data buffer to named register; that is, its cardinality is 1.
2. LD r,(HL) implies two separate register accesses: the first one from register pair HL to address buffer, the second one from data buffer to r. The cardinality is 2.
3. LD r,(IX + d) implies three separate register accesses; that is, read from IX, write (IX) added with displacement d into address buffer, load from data buffer to r. The cardinality is 3.
4. POP qq: two accesses are implied; i.e., the contents of two adjacent stack words are transferred to memory and in correspondence SP is updated. The cardinality is 2.
5. JP cc nn: two accesses are implied; i.e., the Condition Code register is read and evaluated, then value nn is written in the address buffer. The cardinality is 2.
6. JR Z,e: in this case, the cardinality is 3, since the independent accesses are: read flag bit Z and evaluate, read PC contents, add e to PC contents, and write into PC.

Having identified the cardinality of each instruction, the instruction set is reordered into subsets identified by common cardinality. In the next section we will see how the model now introduced leads to the definition of testing criteria.

3. DEFINITION OF TESTING CRITERIA

The scope of testing, in our approach, is to verify for each microprocessor instruction whether at least one operation condition in the corresponding microprogram leads to faulty results, i.e., errors. To this purpose, a suitable instruction sequence will be run, possibly with different initial data sets; actually, assumptions on fault modes may be transferred into the definition of possible errors. Consider, for instance, an "LD r,n" instruction; errors in the "LD r" microorder may be defined, in the simplest case, as the incapacity for writing 0 or 1 into any single bit of r (the testing sequence will be run for the two separate instances), but they could also be made to

reflect far more complex interdependences among the various register bits. We are concerned here in verifying whether, for any given instruction with which an error set has been associated, it is possible to define a test sequence leading to (nonambiguous) coverage of all errors. How close the error set is to real error instances depends not on the "functional" model but rather on structural and physical fault assumptions underlying error definition.

It must be underlined that, given a basically functional approach, test coverage can be considered only with reference to microorders and to instructions; functional units are defined in a virtual way, and it would be meaningless to consider errors and error coverage as related to them. In the same way, it is not possible to employ any assumption of single or multiple faults. Rather, in the sequel we assume that in any microprogram there are never two microorders whose errors lead to error masking.

In the sequel, we denote by $\{i_k\}$ the set of all instructions having cardinality k, by $E\ (I_k)$ the error set associated with I_k (i.e., the set of all errors that can be originated by microorders in I_k, or by their faulty sequencing).

Theorem 1: for any I_k belonging to $\{i_k\}$, with $k>1$, there is at least one I_{k-1} belonging to $\{i_{k-1}\}$ such that

$$E\ (I_{k-1}) \subset E\ (I_k)$$

Proof: Cardinality $k>1$ can be reached by one of the two following instances:

1. Two (or more) independent registers are accessed in separate "write" operations (possibly with a functional command interposed).
2. Two (or more) independent registers are accessed, but at least one of them is accessed through a "read" command.

In Case 2 it can be immediately noted that, unless an instruction with cardinality $h<k$ is allowed to perform a "write" operation on the register(s) from which a "read" is performed in I_k, Instruction I_k would operate on registers whose content is random: in other words, a design error would be present. Case 1 actually, in a well-designed microprocessor, necessarily leads again to case 2: unless the different registers were accessed in parallel through one "write" operation (and this would not increase cardinality), there should be another "read" operation interposed, if for nothing more than to address separate "write" operations. Thus, it can be concluded (in the assumption of a well-designed and testable system) that the theorem is proved.

Corollary 1.1: The necessary condition for achieving nonambiguous error coverage with reference to $\{i_k\}$, $k>1$, is completion of tests giving error coverage for $\{i_{k-1}\}$.

Proof: Assume that coverage for $\{i_{k-1}\}$ has not been achieved. This means that at least one register access (possibly through a functional operator) has not been tested and that one or more errors in at least one $E(I_{k-1})$ have not been covered. Since there may be an I_k containing a microorder capable of masking such error(s), ambiguity in testing may result. Moreover, since an access error in $E(I_k)$ may counterbalance a previous access error in $E(I_{k-1})$, error coverage may not be achieved.

Up to now we have made use of the concept of "instruction sequence" as the means for testing an instruction I_k; in fact, while in the simplest case of a directly observable instruction not requiring any setup for testing such sequence consists of the instruction itself, in all other cases there will be a sequence $I_i^a \ldots I_l^b$.

Theorem 2: The necessary condition for a nonambiguous error coverage for a not directly observable instruction $I_k^j \in \{i_k\}$ is the existence of a testing instruction sequence $I_i^a \ldots I_l^b$ consisting only of instructions with cardinality not greater than k.

Proof: Assume first $l>k$. I_l^b is of necessity observable (at least partially), but if the condition in Corollary 1.1 has been satisfied, I_l^b has not yet been tested; therefore, it is possible that I_l^b masks error(s) in I_k^j. Now let the sequence be $I_i^a \ldots I_m^c \ldots I_l^b$, with $m>k$, while all instructions following I_m^c and comprising I_l^b have cardinality $\leqslant k$. We can assume that all instructions following I_m^c have already been tested and that they do not therefore introduce further error possibilities; the problem, then, relates only to I_m^c, and it can be reduced to the previous considerations.

Obviously, the instruction set of a microprocessor can be such that Theorem 2 is not satisfied; for instance, Z80 exhibits this problem. In fact, we may have two different instances:

1. there is the not directly observable instruction I_k^j whose simplest testing sequence, making it observable, requires use of an I_h^l (with $h \geqslant k$) that in turn requires I_k^j in its setup sequence. For instance, a "load register immediate" ($k=1$) can be made observable by means of a "write register to memory" ($h=2$) (any other possible instruction sequence would involve instructions of no lower cardinality), but this in turn requires in its own test sequence at least one "load immediate" for setup. We can accept a "temporary" test result for the pair $I_k^j\!-\!I_h^l$. At the end of the testing actions, if there is at least one sequence relating to cardinality h (or higher) that
   -does not involve the pair $I_k^j\!-\!I_h^l$
   -is not critical for testing any instruction I_h^i
   -makes I_k^j observable by means of separately tested instruction(s),

   then I_k^j (and possibly I_h^l) can be separately tested, and nonambiguous error coverage can be reached. Otherwise, the test is valid only with respect to the pair $I_h^l\!-\!I_k^j$, and error masking is possible.
2. Any test sequence for I_k^j involves instructions with higher er cardinality, but none of these requires I_k^j in its test sequence. Testing on I_k^j will be "suspended" until all instructions in its test sequence have been validated.

Note that, while conclusions in Theorem 2 are independent of error hypotheses, possible instances of ambiguity and error masking when Theorem 2 is not satisfied strongly depend on definition of microinstructions and of error hypotheses. All

the above leads to identifying error coverage and nonambiguities when defining test sequences. In the next section we outline testing procedures based on such considerations and accounting also for fast test sequences.

4. DESIGN OF TESTING PROCEDURES

We do not at present consider it possible to implement automatic definition of testing procedure following our approach; rather, the criteria we introduce allow computer-assisted development of such procedures (it is possible to make such steps as congruence verification and minimization automatic).

The first step, definition of microorders and error set, must necessarily be performed by hand. Subsequently, cardinalities are computed for all instructions, and an error set is associated with each instruction. Error sets of two different instructions may have a nonvoid intersection, but they cannot completely overlap, since (at least) instruction decoding errors will be different. Thus it will not be possible to exclude any instruction from the set of test sequences; on the other hand, whenever a subset of the error set has already been covered by other test sequences, it becomes unnecessary to introduce the related test actions in new test sequences. Actually, this criterion has been widely adopted in hardwired-logic testing; here, we simply extend it in the context of programmable logic.

The instruction set can be further reordered on the basis of "functional families" (e.g., Load, Add). In each of these families, minimum cardinality is identified; an automatic congruence validation is now possible regarding evaluation of cardinalities, through a comparative examination of the various families. In each family, instructions are ordered following increasing cardinalities; obviously, this does not imply that in any given family (or, in fact, in the whole instruction set) cardinalities corresponding to all integers between minimum and maximum values will be present.

Having completed this "setup," the basic functions of the "fetch" phase (excluding instruction decode) are tested by using 0-cardinality instruction(s). When variable-length instructions, and thence "fetch" phases, are present, the corresponding "fetch" sequences will be tested once by means of the instruction of corresponding length with simplest functionality.

Afterwards, considering the set $i_1, i_2, \ldots, i_n$ of cardinality values, and starting with the minimum value i_1, instructions are tested following criteria outlined in the previous section. For each instruction we look for the fastest test sequences allowing us to cover all (or the maximum number) of the associated error set, or, better, all such errors not already covered by previous test sequences. Whenever more than one such sequence is identified, preference is given to the one better satisfying the criteria given in Section 3. Inside a given set $\{i_k\}$, a practical rule may be to start by defining, whenever possible, test sequences for instruction belonging to families with simplest functionality—following a "start-small" philosophy, as advocated by most researchers in the field.

Obviously, our approach allows to "reduce" the length and complexity of test sequences set, not to definitely minimize them, since the possibility of reduction of test sequences for any I_k strongly depends on choices performed previously for all other I_l $(l < k)$ already considered. Examples of instruction ordering and error set definition and the outline of a testing procedure (related to Z80) are given in the Appendix.

5. CONCLUDING REMARKS

An approach in terms of microprogramming to functional modeling of microprocessors makes it possible to guide testing procedures so as to satisfy some necessary conditions for error coverage. Error classes are defined as corresponding to microorders, i.e., again in functional terms; the complete error set, on the other hand, derives from physical fault assumptions. Model and error classes can be built from user-available information; coverage is considered with respect to error classes.

The approach is completely general, and it can be used also in the case of intrinsically microprogrammed bit-sliced systems. It seems possible to adopt it also for a wider class of programmable microdevices, such as intelligent interfaces.

Though the functional criteria used are (at least partially) present in previous literature, use of a formal methodology such as microprogramming and of ordering and classification methods related to it makes it possible to formulate semiautomatic systems for test procedure generation.

Test procedures for one microprocessor are being completed; their use will allow an experimental evaluation of the approach.

REFERENCES

1. Thatte, S. M., and J. A. Abraham. "Test generation for general microprocessors architectures." Ninth IEEE Fault Tolerant Computing Symposium, 1979, pp. 203–210.
2. Thatte, S. M., and J. A. Abraham. "A methodology for functional level testing of microprocessors." Eighth IEEE Fault Tolerant Computing Symposium, Toulouse, 1978.
3. Courtois, B. "On line oriented functional testing of control sections of integrated CPUs." *Proceedings of Euromicro 81,* Paris, September 1981. Amsterdam: North-Holland, pp. 221–231.
4. Sridhar, T., and J. P. Hayes. "A functional approach to testing bit-sliced microprocessors." *IEEE Transactions on Computers,* C-30 (1981), pp. 563–571.

APPENDIX

Classification of the Z80 complete instructions set, following the ordering on increasing cardinality–increasing functionality, is given in Table I. Let us consider a simple example in order to outline the definition and choice of test sequences.

Consider ADD A,n $(k = 2)$. Criteria introduced in Sections 3 and 4 make it possible to state that when testing it, all Cardinality-1 instructions will have been tested (at least in a preliminary way), and so will branch and transfer instructions of Cardinality 2. In particular, therefore, errors related to microorders—"access register A (read/write)" and "access data buffer"—will have been covered; only errors related to "ADD command to ALU" and "access carry bit (write)" must be covered, besides instruction decoding. Thus, test sequence will be as follows:

LD A,m	Values of m, n are chosen (and the sequence is repeated)

ADD A,n	only with reference to the ADD and Carry microorders error set
LD qq,HL	
LD $(HL),A$	Add microorder is made observable
RRC A	
LD $(HL),A$	Carry is made observable

If now ADD A,r (still with Cardinality 2) is considered, only the instruction decode will be tested, since accesses to r, A, and ALU have already been covered.

The same holds at Cardinality 3 for instructions such as ADD $A,(HL)$. Actually, instruction decode testing involves a full sequence related to *basic* functionalities, but it need not be repeated for access or operators validation; all functionalities deriving not from instruction decode and microprogram sequencing, but from correct operator working, will also be excluded from the sequence (in our example, the test on carry setting).

Class of Card.	Classific. of instruc.	Instructions
ONE	Branch	JP nn / RET / RETI / RETN / JP(HL) JP(IX) / JP(IY)
	Transfer	LD r,r' / LD r,n / LD A,I / LD A,r LD I,A / LD R,A / LD dd,nn / LD IX,nn LD IY,nn / LD SP,HL / LD SP, IX LD SP,IY / IN A,(n) / IN r,(C) OUT (n),A / OUT (C),r
	Manip.	CP s / CPL / DAA / RES b,r / RLA / RRA RLCA / RRCA / RL r / RR r / RLC r RRC r / SCF / SET b,r / SLA r / SRA r SRL r
	Other	HALT / DI / EI / IM 0 / IM 1 / IM 2
TWO	Branch	CALL nn / JP cc,nn / JR e / RET cc
	Transfer	LD r,(HL) / LD (HL),r / LD (HL),n LD A,(BC) / LD A,(DE) / LD A,(nn) LD (BC),A / LD (DE),A / LD (nn),A PUSH qq / PUSH IX / PUSH IY POP qq / POP IX/ POP IY EX DE,HL / EX AF,AF'
	Manip.	ADD A,n / ADD A,r / ADD HL,ss ADD IX,pp / ADD IY,rr / AND s / BIT b,r CCF / DEC IX / DEC IY / DEC ss / DEC r INC r / OR s / RES b,(HL) / RL (HL) RR (HL) / RLC (HL) / RRC (HL) SET b,(HL) / SLA (HL) / SRA (HL) SRL (HL) / SUB s / XOR s

TABLE Ia—Instructions ordered by their cardinality

Class of Card.	Classific. of instruc.	Instructions
THREE	Branch	CALL cc,nn / JR c,e / JR NC,e / JR Z,e JR NZ,e / RST p
	Transfer	LD r,(IX+d) / LD r,(IY+d) / LD (IX+d),r LD (IY+d),r / LD (IX+d),n / LD (IY+d),n
	Manip.	ADC HL,ss / ADC A,s / ADD A,(HL) BIT b,(HL) / NEG / RES b,(IX+d) RES b,(IY+d) / RL (IX+d) / RLC (IX+d) RR (IX+d) / RRC (IX+d) / RL (IY+d) RLC (IY+d) / RR (IY+d) RRC (IY+d) SBC A,s / SBC HL,ss / SET b,(IX+d) SET b,(IY+d) / SLA (IX+d) / SLA (IY+d) SRA (IX+d) / SRA (IY+d) / SRL (IX+d) SRL (IY+d)
FOUR	Branch	DJNZ e
	Transfer	LD HL,(nn) / LD dd,(nn) / LD IX,(nn) LD IY,(nn) / LD (nn),HL / LD (nn),dd LD (nn),IX / LD (nn),IY / EX (SP),HL EX (SP),IX / EX (SP),IY
	Manip.	ADD A,(IX+d) / ADD A,(IY+d) BIT b,(IX+d) / BIT b,(IY+d) / DEC (HL) INC (HL) / RLD / RRD
FIVE	Manip.	DEC (IX+d) / DEC (IY+d) / INC (IX+d) INC (IY+d)
SIX	Transfer	EXX / INI / IND / OUTI / OUTD
	Manip.	CPD/ CPI
SEVEN	Transfer	INIR / INDR / OUTIR / OUTDR

TABLE Ib—Instructions ordered by their cardinality

Class of Card.	Classific. of instruc.	Instructions
NINE	Transfer	LDI / LDD
TEN	Transfer	LDIR / LDDR

(*) NOP instruction → cardinality zero

TABLE Ic—Instructions ordered by their cardinality

A methodology for the development of special-purpose function architectures

by RAYMOND A. LIUZZI*
Rome Air Development Center
Griffiss Air Force Base, New York

and

P. BRUCE BERRA
Syracuse University
Syracuse, New York

ABSTRACT

The research described in this paper concerns a generalized methodology for the development of special-purpose function architectures (SPFA). The development methodology can be used to introduce the concept of an SPFA approach to an organization.

The methodology provides an organized set of processes that can be followed to tailor the development of SPFAs to specific applications. This methodology consists of processes for identification, creation, testing, evaluation, and substitution of SPFAs. It permits a user to carefully select sets of database management functions as candidates to be moved from software into hardware, develop one or more SPFAs that perform this function, and evaluate the consequences of having the function performed as a new hardware architecture. A set of tools/components with which to carry out this methodology are included in the environment of a proposed database machine architecture development facility.

*This research was performed while the author was pursuing the Ph.D. degree at Syracuse University.

INTRODUCTION

Interest in computer architecture research, as applied to database management, has recently increased because of the advancing state of the art in inexpensive, fast new hardware components. Hardware technology is advancing primarily in three areas: central processing units (CPU), semiconductor random-access memory (RAM), and all-electronic bulk memories. The cost-to-performance ratio of CPUs will decline rapidly over the next 10 years. Low-cost CPUs with the performance capabilities of today's medium-priced minicomputers will be available for hundreds of dollars in the 1980s. New technologies in memories using bubbles or charge-coupled devices will rival existing fixed-head discs.

These trends have made it feasible to examine new hardware architectures that can perform database management system (DBMS) functions currently performed in software. How to determine which functions to implement in hardware and how to choose their optimal architecture, for a given user application, becomes a very difficult task. In order to help ease this transition, database machines have been introduced as new hardware architectures designed to perform DBMS functions.[5]

The notion of a database machine (DBM) has evolved primarily because of the need to accomplish database management tasks more efficiently. The evolution to the current class of DBMs can be traced by viewing Figure 1. DBMSs were originally developed to execute on large sequential systems and had to rely on the services of a generalized host execution system to perform many of their tasks. Examples of this class of system include the IDS system on an H6000, IMS on an IBM 360/370, and System 2000 on several large machines.[15] However, much of the processing efficiency of these systems is compromised by the inefficiency of I/O operations for processing data. The operating systems on these large machines must multitask a number of activities.

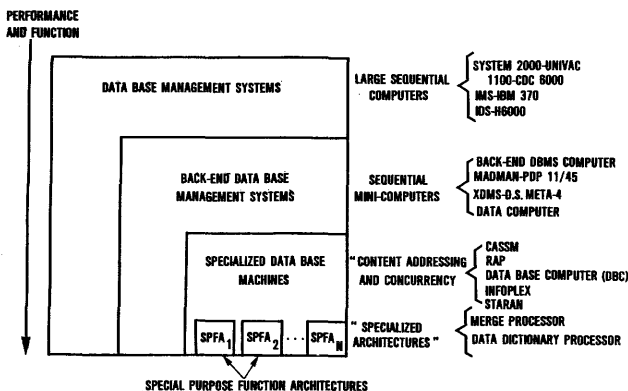


Figure 1—Database machine evolution

As minicomputer development progressed, it became apparent that many database management tasks could be accomplished more efficiently by removing them from the large sequential machine to a machine dedicated entirely to database management. Such a machine is called a back-end machine. Canady et al. outlined an architecture for performing various DBMS tasks on a back-end Digital Scientific Meta-4 Computer.[2] Numerous advantages were cited, including security, reliability, and efficiency.

Further development in semiconductor technologies has produced the microprocessor and the microcomputer, along with the notion that it is economically feasible to develop computers that are primarily designed for database management. Previous research efforts have substantiated the fact that computer architectures that provide concurrency and content-addressing constructs can provide order-of-magnitude increases in the performance of certain database management functions.[20,26] As a result, a new class of machines has emerged, with various unique architectures designed to provide these constructs. Liuzzi and Berra[17] have defined a set of characteristics for a range of these DBMs that consists of the following:

1. An overall architecture composed of one or more special-purpose function architectures (SPFAs)
2. An architecture based on parallelism and content addressing
3. A set of compatible memory units for the storing and efficient retrieval of data
4. An architecture that is a back-end machine.

Several types of machines have been reported in the literature with these characteristics; some are given below.

The logic per track architecture of the University of Florida's Context Addressed Segment Sequential Memory Organization (CASSM) System,[6] the multicell CASSM system,[29] and the University of Toronto's Relational Associative Processor (RAP) system[24,25] attain concurrency by moving logic from the central processing unit to the individual disk heads that read data from each track on a fixed-head disk. The RAP system has recently been extended to include a semiconductor charge-coupled device (CCD), random-access memory (RAM), or bubble memory.

The Ohio State Data Base Computer (DBC) proposed by Hsiao[1,13] consists of a unique architecture that interconnects several specialized processors aimed at supporting secure large-scale databases. Each database is stored on content-addressable moving-head disk devices, and emerging technologies such as magnetic bubbles and CCDs have been chosen for part of the system.

The INFOPLEX system proposed by Madnick[18] takes

advantage of new memory and processor technologies to organize a smart memory hierarchy to handle the storing and retrieval of information. Its information management functions are decomposed into a functional hierarchy implemented by a hierarchy of microprocessors.

The DIRECT system proposed by DeWitt[9] is a multiprocessor organization for supporting relational DBMSs. DIRECT has a multiple-instruction, multiple-data stream architecture. It can simultaneously support both interquery and intraquery concurrency.

Associative processors have been experimentally examined for database management applications. Early studies by DeFiore, Stillman, and Berra[8,28] using the Goodyear Associative Memory and later the STARAN Associative Array Processor established that searching a database was significantly improved by associative techniques.

The RELACS system proposed by Oliver[22] is a DBMS using associative processors to implement the relational data model. RELACS is capable of supporting many functions of a database management system including retrieve, updating, deletion, modification, and addition.

In addition to this class of DBMs, a set of specialized architectures have emerged. As Figure 1 indicates, these specialized architectures can form portions of a DBM. They are primarily designed to optimize a single database management function and are called special-purpose functional architectures. Several different types of database functions have been designed as SPFAs.

Roberts[23] has proposed a specialized parallel computer architecture for high-speed searching of large textual files. The database to be searched is partitioned among independent high-speed serial-access memories that are searched in parallel by dedicated microprocessors connected to a common communication bus.

Hollaar[10] has proposed a specialized merge processor that combines data from sorted input lists into a sorted output file. This processor is designed with architectural constructs that form the merge operation. Stellhorn[27] proposed an inverted file processor that uses a specialized architecture to access files of document identifiers and perform the processing associated with a Boolean search request. Hollaar and Stellhorn[11] propose a specialized architecture for textual information retrieval. The basic architecture of the system consists of several parallel search modules connected to a disk via a parallel/serial interface. This architecture is especially suited for list merging, updating, and sorting operations. Hollaar[12] has also extended this work to include the design of a list-merging network.

Mukhopadhyay[21] has proposed specialized hardware algorithms for nonnumeric computation. These algorithms can be implemented with various LSI technologies for high-speed pattern-matching needs.

Capraro[3] has proposed to integrate a data dictionary as an SPFA using associative processors. Singhania and Berra have designed a special-purpose function architecture using associative memories for pipelining a directory to a very large database. The results of this study indicated that the pipeline system provides faster retrieval than sequential inverted list systems, especially in the case of multiple-key retrievals. Karlowsky and Leilich[14] from the Technical University of Braunschweig have proposed an SPFA called a search processor to search data stored on a mass memory without using the CPU and I/O of a host computer.

The Content Addressable File Store (CAFS) is a specialized hardware architecture that performs parallel processing techniques for implementing multifactor selection across either single files or the join of multiple files.[19] This SPFA performs concurrent execution of powerful selection and retrieval functions on multiple data streams arising from the simultaneous reading of many disk channels.

The introduction of these SPFAs has provided users with a new approach to gaining specialized improvements in their database applications. Each of the SPFAs described can replace a DBMS function or functions currently being performed in software on a sequential machine. This notion that software functions can be either improved or replaced by hardware has been characterized as the SPFA approach. This paper examines the effect the SPFA approach can have on an organization, describes the need for an organized methodology to introduce the SPFA approach to an organization, and presents a generalized methodology that can be used to help in the development of SPFAs.

NEED FOR METHODOLOGY

The emergence of various types of SPFAs has prompted the need for an organized methodology that can be used to develop SPFAs for specific user applications. Typically, an examination of a DBMS shows that it is composed of several types of functions. First, a set of basic functions for each DBMS is used to manipulate data into a form acceptable to the application program. Examples of these functions are search, update, and modify. A second set of functions maintain data in a data dictionary or a database. The functions permit a logical expression of the database and maintain a physical access to the stored data. Next, a set of functions provide user interface capabilities via query generation modules and request generation modules. These functions provide various levels of natural user interface to a DBMS. Finally, a set of application modules is used to support functions that provide various editing capabilities to a DBMS user. Each of these functions are typically performed in software and are candidates to be developed as SPFAs.

If an organization wishes to seek ways to upgrade a current DBMS capability, it might want to introduce one or more of these functions as SPFAs in the form of hardware assist modules into its current environment. However, in order to exploit this SPFA approach fully, several questions must be examined:

1. What are the important factors to consider when choosing a function that can be developed as an SPFA?
2. What are the various algorithmic approaches and architecture considerations to implement the function in hardware?
3. How will new hardware technology affect the function's implementation in terms of performance, cost, reliability, and other relevant matters?

For each database management function that is a candidate for a move to hardware, several architectural options may be available. To evaluate each option, designers may have to build the actual hardware. If more than one architecture is being considered, several options may have to be built. Once these hardware options are built, procedures to test and evaluate them need to be established. However, if several SPFA options exist for a given function, the actual hardware construction may not always be feasible because of the following three problems:

1. The expense of actually developing a number of hardware options
2. Time constraints
3. Inability to alter each SPFA easily after it has been built

Finally, several factors must be considered in choosing the actual hardware technology used to implement a SPFA. The technology chosen by a user depends on specific user application requirements. For instance, a comparison of competitive technologies that may be used for an implementation may indicate that one is faster than the other but is less reliable. Another factor may indicate that one may improve performance, but at a higher cost.

Thus, a generalized methodology can be very useful in providing an organized mechanism to introduce SPFAs for improving overall DBMS capability. The methodology must consider choice of DBMS functions, architecture options for the function, and implementation strategies for the function for each specific user application. A methodology has been developed[16] and can be used in conjunction with a database machine architecture development (DMAD) facility. This methodology and a brief description of the DMAD facility are now described.

METHODOLOGY TO DEVELOP SPFAs

A methodology to develop SPFAs requires the following set of processes:

1. Select candidate function.
2. SPFA create.
3. SPFA test.
4. SPFA evaluate.
5. SPFA substitute.

The select-candidate-function process helps determine DBMS software functions that are candidates for replacement as hardware architecture SPFAs.

The create process transforms a description of each SPFA from a set of architectural considerations to a set of language statements. This set represents a functional description of an architecture that performs the DBMS function.

The test process functionally verifies that the SPFA performs the desired DBMS function. This process permits the DBM architect to examine the architectural constructs of the SPFA to insure that it meets its design goals.

The evaluate process enables the DBM architect to evaluate competing SPFAs. An architecture evaluation is conducted to

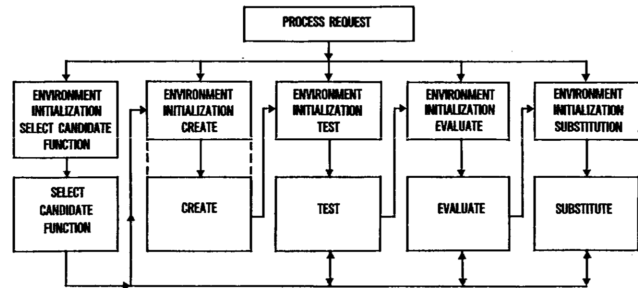


Figure 2—Development process flow model

generate performance timings of the SPFAs. These timings are based on using an assumed set of hardware characteristics to perform operations required by each SPFA.

Finally, the substitute process is composed of a set of procedures to enable the DBM architect to selectively integrate an SPFA within a DBMS capability. The substitute process helps the DBM architect assess the effect on the system of having a DBMS software function replaced by hardware.

The complete development is illustrated by a process flow model in Figure 2. The flow of this model indicates that the environment is initialized for each process request. This initialization configures the tools needed to complete a process. Several feedback loops are provided in this model to allow refinements during the development of the SPFA. These loops permit reuse of the complete set of tools for all processes. For instance, after an evaluation process is completed, the DBM architect may choose to alter an SPFA by modifying a portion of the architecture description. This may result in the recreation of the SPFA. Similarly, a single process can be repeated interactively so that an exhaustive series of tests or evaluations can be performed.

In addition, the final process, substitute, permits the DBM architect to assess the effect of the newly developed SPFA on the DBMS system. This process is used to integrate the SPFA into the DBMS and to help determine potential problems. The data collected following this process can help determine if the function is a logical candidate to be moved to hardware for a specific application.

The use of the process flow model also permits a user to tailor an SPFA development to a specific application. This helps insure that the developed SPFA meets the unique requirements of the application.

ARCHITECTURE/INTERFACES OF THE DMAD FACILITY

A database machine architecture development facility has been proposed[16] as a specialized environment that hosts the tools and components needed to perform each of the processes in the generalized methodology. The DMAD facility consists of the following components, illustrated in Figure 3:

1. a service host machine (SHM) that is responsible for monitoring requests in the DMAD facility, staging input for the database function execution machine (DBFEM),

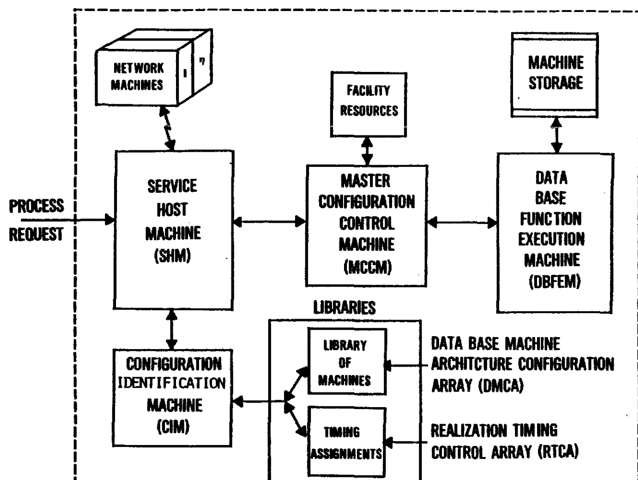


Figure 3—Architectural/interfaces of the DMAD facility

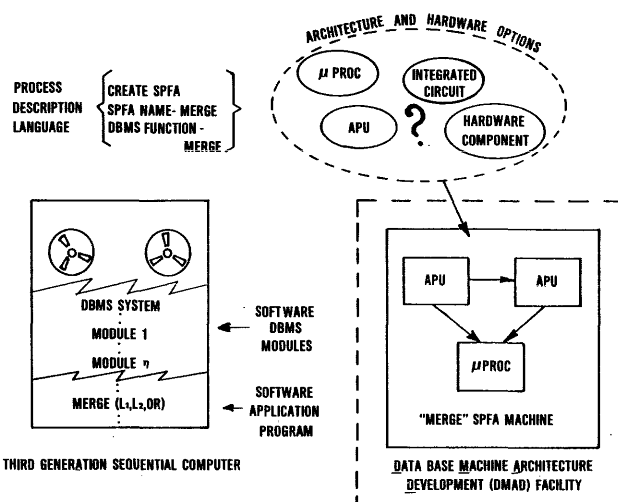


Figure 4—Developing a special-purpose function architecture

and providing a programming environment to described SPFAs. The SHM interfaces to network machines that may help in the execution of a SPFA.

2. a database function execution machine (DBFEM) that is responsible for hosting the execution of SPFAs as machines in the facility. This machine serves as a back end to the SHM and is capable of emulating a variety of computer architectures.
3. a master configuration control machine (MCCM) that interfaces the SHM and DBFEM. This machine acts as the configuration manager for process requests to the DMAD facility. In this capacity the machine controls resources needed to support execution of an SPFA machine on the DBFEM.
4. a configuration identification machine (CIM) that interfaces to the SHM and is used to identify configuration requirements needed to execute SPFA machines. Specialized libraries are maintained and can be loaded on the CIM to help identify these requirements.

ILLUSTRATION OF THE SPFA APPROACH

An example of introducing the SPFA approach to an organization is shown in Figure 4. Assume that an organization requires improving performance and reliability in merging lists for its present applications. This MERGE function is currently being performed in software, as part of a DBMS, on a sequential computer. However, several competitive new merge hardware architectures can also perform this function.[11,27] This organization needs to choose the merge hardware architecture that can optimize the overall performance and reliability of the MERGE function for this application.

Within a DMAD facility the merge hardware architectures are created as SPFA machines to perform the DBMS MERGE function. For instance, illustrated in Figure 4 is a MERGE SPFA machine that is created from among several architecture options and is introduced to the DMAD facility. The SPFA is described in hardware description language. This description is compiled and debugged to produce an executable version. When this version is complete, it is configured in the DMAD facility to produce a MERGE SPFA machine. During execution, complete control of the SPFA machine is maintained by a user with access to the facility. This permits the examination of all states of execution. Testing in this environment is done with a set of tools to verify that the SPFA performs its intended DBMS function.

After testing, an evaluation of the SPFA is performed. This evaluation consists of accumulating the time needed by the SPFA machine for a sequence of its operations. The timing of these operations is chosen by examining possible hardware implementations and associated timings.

For instance, in this example, since both performance and reliability improvements are sought, a user can examine hardware technologies that have high reliability characteristics as candidates for the MERGE SPFA machine's operations.

In order to assess the effect of varying these choices, one or more realization assumptions can be described. Each choice becomes a separate realization of the SPFA and is used to generate separate sets of data on performance and reliability.

Once performance and reliability data are established for a specific SPFA description, the procedure described above can be repeated by varying some architectural features of the original SPFA description or developing one of the competitive MERGE SPFAs. This process can continue for a number of architectural options that may be available for this function.

Once a MERGE SPFA is chosen, the next development stage can be its substitution within a current DBMS. This process can be performed as illustrated in Figure 5. First, a computer system that can support a set of database management functions is referred to as a DBMS machine. Next, assume that this DBMS machine is emulated to execute in the DMAD facility. A DBMS software application program (SAP) is chosen to execute on the DBMS machine and call the services of the DBMS functions supported. The SAP typically calls a sequence of DBMS functions such as FIND, ORDER, MERGE, CLOSE, etc., as illustrated in Figure 5. Whenever the MERGE function is called, the option chosen for the

MERGE SPFA machine is executed instead of the original MERGE function. The actual interfaces and the effect of substituting the MERGE SPFA can now be examined in terms of pertinent hardware/software tradeoff issues.

A positive assessment of the SPFA's integration may lead the organization to choose to actually build a hardware prototype Merge SPFA.

SPFA DEVELOPMENT METHODOLOGY PROCESS FUNCTIONS

Described in this section are a set of procedures for the generalized methodology to help develop SPFAs. This methodology can be divided into the following phases:

1. identification of candidate DBMS functions
2. creation of SPFA's
3. execution of an SPFA machine

Identification of Candidate DBMS Functions

The identification of candidate DBMS functions to be moved from software to hardware is made by examining the typical requirements of a range of applications. Pertinent factors include current usage of the function, the ability to clearly define interfaces to the function, and the potential of the function for improving system performance and cost. This process requires examination of several functions that are portions of a current DBMS.

A DBMS machine identification procedure is used to configure emulations of machines that support current DBMSs. Each machine is configured with a DBMS, required system support software, and sample application programs. A set of DBMS functions within the DBMS are then identified as candidates for being replaced by hardware. These functions are currently performed by sets of software modules that are executed when the function is called.

A variety of criteria may be used in selecting candidate functions. These include the frequency of a function's use

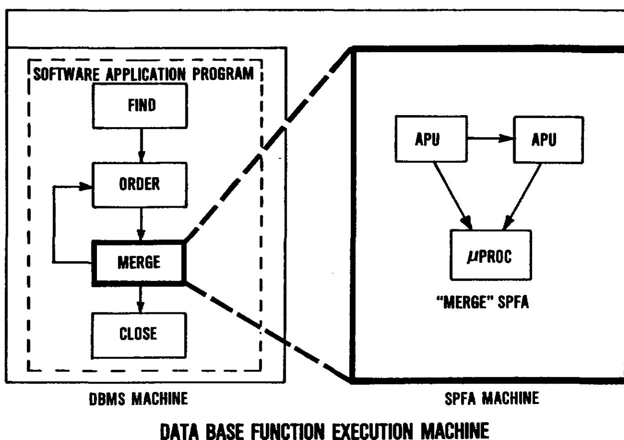


Figure 5—Substitution of a special-purpose function architecture

(i.e., number of calls), the amount of time taken to execute the function, the complexity of the function in relation to other functions within the DBMS, and the potential for improvement in DBMS system quality if the function is moved from software to hardware.

Statistics on use of a specific function can be obtained by establishing break points at the entrance to the software module performing the function during execution. The rate of use can be determined from the number of times the break point is encountered. Another procedure is to use a performance monitoring tool to identify frequency of calls for a DBMS function. Such a capability may also be provided in conjunction with the description of each emulated DBMS machine by establishing a count for a specific instruction execution. For instance, the SMITE hardware description language provides a performance capability[30] that permits a software monitor to accumulate the number of times an instruction is encountered during execution.

Once data on use are collected, the actual execution path of frequently called functions can be examined. This path is examined by actually stepping the execution of the function, instruction by instruction. This permits all entrances and exits to and from the function to be properly identified and documented. This procedure helps to identify the complexity of this function in relation to other functions and to identify all the interfaces required to and from the function within the DBMS.

Next, quality considerations may be examined. Such an analysis is based on assessing the overall improvement in the quality of the system that may be obtained by moving the function to hardware. For instance, will the movement of this function to hardware increase DBMS system performance but at the same time decrease the system's portability or reliability?

Questions such as these may be examined by establishing metrics for specific quality factors concerning the DBMS function. These metrics can be computed for such factors as reliability, maintainability, and flexibility.[4] If the movement of this function to an SPFA can improve the quality of the DBMS in relation to a given application, then it may become a candidate function.

In summary, the choice of specific DBMS functions to be moved from software into hardware may be based on criteria such as use, performance and complexity, and quality improvements gained within the system. Once one or more candidates are selected, competitive SPFAs that can perform the desired DBMS functions must be examined.

Creation of SPFAs

A create process is selected to describe an SPFA that performs a candidate DBMS function. The objective of the create process is to translate a conceptual architectural description of an SPFA into an executable SPFA machine. This process consists of two procedures:

1. SPFA description development
2. SPFA introduction

The SPFA description development procedure requires the specification of a set of architectural constructs that, when executed as a machine, perform a DBMS function. These constructs can be specified in a hardware description language that defines a machine representation of an SPFA.

A specialized programming environment that exists within the DMAD facility is used to describe each SPFA in a hardware description language (HDL). This includes identifying the machine representation of the SPFA in terms of registers, interconnections, flow of information, and specific operations. Both control and concurrency dependencies among SPFA operations are described. As part of a description, each SPFA is created in such a way as to be fully compatible with the same interfaces as the software function it will replace. Once completed, a compilation procedure translates this SPFA description into a source and subsequently into an object file. The source file is debugged within the programming environment to eliminate source programming errors. If errors are identified, the SPFA description is modified and recompiled. This procedure continues until a correct SPFA is described. The SPFA compilation also produces an object file that consists of a set of microinstructions that are compatible with the DBFEM. These microinstructions are used to transform the facility into an SPFA machine using an SPFA introduction procedure.

This procedure identifies each SPFA to the DMAD facility. SPFAs introduced to the facility are entered into a database machine architecture configuration array (DMCA). This array is used to identify configuration requirements needed for executing each SPFA as a machine in the facility.

Execution of an SPFA Machine

An SPFA machine is ready for execution once all configuration resources are made available. These resources include access to the database function execution machine and any other specialized resource support. When an SPFA machine begins execution, the processes of testing, evaluation, and/or substitution can be performed.

The objective of the testing process is to determine whether the SPFA machine accurately performs the desired DBMS function. The testing process consists of execution of the SPFA machine by means of test cases, verification of the proper sequence of SPFA machine states, and debugging the SPFA machine.

In order to begin testing, a set of test cases are specified. They consist of specific input to the SPFA machine to insure that it executes properly.

The SPFA machine executes by moving from state to state. A state can be specified at the SPFA source description level or at the microinstruction level. The microinstruction level permits identification of states at a much lower level than the source level. The level may be needed for detailed verification or debugging the SPFA machine. Types of testing capability include tuning, verification, probe, and visual examination of the machine.

One verification technique that can be used is the examination of the states of the machine at given instants of time. This examination can be conducted by establishing control points in the SPFA description. When these control points are reached during execution of the SPFA machine, a DBM architect performs an extensive verification of the state of the SPFA machine. For instance, registers, information resources, and control indicators can be examined. If they conform to preselected values, the state verification of the SPFA machine is established. However, if an error or inconsistency is found at one of these control points, the SPFA machine may not be verified, and debugging procedures are necessary.

A specific procedure that can be used to formally verify states of a SPFA has been proposed by Crocker.[7] This procedure examines the execution of an SPFA machine and refers to a state change as a state delta. These state deltas are then examined in relation to predefined lists. If a state delta results in the formation of an improper list, the execution of the SPFA machine during the delta is questioned for possible error.

The debugging procedure performed in the facility for an SPFA machine consists of the identification of an error and the isolation of its causes by the DBM architect, who isolates the error by verifying the SPFA machine states until the error occurs. This isolation can be performed at the source description level or at the microinstruction level, if necessary, to insure that the error is found.

During debugging the DBM architect views the actual representations of the SPFA machine. This permits all name variables and conditions in the SPFA description to be inspected. During the debugging exercise the DBM architect has complete control of the SPFA machine and can step it through various execution states.

After the SPFA machine has been tested, the evaluation process may be requested. The objective of this process is to evaluate SPFA machines by using hardware operations identification and SPFA performance procedures.

The hardware operations identification procedure examines each SPFA description to identify specific hardware operations. A realization assumption (RA) defines a set of specialized hardware implementation characteristics for these operations, and these characteristics are used to generate timings for the specific hardware operations identified for an SPFA. The times are entered into a realization timing control array (RTCA) for each realization assumption. Each SPFA machine is then executed, and the results of separate executions are collected and analyzed.

Once an SPFA has been evaluated, the final execution process is substitution. The objective of this process is to investigate the effects of substituting an SPFA for a candidate software DBMS function. The substitution process consists of DBMS/SPFA machine identification, selective integration, and evaluation.

The DBMS/SPFA machine identification procedure consists of identifying the configuration requirements used to transform the DMAD facility into a DBMS/SPFA machine. A set of requests are executed on the DBMS/SPFA machine that call the supported DBMS functions. These functions are performed by the appropriate sets of software modules. However, when a request is made for the specific DBMS function supported by the SPFA, the SPFA machine is automatically called to perform the DBMS function.

A DBMS/SPFA machine selective integration function pro-

cedure performs the actual substitution of the SPFA machine. A virtual database machine monitor (VDMM) can be used as the tool for this procedure. The VDMM is designed as a virtual machine monitor that supports control of both a DBMS machine and an SPFA machine on a microprogrammable execution machine. The VDMM permits the switching of control between a DBMS machine and an SPFA machine. Each time an SPFA machine is called, it completes the DBMS function, then returns control to the DBMS machine.

A DBMS/SPFA machine evaluation procedure enables a user to assess the impact of integrating an SPFA machine and a DBMS machine. This procedure includes executing the same set of requests to supported DBMS functions with and without the presence of the SPFA machine. This procedure is used to identify any further interface problems that may result from the presence of the SPFA machine, helps assess the system feasibility of having the function performed as an SPFA, and enables performance comparisons to be made.

CONCLUSIONS AND FUTURE RESEARCH

The effect of continuing advancements in hardware technology is promoting the feasibility of having SPFAs perform many database management functions.

This notion, referred to as the SPFA approach, can serve as a vehicle for increasing the overall DBMS capability in an organization. A DMAD facility, used in conjunction with the methodology defined in this paper, can serve as a specialized model to help introduce SPFAs to an organization via the SPFA approach. This methodology is organized and presented as a set of specific processes. Each of these processes is designed to permit a user to tailor the development of an SPFA to a specific application.

A set of tools/components to perform specific procedures for this methodology are also included in the proposed environment of the DMAD facility.

The extensive use of this methodology can also be directed toward examining critical tradeoff issues for defining a proper hardware/software mix in an overall system for a specific application. The methodology, used in this fashion, can serve as a vehicle to help choose which functions should migrate from software to hardware from an overall system architecture view.

In order to use the methodology in system optimization, further research is needed to expand use of specific optimization methods that can be used to formulate a direct relationship between SPFAs and sets of user requirements. In order to assume this role, modeling techniques may be added as procedures to follow each of the development processes of the generalized methodology. These procedures include several evaluation techniques, among them mathematical modeling and simulation. Some specific techniques that can be used in a hardware/software system tradeoff have been proposed by Vemuri.[31] Further work is needed to expand the detailed use of these techniques.

REFERENCES

1. Banerjee, F., D. Hsiao, and K. Kannar. "DBC—A Database Computer for Very Large Databases." *IEEE Transactions on Computers,* C-28 (1979), pp. 414–429.
2. Canaday, R. H., R. D. Harrison, E. L. Ivie, J. L. Ryder, and L. A. Wher. "A Back-end Computer for Data Base Management." *Communications of ACM,* 17 (1974), pp. 575–582.
3. Capraro, G. R. "A Data Base Management Modeling Technique and Special Function Hardware." Ph.D. Dissertation, Syracuse University, February 1978.
4. Cavano, J. P., and J. A. McCall. "A Framework for the Measurement of Software Quality." *Proceedings of Software Quality Assurance Workshop,* San Diego, California, November 1978. New York: ACM, 1979, pp. 133–140.
5. Champine, G. A. "Current Trends in Data Base Systems." *Computer,* 12 (1979), pp. 27–41.
6. Copeland, G. P., G. J. Lipovski, and S. Y. W. Su. "The Architecture of CASSM: A Cellular System for Non-Numeric Processing." *Proceedings of First Annual Symposium on Computer Architecture,* December 1973, New York: ACM, 1974, pp. 121–128.
7. Crocker, S. D. "State Deltas: A Formalism for Representing Segments of Computation." Ph.D. Dissertation; Information Systems Inc. Research Report ISI, Marina del Rey, California, October 1977.
8. DeFiore, C. R., and P. B. Berra. "A Quantitative Analysis of the Utilization of Associative Memories in Data Base Management." *IEEE Transactions,* 23 (1974), pp. 121–123.
9. DeWitt, D. J. "Direct—A Multiprocessor Organization for Supporting Relational Data Base Management Systems." *IEEE Transactions on Computers,* C-28 (1979), pp. 395–406.
10. Hollaar, L. A. "An Architecture for Efficient Combining of Linearly Ordered Lists." *Second Workshop on Computer Architecture for Non-Numeric Processing,* Gainesville, Florida, 1976. New York: ACM, 1976.
11. Hollaar, L. A., and W. E. Stellhorn. "A Specialized Architecture for Textual Information Retrieval." Technical Report UNIVCDCS-R-74-637, Department of Computer Science, University of Illinois, 1975.
12. Hollaar, L. A. "A Design for a List Merging Network." *IEEE Transactions on Computers,* C-28 (1979), pp. 406–413.
13. Hsiao, D. K., and S. E. Madnick. "Data Base Machine Architecture in the Context of Information Technology Evaluation," *Proc. Third Int. Conference on Very Large Data Bases,* ACM, NY, 1977.
14. Karlowsky, I., H. O. Leilich, and H. C. H. Ziedler. "Content Addressing in Data Bases by Special Peripheral Hardware: A Proposal Called "Search Processor." *Infomatch Fahrberichte* 4, Computer Architecture Workshop of the Gesellschaft Für Infomatch, Erlangen, May 1975, pp. 113–131.
15. Koehr, G. I., J. T. Connolly, P. P. Rhymer, B. L. Girken, and E. V. Sahr. *Data Management Systems Catalog.* Mitre Technical Report 139, Mitre Corp., Bedford, Massachusetts, January 1973.
16. Liuzzi, R. A. "The Specification of a Data Base Machine Architecture Development Facility and a Methodology for Developing Special Purpose Function Architecture." Ph.D. Dissertation, Syracuse University, April 1980.
17. Liuzzi, R. A., and P. B. Berra. "A Data Base Machine Design and Evaluation Facility." *Proceedings of IEEE-CS Comp Sac 78, Computer Software and Applications Conference,* November 1978, Chicago, Illinois. Piscataway, New Jersey: IEEE, 1979, pp. 716–721.
18. Madnick, S. E. "Infoplex: A New Concept in Data Base Management Technology." *Proceedings of the Third International Conference on Very Large Data Bases,* October 1978. Piscataway, New Jersey: IEEE 1978.
19. Maller, V. A. I. "The Content Addressable File Store—CAFS," *ICL Technical Journal (ICL Research and Advanced Development Centre).* (1979). International Computer Ltd., United Kingdom.
20. Molder, R., "An Implementation of a Data Base Management System on an Associative Processor," *AFIPS, Proceedings of the National Computer Conference* (Vol. 42), 1974, pp. 171–176.
21. Mukhopadhyay, A. "Hardware Algorithms for Non-Numeric Computation." *IEEE Transactions on Computers,* C-28 (1979), pp. 384–394.
22. Oliver, E. J. "RELACS, An Associative Computer Architecture to Support a Relational Data Model." Ph.D. Dissertation, Syracuse University, June 1979.
23. Roberts, D. C. "A Special Computer Architecture for High-Speed Text Searching." *Second Workshop on Computer for Non-Numeric Processing,* Gainesville, Florida, 1976. New York: ACM, 1976.

24. Schuster, S. A., E. A. Ozkarahan, and K. C. Smith. "A Virtual Memory System for a Relational Associative Processor." Technical Report CSRG-64, University of Toronto, December 1976.
25. Schuster, S. A., H. B. Nguyen, E. A. Ozkarahan, and K. C. Smith. "RAP.2—An Associative Processor for Data Bases and Its Applications." *IEEE Transactions on Computers,* C-26 (1979), pp. 446–458.
26. Singhania, A. K., and P. B. Berra. "A Multiple Associative Organization for Pipelining a Directory to a Very Large Data Base." *Spring Computer Conference 76 Digest of Papers,* 1976, pp. 109–112.
27. Stellhorn, W. E. "An Inverted File Processor for Information Retrieval." *IEEE Transactions on Computers,* C-26 (1977), pp. 1258–1267.
28. Stillman, N. J., C. DeFiore, and P. B. Berra. "Associative Processing of Line Drawings," *AFIPS Conference Proceedings, Second Joint Computer Conference* (Volume 38), 1971, pp. 557–562.
29. Su, S. Y. W., L. H. Nguyen, A. Eman, and G. J. Lipovski. "The Architectural Features and Implementation Techniques of the Multicell CASSM." *IEEE Transactions on Computers,* (1979), pp. 430–445.
30. TRW Defense and Space Group. *Advanced SMITE Reference Manual.* Contract F30602-78-C-0016, CDRL 007, RADC-TR-80-66, TRW, Redondo Beach, California, February 1980.
31. Vemuri, V., R. A. Liuzzi, J. P. Cavano, and P. B. Berra. "Issues in the Performance Evaluation of Data Base Machine Designs." *1980 Computer Architecture for Non-Numeric Processing Conference,* Asilomar Conference Center, March 1980. Piscataway, New Jersey: ACM, 1980.

Applications of SIMD computers in signal processing

by LAXMI N. BHUYAN and DHARMA P. AGRAWAL
Wayne State University
Detroit, Michigan

ABSTRACT

This paper analyzes in detail how far the proposed Single Instruction Multiple Data (SIMD) computers with interconnection networks are applicable in the signal processing area. Decimation in the time radix-2 fast Fourier transform (FFT) algorithm is considered here for implementation in a multiprocessor system with shared bus and an SIMD computer with interconnection network.

Results are derived for data allocation, interprocessor communication, approximate computation time, speedup, and cost effectiveness for an N-point FFT with any P available processors. Further generalization is obtained for a radix-r FFT algorithm. $N \times N$ point, two-dimensional discrete Fourier transform (DFT) implementation is also considered, with one or more rows of input matrix allocated to each processor.

Various curves are plotted and a comparison in performance is carried out between a shared-bus multiprocessor and SIMD computer with interconnection network. It is shown that the latter gives much higher speedup for $P > 16$ and is more cost-effective even with the high cost of switches. N, P and r, considered here, are all powers of 2.

I. INTRODUCTION

There is a growing interest in the area of parallel processing, and it is worthwhile investigating how far the proposed parallel systems are suitable for different applications. A Single Instruction Multiple Data (SIMD) type of computer usually consists of a single control unit and a number of processing elements (PEs) connected through an interconnection network. The control unit broadcasts each instruction, and those are executed by the active PEs. The interconnection network makes possible simultaneous transfer between the PEs. In this paper the advantage of using such parallel systems in the area of signal processing will be studied.

Real-time signal processing is a potential field of application of parallel computers because of the time limitation in processing data. Fast Fourier transform (FFT) forms the core of signal processing, and hence its implementation will be studied in detail. In a highly parallel algorithm like the fast Fourier transform (FFT), the computation time in various organizations of P processors is almost the same. The communication overhead due to interprocessor data transfer is extremely important and decides the actual performance of an algorithm on a certain multiprocessor architecture.[2]

Although considerable work has been done in the design of special-purpose FFT processors, very few researchers have studied the performance of the FFT algorithm on a general-purpose multiprocessor system. Among them Bergland's algorithms[3] on the PEPE type of computer and Wallach's analysis[8] on the Alternating Sequential Parallel (ASP) computer are noteworthy.

The FFT, as it is, is a highly parallel algorithm; and there seems to be no need for exploiting further parallelism in it. Siegel et al.[4] developed Single Instruction Multiple Data (SIMD) algorithms for both one- and two-dimensional discrete Fourier transforms (DFT) using an interconnection network. They presented decimation in frequency algorithms for implementation in $N/2$ and $N/4$ number of processors without any analysis. Though these types of computers are yet to be commercially available, much research in the area indicates their potential advantage in various types of applications. The Shuffle Exchange network of Stone[5] and the indirect binary n cube network of Pease[6] are very suitable for FFT implementation.

In this paper, decimation-in-time FFT algorithms are considered. The input data and number of butterfly computations are divided equally between the P-available processors, and the amount of time spent during interprocessor communication has been worked out. The performance of an algorithm in a computer depends heavily on the machine constants. Under a few basic assumptions, expressions for speedup and cost effectiveness are worked out for a multiprocessor with shared bus and SIMD computer with Pease's indirect binary n cube network. It has been assumed throughout that the individual processors take care of data allotment in the proper location of their local memory, once the data are available to them.

II. RADIX-2 FFT COMPUTATION

In a decimation in time N point radix-2 FFT algorithm, $n = \log_2 N$ stages of computation is required with $N/2$ butterfly computations at each stage. With P number of processors, $N/2P$ butterfly computations are carried out in each processor per stage. As an example, partitioning of a 16-point FFT with four processors is shown in Figure 1. The algorithm works as follows:

1. Each processor computes $N/2P$ butterflies per stage until $\log N/2P$ stages.
2. Processor i sends $N/2P$ data items to processor j. $1 \leqq i, j \leqq P$.
3. Each processor computes $N/2P$ butterflies.

The process is continued until n stages are completed. It will be assumed throughout this paper that the data allocation at the proper location in a local memory is exclusively the job of the local processing element, and hence does not add to the communication time. The following definitions are needed.

Digit reversal of a number $0 \leqq x \leqq N-1$ in radix r is given by

$$\begin{aligned}\rho_r(x) &= \rho_r(x_{n_r-1}x_{n_r-2}\ldots x_o)\\ &= (x_o x_1 \ldots x_{n_r-1}),\\ &\quad x_i \in \{0, 1, 2, \ldots, r-1\} \text{ and } n_r = \log_r N\end{aligned}$$

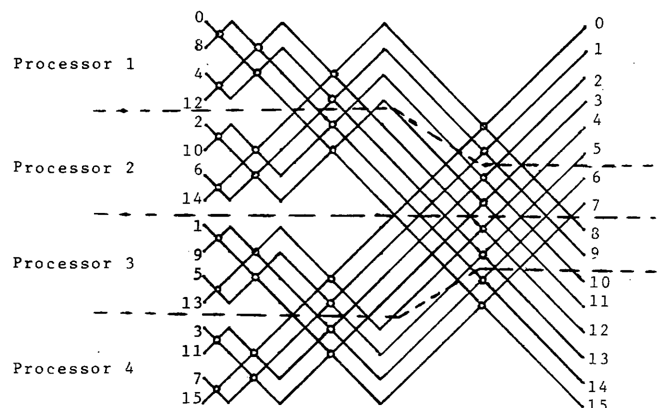


Figure 1—16-point radix-2 FFT computation in 4 processors

Bit reversal is a special case of digit reversal where $r = 2$.

$$\rho_2(x) = \rho(x) = \rho(x_{n-1}x_{n-2}\ldots x_o) = (x_o x_1 \ldots x_{n-1}),\quad x_i \,\varepsilon\, \{0, 1\}$$

$$\textit{Speedup}\ \sigma = \frac{t}{T}$$

where
t = time taken for FFT implementation in a single processor
T = time taken for FFT implementation in P processors

$$\textit{Cost efficiency}\ \xi = \frac{\sigma}{CP}$$

where C = a cost factor dependent on the architecture in consideration.

For a multiprocessor with shared bus, C will be assumed to be unity.

In a decimation in time algorithm, the inputs are bit reversed and the outputs are ordered. We assume that the input data x are numbered from 0 to $N-1$; the processors are numbered from 1 to P and $P = 2^m$. The following general results are obtained.

1. Number of butterfly computations per processor per stage $= N/2P$.
   Assuming each butterfly computation takes one unit of time, computation time per stage $= N/2P$, since the rest of the butterflies in that stage are also simultaneously computed by other processors. Hence, for n stages, total time of butterfly computation $= n(N/2P)$ units.
2. For an N-point FFT with P available processors, number of data per processor $= N/P$. The input data are bit reversed and the output data are ordered. Hence, the ith processor will contain input data $\rho\left[\frac{N}{P}(i-1)\right.$ to $\left.\left(\frac{N}{P}i-1\right)\right]$ and give output data $\left[\frac{N}{P}(i-1)\right.$ to $\left(\frac{N}{2P}i-1\right)$ and $\frac{N}{P}(i-1)+\frac{N}{2}$ to $\left.\left(\frac{N}{2P}i-1\right)+\frac{N}{2}\right]$. $0 \leqq$ input data x, output data $X \leqq N-1$ and $1 \leqq i \leqq P$.
3. Each processor contains $N/2P$ butterflies. Transfer is needed only after N/P point FFTs are calculated internally. Then a P-point FFT between the processors yields the result. After $(n-m)$ stages, the processors are grouped with a difference of 2, 4, 8, 16, etc., until n stages are completed, so the difference grows as a power of 2.
   The processors i and j are exclusively involved in data transfer for kth stage of computation. The difference between i and j with respect to k is given by

$$\begin{aligned} |i-j| &= 0 && \text{for } 1 \leqq k \leqq (n-m) \\ &= \frac{P}{N}2^{(k-1)} && \text{for } (n-m) < k \leqq n \end{aligned}$$

III. ANALYSIS FOR RADIX-2 FFT

A. Multiprocessor With Shared Bus

The organization of this type of system is shown in Figure 2. In addition to the main memory, each processor has its local memory. Interprocessor communication is achieved by first sending the data to the main memory and transferring them again to another processor from the main memory. This is achieved under a central control. Hence, a single data transfer between processors i and j will involve two data transfer times τ. As mentioned, in an FFT calculation no data transfer is necessary for $k \leqq (n-m)$. For stages $k > (n-m)$, each processor i keeps one item of data out of each butterfly for computation in the next stage and transfers the other data to processor j. With $N/2P$ number of butterflies per processor, these data are transferred sequentially over the bus for all $1 \leqq i, j \leqq P$. A time 2τ is necessary for transfer of a single item of data. Each processor takes $N/P\ \tau$ time per stage. For P processors and m stages, time consumed $= mN\tau$.

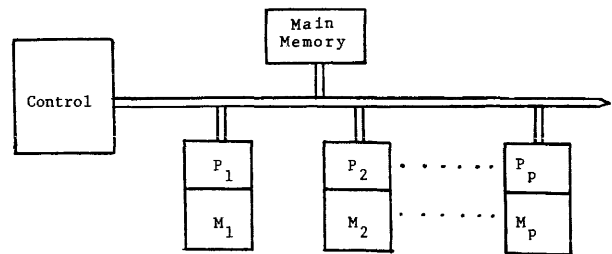


Figure 2—A multiprocessor with shared bus

Speedup and cost efficiency

Let B = time for calculating a single butterfly.

Time taken on a uniprocessor $= t = n \cdot \frac{N}{2} \cdot B$.

Butterfly computation time in P processors $= n \cdot \frac{N}{2P} \cdot B$.

Approximate transfer time needed $= m \cdot N \cdot \tau$.

Hence, time taken for FFT $= T_1 = n \cdot \frac{N}{2P} \cdot B + mN\tau$.

$$\text{Speedup } \sigma_1 = \frac{t}{T_1} = \frac{P}{1 + 2P\left(\frac{m}{n}\right)\left(\frac{\tau}{B}\right)}$$

$$\text{Cost efficiency } \xi_1 = \frac{\sigma_1}{C_1 P} = \frac{1}{1 + 2P\left(\frac{m}{n}\right)\left(\frac{\tau}{B}\right)}$$

Cost factor C_l has been assumed to be unity. It is also assumed that no additional synchronization time is needed for the controller to set up the transfer. Although this assumption may seem unrealistic, it stands well in comparison with the analysis of the interconnection network under different assumptions.

B. SIMD Computer With Interconnection Network

This type of computer will consist of P processors connected through an interconnection switch. An example of eight processors connected through indirect binary n cube network[6] is shown in Figure 3. Each processor can have independent input and output registers for efficient implementation of an algorithm.

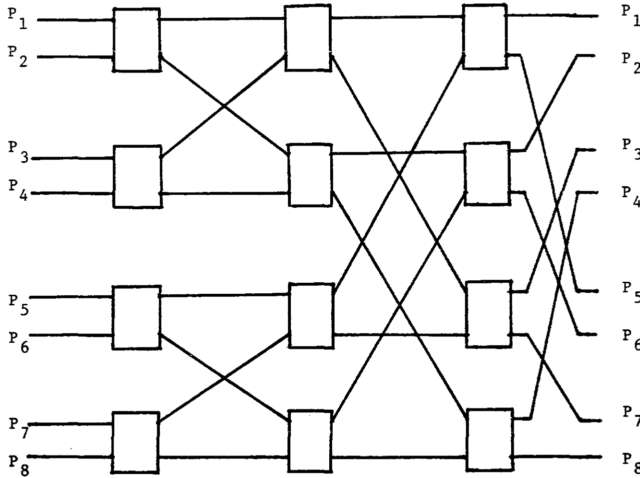


Figure 3—SIMD computer with Indirect binary n cube network

If a processor is expressed in binary as $(p_m p_{m-1} \ldots p_q \ldots p_1)$, then $\text{cube}_q(i) = (p_m p_{m-1} \ldots \bar{p}_q \ldots p_1)$ and $\text{cube}_q(i) \sim i = (00\ldots1\ldots0)$, 1 being at the qth position with weight 2^{q-1}.

The intercommunication involved in FFT calculation is basically a *PM2I* connection with $j = \text{cube}_q(i)$. Hence, $q = (k - n + m)$ for the kth stage of computation.

Again, no data transfer is necessary for $k \leqq (n - m)$. After that, all the P processors are capable of transferring data simultaneously at a single transfer time, once the switch has been set. After each stage of computation, each processor will send $N/2P$ data items. For m stages, the total data transfer time for an N-point FFT implemented on P processors with interconnection network is $m \cdot N/2P$ units.

Speedup and cost efficiency

The control setting of an indirect binary n-cube will require $0(P\log_2 P)$ time, and each data transfer will undergo $0(\log_2 P)$ gate delays. It is assumed here that, once the control switch has been set, $N/2P$ data items are transferred from each processor in $N/2P\tau$ time. The combinational gate delays are neglected.

Even further time can be saved if the controller is allowed to set the switches by a table lookup while the processors are involved in butterfly computation.

Allowing a switching time of $m \cdot P \cdot \tau$,

$$T_2 = n \cdot \frac{N}{2P} \cdot B + m\left(\frac{N}{2P}\tau + mP\tau\right)$$

$$= n \cdot \frac{N}{2P} \cdot B + m \cdot \frac{N}{2P}\tau\left(1 + 2\frac{mP^2}{N}\right)$$

$$\text{speedup } \sigma_2 = \frac{t}{T_2} = \frac{P}{1 + \left(\frac{m}{n}\right)\left(1 + 2\frac{mP^2}{N}\right)\left(\frac{\tau}{B}\right)}$$

$$\text{and cost efficiency } \xi_2 = \frac{\sigma_2}{C_2 P}$$

From the above results, it is clear that the speedup depends heavily on factors (τ/B) and (a/B). These are machine-dependent constants and vary from one computer to other. In Figure 4 we have plotted the speedup for a multiprocessor with shared bus having $P = 16$ for various realistic values of (τ/B). As expected, the speedup reduces with increase in (τ/B). Speedups obtained in different computers with $P = 8$, 16, and 32 are plotted in Figure 5. (τ/B) and (a/B) are assumed to be 0.02 and 0.2 respectively. σ_{20} is the speedup obtained in an SIMD computer with interconnection network when the control switching time is avoided by setting the switches during butterfly computation. The cost effectiveness depends on the cost factors C_1, C_2, and C_3. The exact values of these constants are difficult to predict. For comparison C_1 was assumed to be unity and curves were drawn with probabilistic values of C_2 between 1.2 and 1.5. This showed an overall degradation in cost efficiency with increase in the number of processors. For higher values of N and for higher numbers of processors, the interconnection network proved to be more cost-effective than the other.

IV. RADIX-r FFT COMPUTATION

In this section the results obtained in Section II are extended for a radix-r implementation of N-point FFT with P available processors; r is assumed to be a power of 2 and N and P are powers of r. The algorithm works in exactly the same way.

1. Each processor computes N/rP butterflies per stage till $\log_r N/P$ stages.
2. Processor i sends N/rP data items each to $(r - 1)$ other processors.
3. Each processor computes N/rP butterflies.

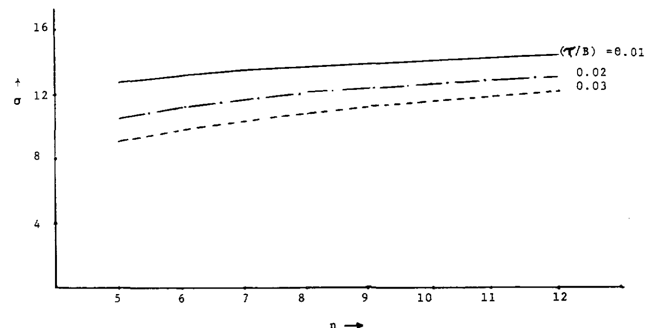


Figure 4—Variation of speedup with (τ/B) on a shared bus organization

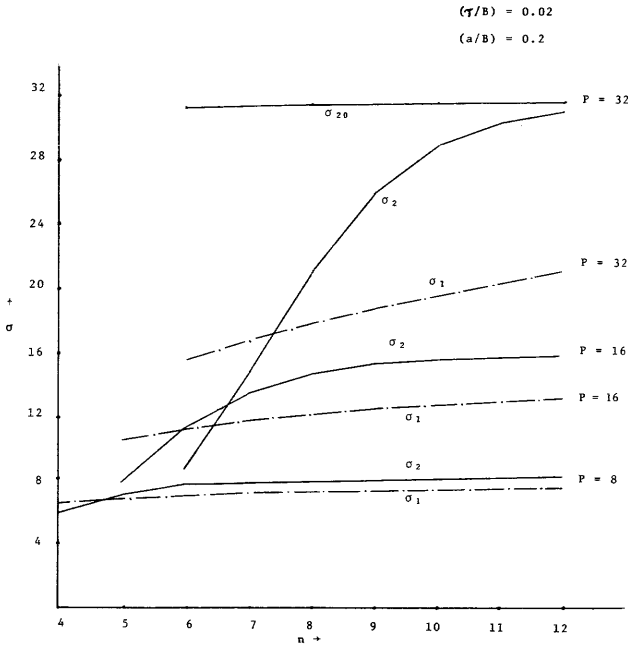


Figure 5—Speedup for radix-2 FFT computation in different organizations
σ_1: Shared bus organization.
σ_2: SIMD computer with interconnection network.
$_{20}$: Interconnection network without control setting time.

The process is continued till $n_r=\log_r N$ stages are complete. The previous results are modified as below.

1. The total butterfly computation time for an N-point radix-r FFT with P processors is $n_r N/rP$ units, $n_r=\log_r N$.
2. The ith processor will contain input data $\rho_r\left[\frac{N}{P}(i-1)\right.$ to $\left.\left(\frac{N}{P}i-1\right)\right]$ and will give output data $\left[\frac{N}{P}(i-1)+\frac{N}{r}\cdot\alpha\right.$ to $\left.\frac{N}{rP}i-1+\frac{N}{r}\cdot\alpha\right]$ for $0\leqq\alpha\leqq r-1$ and $1\leqq i\leqq P$.

   The input data x and output data X range between 0 and $N-1$.
3. In a radix-r FFT algorithm, r number of processors are involved in data transfer for kth stage computation.

$$|i-j|=0 \text{ for } k\leqq n_r-m_r,\ m_r=\log_r P$$
$$=\ell\cdot\frac{P}{N}\cdot r^{(k-1)} \text{ for } n_r-m_r<k\leqq n_r.$$
$$\ell=1,2,\ldots,(r-1)$$

(a) *Multiprocessor with shared bus*

Each processor computes N/rP butterflies per stage. Out of this, one item of data is sent to each $(r-1)$ processor. Again, each data transfer requires 2τ time, so each processor will keep the bus busy for $2(r-1)\cdot N/rP\ \tau$ time. For P processors and m_r stages of data transfer, the total data transfer time for a radix-r N-point FFT implemented on P processors with a shared bus is $2\cdot m_r(r-1)\cdot N/r$ units.

Speedup and cost efficiency

Let B_r=time for a single radix-r butterfly computation.

$$T_{1r}=n_r\cdot\frac{N}{rP}\cdot B_r+2m_r(r-1)\cdot N/r\cdot\tau.$$

Time taken in a uniprocessor $t_r=n_r\cdot N/r\cdot B_r$.

$$\text{Speedup } \sigma_{1r}=\frac{t_r}{T_{r1}}=\frac{P}{1+2P\left(\frac{m_r}{n_r}\right)(r-1)\left(\frac{\tau}{B_r}\right)}$$

$$\zeta_{1r}=\text{cost efficiency}=\frac{\sigma_{1r}}{C_1P}$$

(b) *SIMD computer with interconnection network*

The transfers are effected in a similar manner, except that at each stage the control switch has to be set up $(r-1)$ times, thus causing a degradation in performance.

Cyclic shift within segments type of permutation[10] is necessary before the kth stage of computation in radix-r FFT algorithm. This type of permutation can be easily implemented with an Indirect binary n-cube network.

After computation of (n_r-m_r) stages, $(r-1)$ outputs will be sent to $(r-1)$ processors from each butterfly through $(r-1)$ cyclic shift permutations. However, simultaneous transfer occurs from each processor through the interconnection network. Hence for N/rP butterflies and in m_r stages, the total number of data transfers in a radix-r FFT implemented on P processors with interconnection network is $m_r\cdot(r-1)N/rP$ units.

Speedup and cost efficiency

$$T_{2r}=n_r\cdot\frac{N}{rP}\cdot B_r+m_r(r-1)\left(\frac{N}{rP}\tau+mP\tau\right)$$

$$\text{Speedup } \sigma_{2r}=\frac{P}{1+\left(\frac{m_r}{n_r}\right)(r-1)\left(1+\frac{rmP^2}{N}\right)(\tau/B_r)}$$

and cost efficiency $\zeta_{2r}=\frac{\sigma_{2r}}{c_2P}$.

The speedup obtained for a radix-4 algorithm in $P=16$ and 64 processors is plotted in Figure 6 with (τ/B_r) assumed to be 0.005, i.e., $B_r=4B$. The comparison is shown between a shared bus computer and SIMD computer with interconnection network. As expected, for higher values of N, the interconnection network gives speedup close to ideal.

V. TWO-DIMENSIONAL DFT COMPUTATION

A $2-D$, $N\times N$ discrete Fourier transform (DFT) is given by Siegel et al.:[4]

$$F(u,w)=\sum_{\ell=0}^{N-1}\sum_{m=0}^{N-1}x_{(\ell,m)}W^{u\ell}W^{wm},$$
$$W=e^{-2\pi j/N} \text{ and } 0\leqq u,\ w\leqq N-1.$$

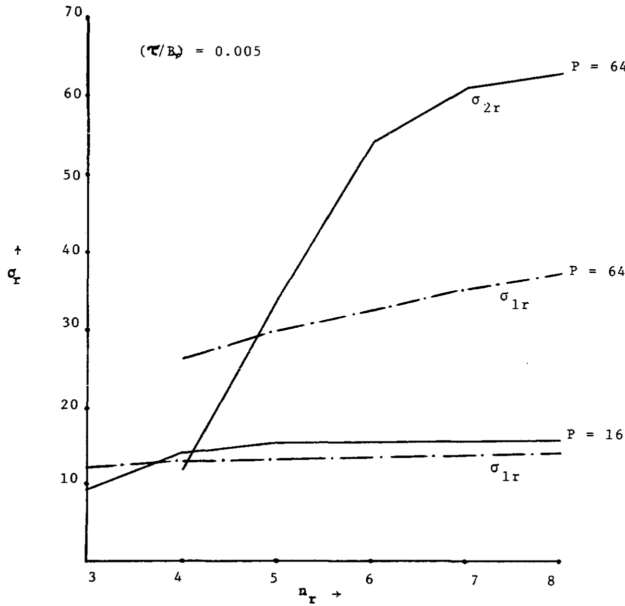


Figure 6—Speedup for radix-4 FFT computation in different organizations
σ_{1_r}: Shared bus organization.
σ_{2_r}: SIMD computer with interconnection network.

This can be decomposed to two one-dimensional DFTs:

$$G(\ell,w) = \sum_{m=0}^{N-1} x_{(\ell,m)} W^{wm},\ 0 \leqq \ell,\ w \leqq N-1$$

and

$$F(u,w) = \sum_{\ell=0}^{N-1} G(\ell,w) W^{u\ell}\ 0 \leqq u,\ w \leqq N-1$$

These one-dimensional DFTs are usually implemented by using FFT techniques. The input data x can be visualized to be arranged in an $N \times N$ matrix. G, also is an $N \times N$ matrix, each row of which is computed by taking a $1-D$ FFT on a row of x. For any available number of P processors, $P = 2^m$ and $\leq N$, one or more rows of x will be allotted to each processor. For computing each column of matrix F, a column of G is necessary so that FFT techniques can be applied.

Unfortunately, G is stored row-wise in the processors. Hence a matrix transpose operation with one or more rows in each processor has to be obtained before another $1-D$ FFT computation can be carried out to yield matrix F. This involves data transfer between the processors, thus affecting the speedup.

The algorithm works as follows:

1. Each processor computes N/P number of N point radix-2, 1-D FFT to yield N/P rows of G.
2. A matrix transpose operation is carried out between the processors.
3. Each processor computes N/P number of N point radix-2, 1-D FFT to yield N/P columns of the result F.

We get the following results:

1. N rows of x will be divided between P processors. Therefore each processor will calculate N/P rows of G. Each row of G takes $n \cdot N/2$ time units for FFT calculation. Therefore each processor will take $n \cdot N^2/2P$ time units. Again, each column of F requires $n \cdot N/2$ units of time and $n \cdot N^2/2P$ time for N/P columns. Hence, total butterfly computation time $= n \cdot N^2/P$ units.
2. Number of rows per processor $= N/P$. The first processor contains rows 0 to $N/P - 1$, second from N/P to $2N/P - 1$, and so on. Hence, the ith processor will contain rows $\ell = N/P\,(i-1)$ to $N/P\,i - 1$ of input data $x_{(\ell,m)}$, for $0 \leqq \ell,\ m \leqq N-1;\ 1 \leqq i \leqq P$.
3. After each processor calculates N/P rows of G, a matrix transpose operation is to be performed.

If matrix $G(\ell,w)$ is partitioned into $P \times P$ square submatrices, for a transpose operation processor r will transfer N^2/P^2 elements of submatrix $G(i,j)$ to processor j; $1 \leqq i,\ j \leqq P$.

It is assumed that the individual processors take care of the internal data arrangement of N^2/P^2 elements in their local memories.

(a) *Multiprocessor with shared bus*

A data transfer between processors i and j will involve a time of 2τ. Each processor i sends $N \cdot N/P - N^2/P^2$ data items to all other processors.

Hence, the total data transfer time for an $N \times N$, $2D$ DFT calculation in P processors with shared bus is $2(N^2/P)(P-1)$ units.

Speedup and cost efficiency

Time taken on a single processor $= t = 2 \cdot N \cdot n \cdot N/2B$
$= n \cdot N^2 \cdot B.$

$$T_4 = n \cdot \frac{N^2}{P} \cdot B + 2 \cdot \frac{N^2}{P}(P-1)\tau.$$

$$\text{Speedup } \sigma_4 = \frac{P}{1 + 2\left(\frac{P-1}{n}\right)(\tau/B)}$$

$$\text{cost efficiency } \zeta_4 = \frac{\sigma_4}{P}.$$

(b) *SIMD computer with interconnection network*

Each processor i will be connected to processors $j = (i+k)$ mod P for $1 \leqq k \leqq P-1$, involving $(P-1)$ stages of data transfer. At each stage N^2/P^2 elements will be transferred from each processor. This is a cyclic shift operation realizable by Indirect binary n cube network through a single pass.

The total data transfer time for an $N \times N$, $2D$ DFT implemented on P processors with interconnection network $N^2/P^2(P-1)$ units.

Speedup and cost efficiency

$$T_5 = n \cdot \frac{N^2}{P} \cdot B + (P-1)\left(\frac{N^2}{P^2}\tau + m \cdot P\tau\right)$$

$$\text{Speedup } \sigma_5 = \frac{P}{1 + \frac{(P-1)}{P}\left(\frac{1}{n} + \frac{m}{n} \cdot \frac{P^3}{N^2}\right)(\tau/B)}$$

and cost efficiency $\zeta_5 = \frac{\sigma_5}{C_2 P}$.

The performance of a multiprocessor with shared bus is compared with respect to interconnection network in Figure 7. For a number of processors greater than 16, the shared bus computer works fairly well compared to the $1-D$ case and is more cost-effective because $C_1 < C_2$.

VI. CONCLUDING REMARKS

The exact performance of a computer is ascertained only after carefully observing its working for many years. In this paper an approximate evaluation of speedup and cost efficiency has been made for FFT implementation in two types of parallel processors. These values depend heavily on machine constants, as shown in Figures 4 and 5. The comparisons are made between a multiprocessor with shared bus and an SIMD computer with interconnection network. For one dimensional radix-2 and radix-4 algorithms, the shared bus computer shows close to ideal performance for $P \leqq 16$. For small values of N, the interconnection network gives very low speedup because of the overhead involved in setting the control switches. However, as N increases, the speedup increases and is close to ideal for large N. At this point it becomes more cost-effective than the shared bus system, even with the high cost of switches. When P, N are very large, the shared bus system is completely unsuitable because of high congestion in the single bus. A similar analysis was also performed for the $2-D$ case. The shared bus system behaves much better than the $1-D$ case. For $P = 32$ in Figure 7 the speedup is very high when compared to Figure 5. If the number of processors available for an $N \times N$, $2-D$ DFT implementation is more than N, completely different results will be obtained, because for each $1-D$ transform also, the interprocessor communication will be necessary.

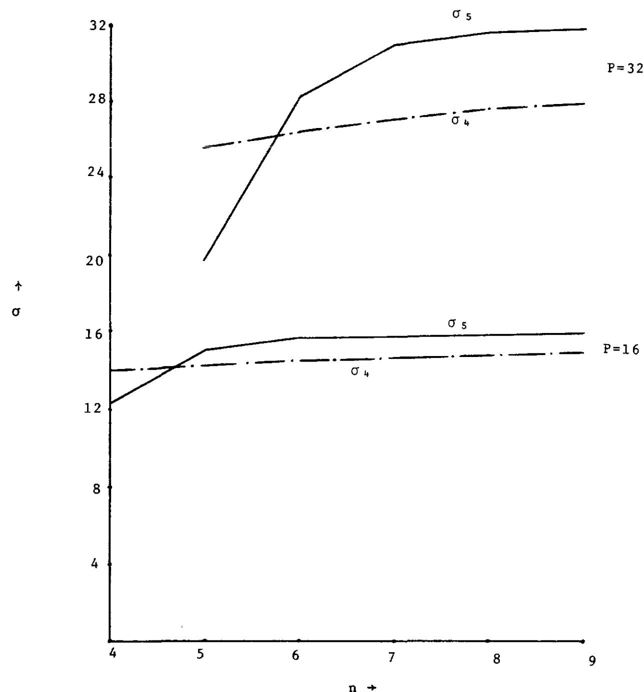


Figure 7—Speedup for 2-D DFT in different organizations
σ_4: Shared bus organization
σ_5: SIMD computer with interconnection network

REFERENCES

1. Aho, A. V., J. E. Hopcroft, and J. D. Ullman. "The Design and Analysis of Computer Algorithms." Reading, Massachusetts: Addison-Wesley, 1976.
2. Lint, B., and T. Agerwala. "Communication Issues in the Design and Analysis of Parallel Algorithms." *IEEE Transactions on Software Engineering,* SE-7 (1981), pp. 174–188.
3. Bergland, G. D. "A Parallel Implementation of the Fast Fourier Transfer Algorithm." *IEEE Transactions on Computers,* C-21 (1972), pp. 366–370.
4. Siegel, L. J., P. T. Muller, and H. J. Siegel. "FFT Algorithm for SIMD Machines." *Proceedings of the 17th Annual Allerton Conference on Comm., Control, and Computing,* University of Illinois, Urbana-Champaign, Oct. 1979, pp. 1006–1014.
5. Stone, H. S. "Parallel Processing with Perfect Shuffle." *IEEE Transactions on Computers,* C-20 (1971), pp. 153–161.
6. Pease, M. C. "The Indirect Binary *n* Cube Microprocessor Array." *IEEE Transactions on Computers,* C-26 (1977), pp. 458–473.
7. Abidi, M. A., and D. P. Agrawal. "On Conflict-Free Permutations in Multistage Interconnection Network." *Journal of Digital Systems,* (Special Issue on Parallel Processing) 4 (1980), pp. 115–134.
8. Wallach, Y., and A. Shimor. "Alternating Sequential Parallel Versions of FFT." *IEEE Transactions on Acoustics, Speech and Signal Processing,* ASSP-28 (1980), pp. 236–242.
9. Rabiner, L. R., and B. Gold. "Theory and Applications of Digital Signal Processing." Englewood Cliffs, New Jersey: Prentice Hall, 1975.
10. Lenfant, J. "Parallel Permutations of Data: A Benes Network Control Algorithm for Frequently Used Permutation." *IEEE Transactions on Computers,* E-27, (1978), pp. 637–647.

A list-processing-oriented data flow machine architecture

by MAKOTO AMAMIYA, RYUZO HASEGAWA, OSAMU NAKAMURA, and HIROHIDE MIKAMI
Musashino Electrical Communication Laboratory, N.T.T.
Tokyo, Japan

ABSTRACT

This paper analyzes some issues concerning list processing under a data flow control environment from the viewpoint of parallelism and also presents a new type of list-processing-oriented data flow machine, based on an association memory and logic-in-memory.

The mechanism of partial execution in each function is shown by example to be effective in exploiting the parallelism in list processing. The lenient cons mechanism is shown to exploit maximally parallelism among activated functions.

1. INTRODUCTION

A data flow machine, whose basic idea was offered by J. B. Dennis[1] and for which several research efforts are being pursued at several places in the world,[2–6] is a very attractive concept as a future computer architecture, from the following viewpoints:

1. A data flow machine exploits the parallelism inherent in problems.
2. Recent noteworthy advances in VLSI technology have been made. A data flow machine makes effective use of numerous VLSI devices and makes possible the implementation of a distributed control mechanism.
3. Functional programming will become increasingly important to the improvement of software productivity. A data flow machine effectively executes programs written in a functional language.
4. Nondeterministic execution[7] will become an important mechanism in future computer systems. A data flow machine is expected to execute nondeterministic programs effectively because of its parallelism.

However, many problems remain to be solved in order to achieve an actual data flow machine in a real environment. Especially when considering Items 3 and 4 just cited, it is necessary to clarify the data flow machine's applicability to nonnumerical problems.

This paper discusses list processing, which is typical of nonnumerical data processing, on a data flow machine, keeping the Lisp data structure and operations in mind. The main reasons why Lisp was considered are that Lisp has a simple and transparent data structure and that it contains the basic problems in structured data manipulation.

First, parallelism in list processing is discussed, and it is pointed out that this can be achieved by parallel evaluation of function arguments and partial execution of the function body. Then it is shown that parallelism increases dramatically with introduction of a lenient cons concept into the data flow execution control. Next, list-processing-oriented data flow machine architecture and structure memory construction methods are presented. Finally, a garbage collection algorithm, based on the reference count method, is discussed.

All programs throughout this paper are described in VALID[8] language, which is designed as a high-level programming language for the data flow machine presented in this paper.

2. LIST PROCESSING UNDER A DATA FLOW CONTROL ENVIRONMENT

The noteworthy data flow execution control effects are as follows:

1. It exploits the maximal parallelism inherent in a given program both on a low level (primitive operation level) and on a high level (function activation level).
2. It effectively executes programs constructed on the basis of the concept of functional programming, which has no notion of program variables and side effects (i.e., rewriting the global variables).

The parallelism of the primitive operation level is achieved by the data-driven control principle; that is, each operation is initiated without attention to other operations when all of its operands have arrived. Function-activation-level parallelism is obtained by the partial evaluation mechanism:

1. Each argument of a function is evaluated concurrently.
2. The execution of a function is initiated when one of the arguments of the function is evaluated, and the caller function resumes its execution when one of the return values is obtained in the invoked function execution.

In this section these parallel execution mechanisms are examined through several examples.

2.1. Parallel Evaluation of Arguments

Programs written in VALID are transformed to equivalent pure functional representation, i.e., the form of prefix notation, and equally translated to data flow graphs. For instance, Program1, which reverses a given list in each level, is translated by the VALID compiler into the data flow graph shown in Figure 1. Block1 in Program1 is equivalently represented in the prefix notation

fulrev(cdr(x), cons(fulrev(car(x), nil), y)).

In this expression the two arguments cdr(x) and cons(...) for the function fulrev are evaluated in parallel; and before evaluating the argument cons(...), its two arguments fulrev(...) and y are evaluated in parallel, and so on. Thus, the evaluation of a function, in general, proceeds from the inner to the outer (i.e., innermost evaluation). This results in highly parallel evaluation of the innermost arguments. In other words, each evaluation is independent of the other evaluations under the condition that the evaluation is initiated only when all values of arguments are obtained (which is called data-driven control).

Program1—Mirror image of tree
fulrev: <u>function</u> (x,y) <u>return</u> (list)

```
= case
  null(x) →y;
  atom(x)→x;
  others  →clause
          ┌ u = cdr(x);
          │ v = fulrev(car(x),nil);
  block1  │ w = cons(v,y);
          └ return fulrev(u,w)
            end
end
```

2.2. Partial Execution of Function Body

The parallelism, based on the parallel evaluation of arguments for each function, is limited because the nesting of arguments is limited in source text. This restriction on parallelism, however, can be overcome by executing the function body partially.

If the data-driven control principle is applied to the function activation, as in the case of primitive operations, every function is activated only after all its arguments are evaluated. In this case, time is wasted unnecessarily in each function activation through waiting for the completion of all its argument evaluations. However, if each value is passed into the function body immediately when it is evaluated, and the function body execution proceeds partially every time the value is passed in, efficient execution can be obtained, because the unnecessary waiting is cut out at function activation time.

The function activation and argument-passing mechanism for the partial function execution is implemented as shown in Figure 2. The data flow graph in Figure 2(c) represents the activation control for the function

$$[y1, y2, \ldots, yn] = f(x1, x2, \ldots xm).$$

The call node, which creates a new environment for the activated function, is initiated by the "or" gating nodes, when one of the tokens (values) has arrived. Here, the call node creates the body first if the body does not exist. Otherwise, it creates only an instantiation name. The "or" gate implementation uses a t/f switch, as shown in Figure 2(b).

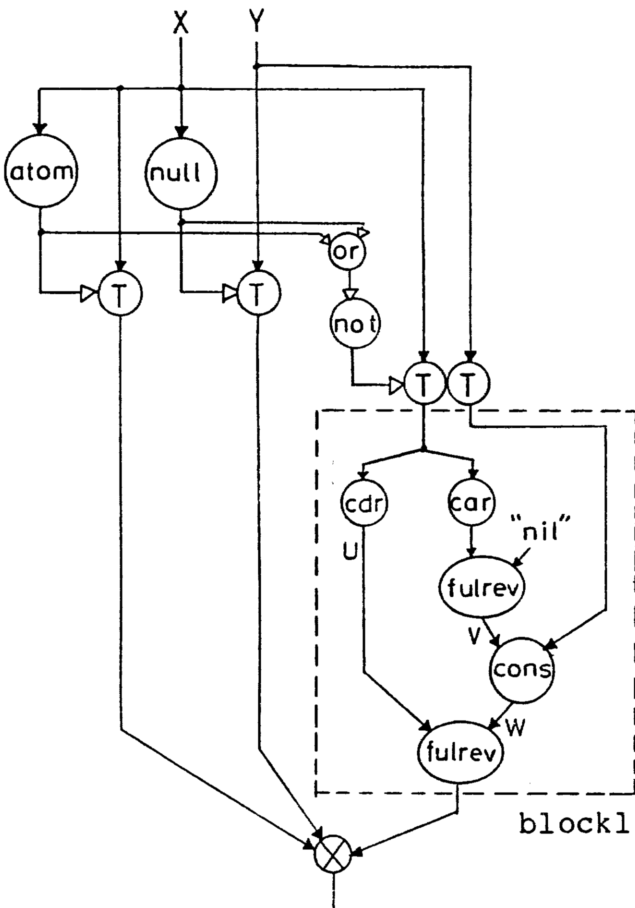
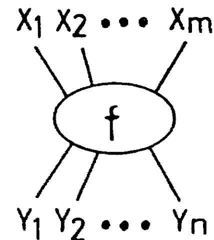
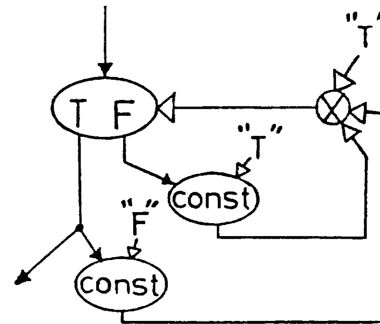


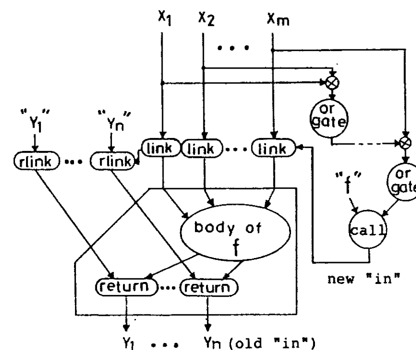
Figure 1—Data flow graph for fulrev



(a) Function invocation node



(b) Or gate implementation



(c) Function activation control

Figure 2—Function activation mechanism

When the new environment is created and the body is ready to run, the token "in" (instantiation name of the activated function) is sent to link nodes and rlink nodes. Each link node passes each argument value x1, x2, . . . , xm to the body of the activated function every time each value has arrived. Each rlink node passes information regarding the place where the return value is sent. These bits of information y1′, y2′, . . . , yn′, each of which is determined at compilation time corresponding to y1, y2, . . . , yn, are attached to each return value to identify its destination. As each return value is passed back to the calling function as soon as it is generated, the calling function can resume and proceed with the execution partially every time the return value is passed back from the called function. Here each function is permitted to return multiple values (i.e., the tuple of values) under the data flow environment.

2.3. Lenient Cons and Parallelism by Pipelined Processing

Although the partial execution of function yields higher parallelism, it is not sufficient for maximally exploiting the parallelism inherent in the given program.

In program2, for instance, the function partition in sort body divides a list into three lists, y1,y2,y3, each of which contains elements less than, equal to, and greater than the first element. As the sort and append are activated immediately after each of y1,y2,y3 is generated, it is expected that the maximal parallelism among functions is obtained. However, parallelism by partial execution of the function body does not work well for reducing the execution time in the order, since the time spent to sort the list of length n is proportional to the square of n in the worst case. (Though it is proportional to n in the best case.) The reason is that since each of the values y1, y2, and y3 is not returned until the append operation is completed in the partition body, the execution of the sort function, which uses those values, must wait until they are returned, and the waiting time is proportional to the length of the list data made by the append operation.

```
program2—Quicksort program
sort: function (x) return (list)
= if x=nil then x
           else clause
              y = list(car(x));
                [y1,y2,y3] = partition(cdr(x),y);
                return
                append(sort(y1),append(y2,sort(y3)))
              end;
partition: function (x,y) return(list,list,list)
= if x=nil then (nil,y,nil)
           else clause
              [w1,w2,w3] = partition(cdr(x),y);
              x1 = car(x); y1 = car(y);
              return
                case
                  x1=y1→(w1,append(list(x1),w2),w3);
                  x1<y1→(append(list(x1),w1),w2,w3);
                  x1>y1→(w1,w2,append(list(x1),w3))
                end
              end;
```

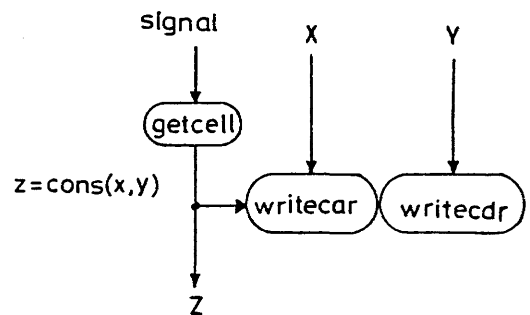
If the former parts of the list, which are partially generated, are returned in advance during the period when the latter parts are appended, the execution which uses the former parts of the list can proceed. Thus the producer and the consumer executions overlap each other. As the append is the repeated application of cons, as Program3 shows, this problem can be solved by introducing leniency into the cons operation.

```
Program3—append
append: function (x y) return (list)
= if x=nil then y
           else cons(car(x),append(cdr(x),y))
```

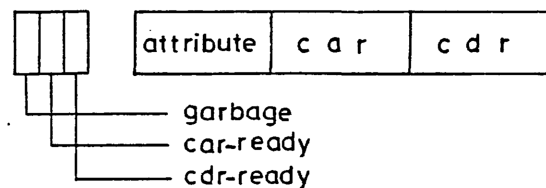
Lenient cons, which is slightly different from the idea of "suspended cons,"[9] means the following: For the operation of cons(x,y), the cons operator creates a new cell and returns its address as a value in advance before its operand x or y arrives. Then the x and y values are written in the car and cdr field of the cell, respectively, when each of them has arrived at the cons node.

In the implementation the cons operator is decomposed into three primitive operators, getcell, writecar and writecdr, as shown in Figure 3. The getcell node is initiated on the arrival of a signal token, which is delivered when the new environment surrounding the cons operation is created. The getcell operator creates a new cell and sends its address to the writecar node, the writecdr node, and the nodes waiting for that cons value.

Each memory cell has, in addition to the garbage tag, the car-ready tag and the cdr-ready tag, each of which controls read accesses to the car field and the cdr field. The getcell operator resets both ready tags to inhibit read accesses. The



(a) Cons mechanism



(b) Data cell structure

Figure 3—Lenient cons implementation

writecar (or writecdr) operator writes the value x (or y) to the car field (or cdr field), and sets the ready tag to allow read accesses to the field.

Lenient cons has a great effect in list processing. It naturally implements the stream processing feature, in which each list item is processed as a stream[4,10] for programs that are normally written according to the list processing concept, without the notion of stream.

3. DATA FLOW MACHINE ARCHITECTURE

The data flow machine is composed of five components: control modules (CMs), an inter-CM communication network (CN), structure memories (SMs), an arbitration network (AN), and a distribution network (DN), as shown in Figure 4. The CM, which is the kernel of data flow execution control, consists of a memory for data flow machine instructions and the enabled instruction fetch mechanism. The CN connects CMs with each other. The SMs store structured data such as list data. The AN and DN connect CMs and SMs.

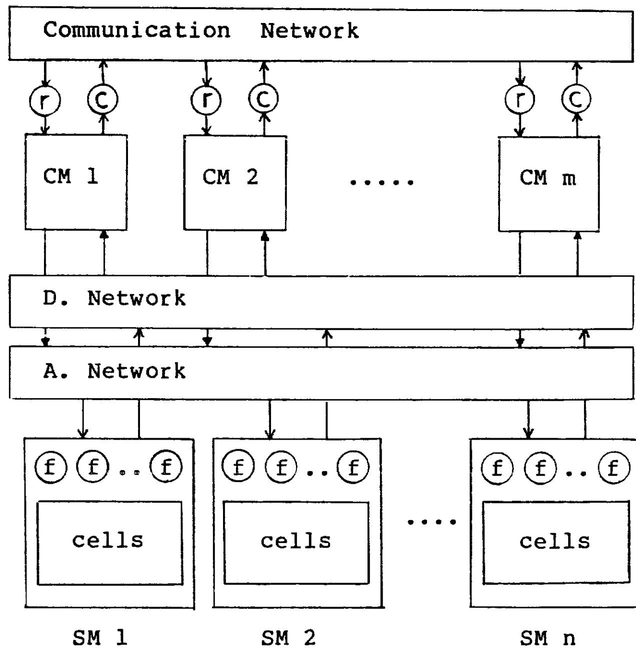


Figure 4—Data flow machine organization

The characteristics of this machine architecture, which is mainly based on the associative memory concept, are as follows:

1. Effective memory utilization can be achieved as a result of dividing the CM memory into instruction memory (IM) and operand memory (OM). The IM, which is a read-only associative memory, contains data flow program (i.e., function body). Here, destination instructions that await a result value are retrieved associatively. The OM acts as a buffer for arriving operands.
2. As function bodies of a program are distributed in each of the CMs, and each CM controls the execution of each function body concurrently, parallelism is achieved among CMs. The call/return parameters among functions are passed through the CN, which logically realizes dynamic tree structure.
3. Operation units are embedded in structure memory. The structure memory is composed of a number of banks, in each of which structured data operation units are equipped.
4. The AN and DN provide paths between CMs and SMs. The AN decodes the operand address in the instruction packet and sends the packet to the addressed SM bank. The DN accepts the result packet, which contains the destination CM address, from SM, and delivers it to the specified CM. The AN and DN are constructed using routing network technique.

This data flow machine architecture can exploit high parallelism due to the concurrent executions among IMs and the pipelined processing between IM and SM.

4. EXECUTION CONTROL

The CM memory which contains data flow machine codes is composed of an IM and an OM, as mentioned before. The IM and OM organization is shown in Figure 5.

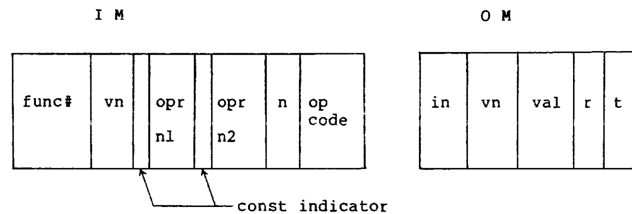


Figure 5—IM and OM field organization

Each memory cell of the IM consists of several fields, function name field (func#), value name field (vn), first operand name field (oprn1), second operand name field (oprn2), operand number field (n), and operation code field (opcode). The OM consists of five fields, instantiation name field (in), value name field (vn), operand value field (val), first/second operand indicator (r), and garbage tag (t). The instantiation name is assigned to a result value so as to share the function body.

The mechanism to deliver result value and fetch an enabled instruction is shown in Figure 6.

When a result packet has arrived at the IM, the func# and oprn1 or oprn2 are examined associatively, using the key ⟨func#, vn⟩, both of which are extracted from the result packet as a search key. If the matched instruction is a one-operand type, an instruction packet is immediately constructed from the matched instruction code and the result value contained in the result packet and sent to the AN.

If the matched instruction is a two-operand type, on the other hand, the in and vn field in the OM are examined for

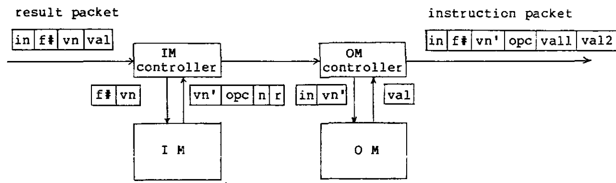


Figure 6—Executive control mechanism

matching associatively against the key ⟨in⟩ in the result packet and the key ⟨vn⟩ extracted from the matched instruction at IM.

If an OM cell is matched, which means one of the two operands has arrived already, the matched data are read out from OM. Then a two-operand type of instruction packet is constructed along with the operand value contained in the result packet and sent to the AN.

If no OM cells are matched, the garbage tag field is accessed associatively to find a free cell. Then, the ⟨in, vn, val⟩ in the result packet and tag r, which indicates whether the val is the first operand or the second operand, are written into the cell taken out.

5. STRUCTURE MEMORY

The method of structured data manipulation is an important problem in the data flow machine architecture.[4] In this section, structure memory design philosophy and its construction method are described from the viewpoint of parallel list processing. In this data flow machine, list structured data are stored in the structure memory, and their pointers to each entry flow in the machine as data tokens.

5.1. Primitive Operation in List Processing and Memory Function

Pure Lisp primitive operations that have no side effect are considered as a basis for structured data manipulation. Among the five primitive operations (cons, car, cdr, atom, and eq), only the cons operation creates a new data cell and writes car and cdr pointer into the cell. Once the value is written into the cell, its contents are never modified. As other operations only refer to the cell, and as programs composed of these five functions have no side effects, the new cell may be created at any location.

List processing is regarded as memory operations which mainly contain readout operations. How to execute the memory operation effectively is a key problem. Memory contention and side effects are serious for exploiting the parallelism in list processing. The parallel execution among memory operations is obtained by preserving functionality, as in pure Lisp.

The data-driven control makes possible the pipelined processing between execution control and memory operation. If the pipe capacity is large enough, execution control is not affected by memory access overhead. Therefore, uninterrupted access to memory cells is possible.

As a new cell may be created by cons at any location, the problem of memory contention can be solved by dividing the structure memory into many banks. In addition, parallelism among memory operations is obtained by providing an operation unit for each memory cell. This idea results in a logic-in-memory concept. When the tradeoff between parallelism and cost is considered, it can be decided whether to embed the operation in a memory device.

5.2. Garbage Collection

As many data elements are copied in the course of the side-effect free data manipulation, how to use structure memory cells effectively is an important problem. Although mark-scan methods are generally used as a garbage collection method in a conventional machine, a reference count method is adopted here, for the following reasons:

1. Since pointers to list data entries are scattered in various parts of the machine, such as instruction memory units, operation units and networks,[10,11] it is very difficult to extract the active cell without suspending execution.
2. As list manipulations have no side effect, no circular lists exist.

In the reference count method, each structure memory cell or memory block has a reference counter field which is updated every time operations, such as car, cdr, etc., are performed. Reference count handling overhead will be serious if the reference count is updated in not only primitive operations but also in T/F switch and function linkage operations. However, this problem can be solved by reducing the reference count update frequency. The method adopted here makes use of VALID language features, that is, (1) block structure and locality of value name, (2) uniqueness of the value name definition (single assignment rule). The reference count management explicitly updates the reference number of the cell by performing the increment and decrement operations. It is not necessary to update the reference number of cells referenced in a block every time operations are performed. Instead, the reference number of the cell which is newly denoted in a block is incremented when the block is opened and decremented when the block is closed.

5.3. Structure Memory Organization

Unlike numerical processing, which handles regular data structures such as vectors and arrays, it cannot be expected that manipulating list-structured data yields locality of access to each list item, since many functions refer to sublists or superlists of a list which is produced by some function, and the sublists and superlists are produced variously during the execution of many functions. In such a case, whether to achieve the locality of access in each function or to distribute access without copying sublist is a tradeoff point in design.

The copying overhead is serious in list processing, because many sublists and superlists are produced in various places in an execution. Therefore, distributing access to lists thoroughly is more effective than copying lists in the data flow machine architecture. New cells are generated in such a way

as to distribute cells uniformly in SM banks, since appropriate cons strategy enables each cell address to be distributed, due to the functionality of list processing, as mentioned above.

The structure memory is composed of a number of memory banks which can control access independently, as shown in Figure 7. The SM bank construction can resolve the memory access bottleneck, because new cells are taken out and distributed uniformly in each SM bank. The reference count management module (cleanup) for garbage collection is provided in each SM bank. As the reference count method is adopted as described above, the function such as a logic-in-memory is required in order to solve the neck of the reference count update operation.

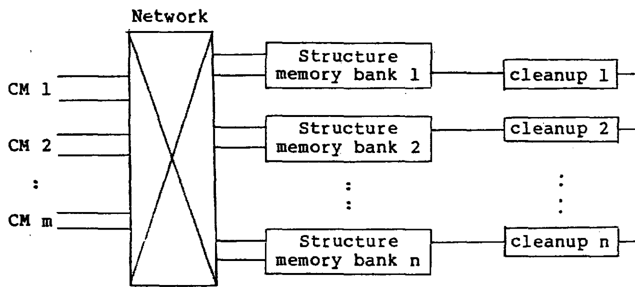


Figure 7—SM structure

The SM bank organization is shown in Figure 8. Data cells in an SM are constructed of three independent blocks, ref, car, and cdr blocks, so as to enhance the primitive-operation-level parallelism. The car(cdr) block consists of car(cdr) ready tag, attr1(attr2) field and car(cdr) pointer field. The attr1 (attr2) field indicates the attribute of the cell pointed by car (cdr) field, i.e., number atom or literal atom or nonatom. Attribute information extracted from the field is also held in an instruction and result packet. The ref block consists of garbage tag and reference counter field which holds the reference number. The ref block is implemented with RAM incorporating the increment and decrement circuits. (The increment and decrement functions are integrated in the memory, based on logic-in-memory concept, so as to reduce the reference count handling overhead in garbage collection management.)

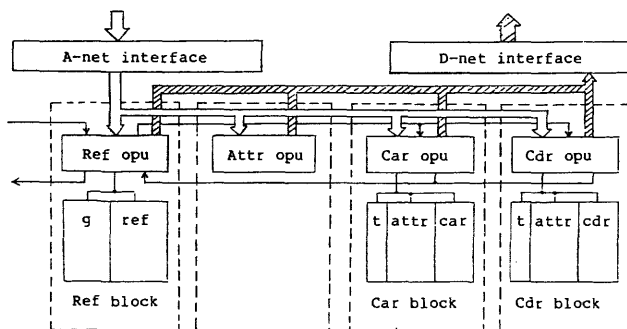


Figure 8—SM bank organization

Specialized operation units are devised for each primitive operation according to the field (i.e., car, cdr, attr and ref) accessed by their operations, as shown in Table I. Car(Cdr) opn performs operations which read the car(cdr) field. Attr opn performs operations which examine the attribute data. Ref opn controls reference count management and performs the getcell operation. Lenient cons operation is decomposed into three operations, getcell, writecar and writecdr, each of which is executed in the Ref opn, car opn, and Cdr opn, respectively. The AN is designed so as to distribute getcell operations uniformly among SM banks.

TABLE I—Primitive functions in SM

OPN	FUNCTION	OPERATION
REF	INCREMENT	increments a reference count
	DECREMENT	decrements a reference count
	GETCELL	gets a new cell
ATTR	ATOM	tests for atomic cell
	EQ	tests for equality on atomic symbols
	NULL	tests for emptiness
CAR	WRITECAR	writes a value in the car field
	CAR	reads a value from the car field
	(CAR-G)	decrements a reference count of the cell pointed by the car field
CDR	WRITECDR	writes a value in the cdr field
	CDR	reads a value from the cdr field
	(CDR-G)	decrements a reference count of the cell pointed by the cdr field

How an operation car(x) is performed is illustrated by an example. The car opn takes an instruction packet from the instruction queue in AN interface and examines attribute information in the instruction packet. If the attribute data indicate that the cell is an atom, the error state is set into the result packet. Otherwise, the memory cell specified by the val field in the instruction packet is read from the ref block and the ready tag is checked. If the ready tag is on, a value z, which is read from the car field of the cell x, is returned to the IM as a result value. If the tag is off (which means the value has not yet arrived), the instruction packet is taken back to the tail of the instruction queue.

The garbage collection mechanism in Ref opn, which utilizes reference counter field, garbage tag field and garbage cell address buffer, is illustrated in Figure 9. The reference number is set to 1 when a getcell operation is executed and explicitly updated by increment or decrement operation.

When the reference count for a cell x becomes zero as a result of the decrement operation, the garbage cell address buffer is checked. If it is not full, the address of the cell x is stored in the buffer. Otherwise, a tag is set at the corresponding address in the garbage tag field. When room is made in the garbage cell address buffer by performing a getcell operation, the garbage tag field is searched and the address of the cell whose tag is set is stored in the buffer. Read and write accesses to the garbage cell address buffer are performed concurrently.

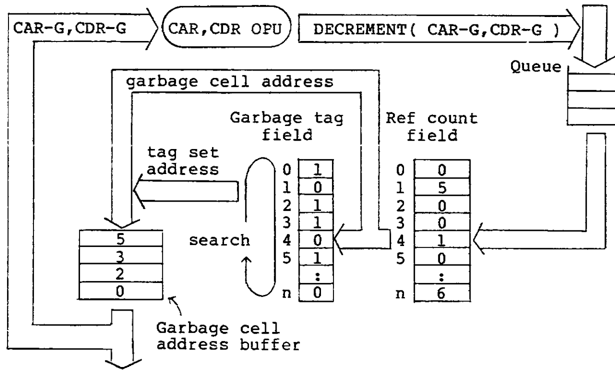


Figure 9—Garbage collection mechanism

The garbage tag search operation is interleaved with tag set operation; it does not itself set a tag. By using the garbage cell address buffer, a free cell address can be quickly obtained in the getcell operation.

6. CONCLUSION

This paper has presented an analysis of some issues concerning list processing under a data flow control environment from the viewpoint of parallelism and has also presented a new type of list-processing-oriented data flow machine, based on an associative memory and logic-in-memory.

The mechanism of partial execution in each function has been shown by example to be effective in exploiting the parallelism in list processing. The lenient cons mechanism has been shown to exploit maximally parallelism among activated functions.

In the list processing under data flow control, memory contention and garbage collection are serious problems. The problem of memory contention can be solved by dividing the structure memory into many banks and by uniformly distributing access in each bank, since new cells may be taken out from any bank.

The reference count is effective as a garbage collection method under a data flow control environment. The garbage collection algorithm presented here works well in the sense that cells are reclaimed whenever they are useless, concurrently with the foreground list operations.

Many problems remain to be solved for the data flow machine to be available for practical use. Several works are in progress to examine the effectiveness of the machine presented here. These include software simulator construction, experimental hardware system design, and VALID compiler implementation.

The simulator, which collects statistical information concerning the lenient cons effect, cons strategy and memory partition effect, and garbage collection overhead, etc., is now running. The experimental hardware system to estimate the cost performance is under development. The VALID compiler written in MacLISP is now under development on the DEC System 20.

ACKNOWLEDGMENTS

The authors wish to thank Dr. N. Kuroyanagi, the director of basic research division, and Mr. K. Yamashita, the director of first research section, for their continuing support and encouragement. They also wish to thank the architecture research group members in the first research section for fruitful discussions.

REFERENCES

1. Dennis, J. B. "A Preliminary Architecture for a Basic Data Flow Processor." *The Second Annual Symposium on Computer Architecture,* Jan., 1975, pp. 126–132.
2. Plas, A. "LAU System Architecture: A Parallel Data-Driven Processor Based on Single Assignment." *Proceedings of the International Conference on Parallel Processing,* 1976, pp. 293–302.
3. Watson, I., and J. Gurd. "A Prototype Data Flow Computer with Token Labelling." AFIPS, *Proceedings of the National Computer Conference* (Vol. 48), 1979, pp. 623–628.
4. Arvind, K., P. Gostelow, and W. Plouffe. "An Asynchronous Programming Language and Computing Machine." Report TR 114a, Department of Information and Computer Science, University of California, Irvine, California, December 1978.
5. Davis, A. L. "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine." *Proceedings of the Fifth Annual Symposium of Computer Architecture,* April 1978, pp. 210–215.
6. Keller, R. M., G. Lindstrom, and S. Patil. "An Architecture for a Loosely-Coupled Parallel Processor." UUCS–78–105, University of Utah, Salt Lake City, Utah, 1978.
7. Dijkstra, E. W. "Guarded Commands, Non-determinacy, and Formal Derivation of Programs." *Communications of the ACM,* 18 (1975), pp. 453–457.
8. Amamiya, M. "A Design Philosophy of High Level Language VALID for a Data Flow Machine." *Proceedings of IECEJ Annual Conference,* 1981, NO. 1486. In Japanese.
9. Friedman, D. P., and D. S. Wise. "CONS Should Not Evaluate Its Arguments." S. Michaelson and R. Milner (eds.), *Automata, Language and Programming,* Edinburgh: Edinburgh University Press, 1976.
10. Dennis, J. B., and K. S. Weng. "An Abstract Implementation for Concurrent Computation with Streams." *Proceedings of International Conference on Parallel Processing,* 1979, pp. 35–45.
11. Amamiya, M., R. Hasegawa, and H. Mikami. "A List Processing Oriented Data Flow Machine and Its Software Simulator." *Proceedings of Meeting on Computer Architecture,* IPSJ, 40–8, 1981. In Japanese.

Lookahead networks

by G. JACK LIPOVSKI
AMBUJ GOYAL
and
MIROSLAW MALEK
University of Texas
Austin, Texas

ABSTRACT

A fail-soft and easily reconfigurable interconnection network is proposed that can function like a bus or like a shift register ring. Its performance as a bus exceeds the performance of an Ethernet, and its performance as a ring is similar to that of a distributed local computer network (DLCN). It can be reconfigured to a sufficient degree to prune out faults or to partition the network into subnetworks that can use possibly different protocols that are the most suitable for the subnetwork. Its multiple-level priority arbitration appears very useful for mixed voice-data networks, to give guaranteed response times to voice packets. Finally, though it functions like a bus or shift register ring, it is physically connected like a tree; so its cost is linear and delay is logarithmic with the number of processors in the network, and it is relatively easy to install in a building by using practices similar to those used in telephone line networks. This paper describes functions of network-level and some data link and physical-level protocols and develops several key mechanisms to achieve ease of diagnosis and fail-softness.

INTRODUCTION

Interest has recently grown in local-area, or establishment, networks, whose work stations (usually processors) are about ten to several hundred meters from each other. Two of the best contending networks for this application are bus oriented, principally the Ethernet,[1] and ring oriented,[2,3,4] principally the distributed local computer network (DLCN),[2] which is based on the shift register insertion mechanism. Two DLCN implementations have been introduced. The first,[2] employing one simplex channel between processors, is referred to here as the simplex DLCN. The second,[5] employing a pair of contradirected channels between processors, is referred to here as the full duplex DLCN. Although the bus and the shift register ring have some advantages over each other, we will be able to show a network that includes both these networks as special cases and has superior characteristics. It is upward compatible to both.

This paper discusses mainly a specific aspect of the protocols used in the proposed *lookahead network*. The specific aspect, in the parlance of the X.25 protocol,[16] is the network level where the interconnection of processors and the determination of routes for messages are defined. We do not discuss at length the link level, where frames or packets are defined, or the transport level, where the breakup and reassembly of user files into frames is defined; and we only focus on a few of the issues at the physical level, where actual physical connections and voltage levels and timings are discussed. The physical-level, link level, and transport level aspects of the protocol can be varied, as determined by further study, to be combined with the aspects described in this paper. Some of our earlier work[6] developed the physical level for optical high-speed interconnections in more detail, and another paper[7] tentatively explored the shift register ring capability.

The following sections describe the lookahead network. In Section 2, a functionally equivalent but much simpler network is introduced. The notions of shift register ring, broadcast bus, and simplex broadcast link are described, using this simpler network; and the priority hardware used for the broadcast bus and the simplex broadcast link is discussed. Section 3 shows some simple applications of the network introduced in Section 2, and thus of the lookahead network. Section 4 introduces the conversion of the simpler network into the lookahead network and describes further ways of reconfiguring the lookahead network to accommodate failures and multiple protocols. Section 5 examines the question of timing at the physical level and proposes a basic structure for the frame at the link level of the protocol. Finally, Section 6 presents some conclusions in support of the claim that this network is better than the Ethernet and comparable to the DLCN network.

2. A FUNCTIONALLY EQUIVALENT NETWORK

The network shown in Figure 1a is similar to the proposed network. It is a ring of AND-OR gates, where each processor has an AND-OR gate by which it can insert data into the OR gate, by means of the GENERATE input, and by which it can permit or inhibit the passage of data through it by means of the PROPAGATE input. The input to each processor is labeled C, and the network is called a *ripple network* because of its similarity to the ripple carry in a full adder. An important special case is one in which the input X is applied to an AND gate, as shown in Figure 1b. Each processor has a two-input multiplexer (MUX) whose switch position is controlled by variable *K*, realized by the two AND gates and the OR gate of Figure 1b, as shown in Figure 1c. This basic function of the ripple network is similar to the network recommended for the ADLC chip of Motorola MC6854,[18] when operated in the ring mode. However, a ripple network is capable of realizing priority circuits as well, as shown in this section.

Trivially, the ripple network can realize the shift register ring or the bus. To realize the shift register ring, put a shift register in each processor whose input is connected to C and whose output is connected to X in Figure 1c, and position each MUX switch to the down position by making *K* equal 0 in each each processor. To realize a bus, select exactly one processor to broadcast data to all the others (in a manner to be discussed shortly). If exactly one processor is selected to broadcast data, that processor's MUX is switched down by making *K* equal 0 in it, and the data are inserted into its X input while all other processor's MUXs are switched to the top position by making *K* equal 1 in them. Then all the processors receive that data on their C input.

A function intermediate between the broadcast bus and the shift ring is the simplex broadcast link. If the MUX in Processors 1 and 4 are positioned downward while all others are switched upward, then the X input in Processor 1 is received as the C input to Processors 2, 3, and 4, and the X input in Processor 4 is received as the C input in Processors 5, 6, and 1. The section of the network beginning with Processor 1, but not including it, and extending to Processors 2, 3, and 4 is a simplex broadcast link; and the section beginning with Processor 4, but not including it, and extending to Processors 5, 6, and 1 is another simplex broadcast link. The ripple network can be partitioned into any number of contiguous nonoverlapping simplex broadcast links at any time. This function is used in the DLCN protocol. Note that the broadcast bus is a special case of this function for one section equal to the entire ripple network and the shift register ring is a special case for each section equal to just one processor in the ripple network.

The broadcast bus and the simplex broadcast link must have

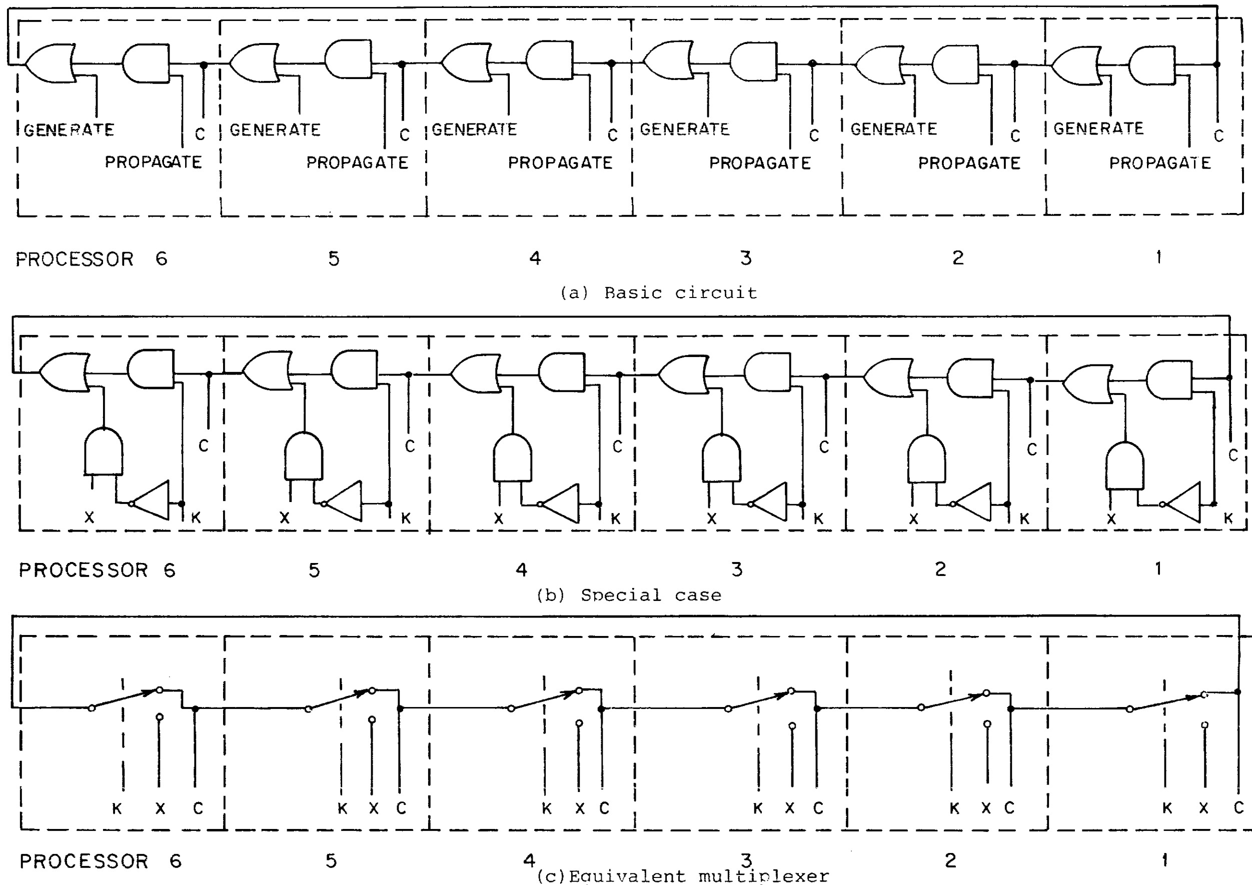


Figure 1—The ripple network circuit

a method to select exactly one, or one or more, broadcasters that will supply data. If this is done by a priori analysis and is then stored in a control memory, the network can be controlled by feeding the output of the control memory to the MUXs.[8] A similar protocol on a broadcast bus is described by Jensen[17] in which every node is provided with a list of the order in which each may send. We will show that the DLCN uses a simplex broadcast link but uses an extra buffer to permit data to be output without knowledge of the state (transmitting or idle) of the other processors. In this case, no priority circuit is needed. In the Ethernet and the Contention Ring networks,[12] exactly one broadcaster is selected by introducing collision detection and random delay capabilities in each node, thereby avoiding the requirement of a priority circuit. Otherwise, an arbiter or priority circuit is needed. In the following discussion, two types of priority circuits are described for the broadcast bus and a priority circuit is described for the simplex broadcast link.

The ripple network contains the logic needed to build a priority circuit for itself so that it can be used as a broadcast bus or a simplex broadcast link. Referring to Figure 1a again, a fixed-priority circuit or a round-robin priority circuit can be implemented for a broadcast bus, as discussed in the following paragraphs.

For a fixed-priority circuit, say with Processor 1 as the dominant processor in the priority evaluation, let PROPAGATE be 0 in Processor 6 alone (or equivalently, do not connect the output of Processor 6 to the input of Processor 1, thus breaking the ring into a chain), and let PROPAGATE be 1 in all other processors. Then if a processor (except Processor 6) requests the use of the bus, it asserts a 1 on GENERATE. Note that this 1 will be received by all processors to the left in the chain. For example, if Processor 3 requests the use of the bus, the C inputs to Processors 4, 5, and 6 are 1. The processor is granted use of the bus if it requests it (GENERATE = 1) and no processor to its right requests it (C = 0). For example, if Processors 3 and 5 request the use of the bus, then both assert GENERATE and no other processor asserts GENERATE. C is 1 in Processors 4, 5, and 6. Only Processor 3 has C = 0 and GENERATE = 1, so Processor 3 is granted the use of the bus. This implements a fixed-priority circuit.

A round-robin priority circuit is subject to an error condition in which either no processor is granted use of the bus or two processors are granted the use of the bus at the same time. An (error-free) round-robin priority circuit is implemented simply by breaking the ring at the processor that last got the grant to use the bus, so that it becomes the lowest priority processor. If Processor i is granted the use of the bus, it sets PROPAGATE to 0, while all others set PROPAGATE to 1. If Processor i again requests use of the bus, it does not assert

GENERATE, but instead it simply receives the grant if its C input is 0 because no other processor requests use of the bus. All other processors operate as in the fixed-priority case, asserting GENERATE if they request the use of the bus and getting a grant if they have GENERATE = 1 and C = 0. Incidentally, if Processor i needs to use the bus for longer than a normal cycle, it can retain GENERATE = 1 until it is through, because this will assert C = 1 in all processors, thus preventing them from getting the grant. This implements the (error-free) round-robin priority circuit.

This round-robin scheme is similar to token passing,[3] but there is only a short delay through combinational logic in bypassing processors that do not need service, whereas the token-passing scheme required each station to hold a token for a memory clock cycle. Moreover, this delay in the round-robin scheme will be further reduced in the lookahead network to be a logarithmic function of the number of processors. This speedup might be significant for fast (optical) networks. It allows successive evaluations of priorities of different priority levels, so that voice packets can have priority over data packets, and bridge or window sources can have priority over other servers, as we will discuss in Section 5. This capability may prove to be critically important in mixed data-voice systems.

A fixed-priority circuit for a simplex broadcast bus can be deliberately or accidentally created if two or more processors set PROPAGATE to 0 while all others set PROPAGATE to 1. Then each partition of the ring through which propagate is 1 has a separate priority circuit, and each will grant the use of the bus. If the bus is configured as a simplex broadcast bus, over which each partition block has a separate priority circuit, then the priority circuit can serve to grant the use of that block to a requesting processor in that block. That is a mildly useful feature.

However, an error can occur in a round-robin full broadcast bus priority circuit in one of two ways. Either a processor that did not get the grant can accidentally set PROPAGATE equal to 0, so two processors have set PROPAGATE equal to 0; or a processor that got a grant can accidentally set PROPAGATE equal to 1, so that no processor sets PROPAGATE to 0. (This is similar to the error in token-passing protocols where two tokens are created, or the token is lost.)

If two processors set PROPAGATE equal to 0, then the priority circuit grants use of the bus to two processors, so the data output on X of a processor will not be received in input C of the same processor for both processors. The data are separated, rather than ORed together, because in the broadcast bus the processor that sends data also sets PROPAGATE to 0 when it is sending data, so the two processors that are sending data will receive data from the other processor.

If two processors are granted use of the bus, both processors will have to request that the fixed-priority circuit mechanism be used in the next cycle, and the next cycle only, and that the data just received be ignored. This request is sent by means of a signal sent into the GENERATE input while the PROPAGATE is set as for the previous broadcast in all processors. Both processors send out a signal, which will be the same signal. One processor will send this signal to all the processors in the ring up to the second processor that broadcast data, and it will send the signal to the other processors up to the first processor. The same effect will be created when more than two processors appear to get grants, and the result will correct the fault.

If no processor sets PROPAGATE equal to 0, then a processor setting GENERATE equal to 1 will automatically set its own C = 1, so no processor will broadcast at all. This is due to the fact that the GENERATE output will propagate through each processor, all the way around the loop, setting the C input to 1 in the same processor that set GENERATE equal to 1. We will assume that the protocol has a frame called a null frame (or idle signal), which appears on the network when no processor is sending a frame, and that this null frame is never sent by any processor as data. If a processor detects that it had requested use of the bus, and that subsequently a null frame appeared on the bus, indicating that no processor used the bus, the same request signal that was used to signal multiple grants is asserted, which causes the fixed-priority mechanism to be used in the next cycle and the current cycle to be ignored. (The bus temporarily has PROPAGATE = 1 at all processors, so it acts as a set-clear flip-flop and is "set" by any processor that wanted to use the bus. This flip-flop latching effect is terminated when one processor breaks the loop to become the lowest-priority processor in the fixed-priority circuit.)

The broadcast bus frame protocol will be defined to have a frame priority error bit for either the generation of multiple grants or the loss of a grant. This bit will appear at the end of the frame.

3. SOME EXAMPLES OF THE USE OF THE RIPPLE NETWORK

Before passing to the lookahead network, we would like to establish the advantages of the simpler ripple network relative to the conventional broadcast bus (Ethernet) and the ring (DLCN). This will be done in a qualitative but fairly rigorous way in this section.

The Ethernet is a contention bus protocol. That means that when two or more processors request use of the bus, they send a signal and listen to the bus. They do not get the signal they sent when two or more processors request the bus, because each processor sends a different signal. They wait a random amount of time and then send the signal again. When one finally makes a request while the other is waiting and not making a request, then that one gets a grant and uses a bus. Although this does not degrade a lightly loaded network significantly, as the load increases, more processors will send signals, more collisions will occur, and more tries will be needed before a processor will eventually succeed in getting a grant. No practical backoff algorithm exists that can reach the theoretical lower limit of 1.72 collisions (on an average) per successful transmitted frame.[9] In fact, the contention protocol busses, like the Ethernet, become unstable and provide excessive response times as the throughput is increased beyond the 67% of the capacity of the channel.[9]

By comparison, the ripple network has a built-in hardware priority circuit. It uses a different priority mechanism but is still a bus protocol like Ethernet. (The lookahead network can implement a contention protocol that uses digital codes rather

than analog levels to determine contention.) When two or more processors request use of the bus, they assert their GENERATE inputs simultaneously, and one processor receives the C = 0 that indicates it may have the bus at the end of a time t that is *independent* of the number of requesting processors. In the ripple network t is proportional to the total number of processors in the circuit, whereas in the lookahead network it is proportional to the logarithm of the number of processors, as will be shown later. This shows that the performance of the ripple network is superior to the performance of the Ethernet in that capacity is significantly increased.

The simplex link DLCN network[2] uses a variable-length shift register in each processor that is sending data and a zero-length shift register in processors that do not send data. As data is sent by a processor, the data that are coming into the processor from the previous processor in the ring are stored in the shift register so that it can be sent out after the data from this processor are sent out. In this way each processor can make a decision about whether to send data based only on local information (comparing the amount of data to be sent to the amount of room left on the variable-length shift register to buffer the incoming data that it will replace), and a priority circuit is not needed. The simplex DLCN network is just a ripple network in which the MUXs in each processor have more than two inputs, so that data can be taken from different taps in a shift register or from the data being input to the processor to implement a variable-length shift register with a bypass. Thus, the simplex DLCN network is a special case of the ripple network (Figure 1a) that is different from the special cases studied above (Figure 1b, Figure 1c). The duplex DLCN network will be compared to the lookahead network later.

The ripple network is capable of being used as a broadcast bus, with similar characteristics, but with better performance than contention busses like the Ethernet and as a shift register ring like the simplex DLCN. The bus protocols are advantageous when a command or some data have to be seen by all processors. The DLCN protocol can have twice the capacity of the bus protocol for randomly generated messages because, on the average, the message will use only half the network and the other half can be used for another message. The ripple network can be used in either mode.

In addition, the ripple network can be used in special cases as a simple shift register ring—for example, in the analysis of data acquired in oil exploration (linear filtering, convolution integration, and correlation), in some office systems (multiple-query analysis on the same stream of data), and in pipelined processes, such as those that might be created in a UNIX operating system. When these special cases exist, the simple shift register protocol can maintain constant capacity between two processors as the number of processors increases, while the other protocols would reduce capacity between any two processors proportionally to the number of processors in the network. (This weakness of the bus and DLCN protocols is partially ameliorated in the lookahead network.) The simplex broadcast link protocol has some advantages where a processor tends to broadcast data to a limited group of processors that can be placed after it in the ripple ring, such as in processing arrays of data where each processor stores a subarray.

Each protocol has significant advantages for some class of problems, and there is no unchallenged claim to universal superiority. But the ripple network and later the lookahead network are capable of performing as these networks do.

Moreover, Bokhari[10] is investigating techniques where local optimization is done using the ring type of interconnection, and global optimization is done using a bus. Since global optimization involves $o(n**2)$ complexity, whereas local optimization involves $o(n)$ complexity, local optimization permits the parameter n to be reduced before it is squared. This flexible bus allows such algorithms to be used without the cost of implementing two separate interconnection networks.

However, the entire ripple network will have to function in one mode or another at any given time. We would like to be able to partition the network so that different parts of it could use different protocols that are adapted to the problem being executed in that part. Also, a failure of any processor, especially a stuck-at-one failure where it keeps broadcasting data forever, will bring down the entire network. For these reasons, we introduce the lookahead network in the next section.

4. THE LOOKAHEAD NETWORK

The ripple network can be used as a chain of MUXs, which may realize many different networks, as we have demonstrated in the last two sections. It is structurally the same as the ripple carry circuit that is used in the parallel adder. As is well known in computer design, one can create a circuit functionally equivalent to any combinational circuit in two levels of logic, just using the sum-of-products expansion; and the sum-of-products expansion of the ripple carry circuit is the full carry lookahead circuit. While it features constant (two-level) delay for arbitrary n (number of bits added), it is a complex circuit, even when put in one chip, and is certainly too complex to be used as a local-area network. Between these two extremes, the ripple circuit and the full lookahead circuit, a recursively defined lookahead circuit can be implemented in a tree structure. Basically, for an n-bit adder, f bits are combined in a group, and this group, as a $2**f$-ary digit, is added with other similar groups to combine them into a group of bits equivalent to a $(2**f)**2$-ary digit, using a full carry lookahead circuit to implement addition in each group. This combination into larger groups is repeated until the entire n-bit number is one group. This creates a tree having log n base f levels, and each nonleaf node in the tree is a carry lookahead circuit. The most common such circuit is that for which $f = 4$, and widely available ICs such as the "carry-lookahead generator" 74182 and the 2902 can be used. The simplest circuit to describe, however, is that for which $f = 2$ (see Figure 2). We will discuss that circuit in this paper, but the theoretically optimum network has $f = 3$;[20] and results will apply to any fixed f, and for that matter, any arbitrary f that can be different at different nodes.

Note that each link in the tree appears to be implemented with three wires, GENERATE, PROPAGATE, and C, and the logic to establish these signals is very simple, as shown in Figure 2a. The C input from the father is the carry into the group of bits and is the C input to the right son. The C input to the left son is 1 if the right son GENERATEs a carry or if the C input from the father is 1 and the right son PROPA-

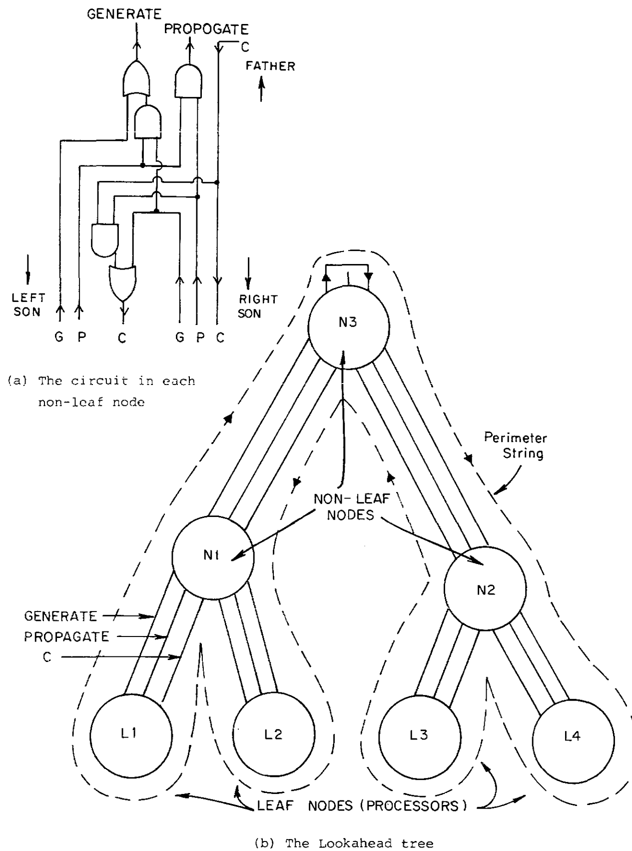


Figure 2—The lookahead network

GATEs this carry. The group GENERATEs a carry if the left son GENERATEs a carry or the right son GENERATEs a carry and the left son PROPAGATEs it. Finally, the group as a whole PROPAGATEs a carry if the left and right sons PROPAGATE the carry. Figure 2b shows a binary tree, whose leaf nodes, L1 to L4, are processors and whose nonleaf nodes, N1 to N3, are those shown in Figure 2a. To implement the ring-structured ripple network, the GENERATE from the root node is connected back into the C input of the root node. This connection does not introduce any unstable feedback loops, because the GENERATE from the root node is logically independent of the C into the root node; indeed, this connection is used for the "end-around carry" in one's complement adders.

The recursively defined carry lookahead circuit, hereafter called the lookahead network, has a physical structure that is a tree, with an arbitrary fanout at each node. The leaves of the tree are processors, and the other nodes are essentially just "carry lookahead generator" chips like the 74182, which may contain a buffer memory or a processor for the repeater, bridge, and gateway mechanisms discussed later. Most nodes are expected not to need these optional capabilities. The interface between the processor and the network is an I/O chip similar to that for the SDLC or ADLC network, such as the Motorola 6854.[18] This structure is simple to wire up in a building. The root of the tree, an arbitrary point, could be wired to a closet in each area where the telephone equipment is currently housed; the lines from there could be wired to each room where the next node appears; and from there it could be wired to each terminal, printer, mass storage device, and so on.

The nonleaf nodes should be implemented as an IC with a fixed fanout f = three, as we see later. If a node N in the implementation, like a node housed in a telephone closet, requires fewer sons than f, the other sons will be pruned by means of a technique introduced later. If the node in the implementation requires more than f sons, the next highest fanout f**i for some integer i can be realized by a tree having i levels, and the leaves of that tree can be connected to the sons of node N.

The lookahead network is simple to use at this point, because it is functionally equivalent to a ripple network (introduced in the previous sections). However, several respected colleagues who have recently learned about the network have expressed some fear about its complexity. In response, we have pointed out that such a circuit has been used in the ALU of most computers since the 1950s and that despite its apparent complexity it has been successfully treated as a gray box[14] that most users use daily but would not care to understand. In this paper we will present the details of the logic. Henceforth, after reading this paper, we hope that the user will simply treat it as a subsystem that has the same functional characteristics as the ripple network and also a few other characteristics, described below.

The lookahead network has the characteristics of the ring network and some further modest advantages. As just discussed, it can be wired as a tree. Moreover, no signal will ever travel further than the distance from a leaf node to the root and back to another leaf node. This physical distance can be shorter than the distance a signal may travel in the ring. Assuming that n processors are arranged in a tree with fanout 7 and each processor takes a cube unit of space, the number of gates can be proportional to the log base 7 of n, and the free-space propagation delay can be proportional to the cube root of n. Our original motivation for studying the lookahead network was to improve its speed characteristics, especially for very fast optical-fiber networks.[6] However, other advantages (pruning and wire-OR bussing) discussed below substantially expand the applicability of this network, and speed has become a rather modest advantage compared to the subsequently discussed advantages.

A significant advantage of the lookahead network over the ripple network is its ability to form the equivalent of the wire-OR bus (see Figure 2). If the PROPAGATE are all 1, then the GENERATE out of the root node (node N3 of Figure 2b) is the OR of the generates out of the leaf nodes (nodes L1 to L4). This is fed back to the C input of the root node. This C input is sent to all C inputs of the leaf nodes. Incidentally, this signal may be ORed with some GENERATE signals from leaf nodes to produce a C input to a leaf node; but since this C input is itself the OR of all GENERATE signals, the effect is that the C input to the root is broadcast to all leaf nodes. Further, since the root node GENERATE output is logically independent of the root node C input, there is no possibility of oscillation or lockup. This possibility of lockup exists for the ripple network when all PROPAGATE are 1. The ripple network behaves like a set-

clear flip-flop. Since lockup does not exist in the lookahead network, that network is capable of supporting the equivalent of a wire-OR bus. Wire-OR busses can be used to collect acknowledgments or status signals from a set of processors. They can also be used to implement a digital contention policy, which is equivalent to the analog contention policy used in Ethernet.

The most significant advantage of the tree is the ability to prune subtrees so that faulty nodes can be pruned and so that the subtree that was pruned away from the rest of the tree can execute the same protocol as the rest of the tree on a smaller loop or ring, or it can execute a different protocol from that used in the rest of the tree. This advantage is similar to that proposed by Arnold and Page[11] for their hierarchical bus architecture, and that architecture can be a special case of this lookahead network. In order to explain the pruning idea, we have to understand the lookahead network functions that are equivalent to the ripple network functions.

Except for wire-OR and pruning functions, the lookahead network is functionally equivalent to the ripple network, as is fairly well known among computer designers. To show the reader that the two networks are indeed equivalent, we will describe in detail the realization of the shift register ring and the broadcast bus implementations in the lookahead network. Comparing these to the previously described implementations of the ripple network, we can verify that they have the same input-output function—that they are functionally equivalent. Other functional equivalences—priority evaluation and simplex broadcast bussing—are also easily demonstrated, but they are omitted to shorten the paper.

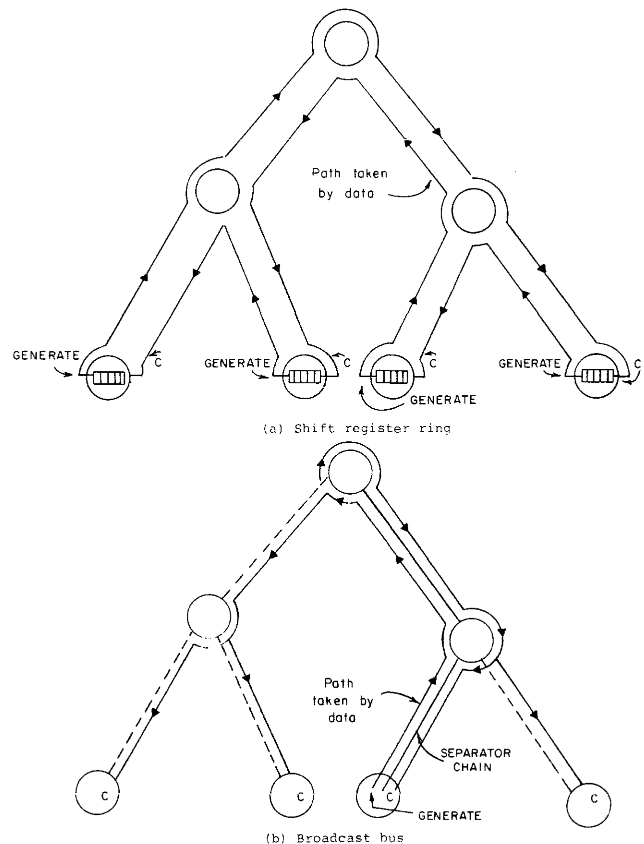


Figure 3—Some basic functions on the lookahead networks

Figure 3a shows the lookahead network as it is used for a shift register loop. The PROPAGATE is 0 in each processor, just as in the ripple network. Note from observing Figure 2a that the PROPAGATE is then 0 in all links, since the PROPAGATE sent by a node toward the root is the AND of the PROPAGATE sent into it. Therefore, the C line from a node toward a right leaf is the C line from the rootward node, the C line toward a left node is the GENERATE from the right node, and the GENERATE toward the root node is the GENERATE from the left leafward node. The signals travel around the tree as if it were on a string wrapped tightly around the tree, which we call a perimeter string, as shown in Figure 2b. The string is not really there, but the signal travels through the C and GENERATE lines just as if it traveled through the string. The effective communication path is as shown in Figure 3a. The most significant concept here is that the circuit is functionally equivalent to the ripple network in the implementation of the shift register ring.

Figure 3b shows the lookahead network as it is used for a broadcast bus. One processor is selected to broadcast data to all processors. The selected processor sets PROPAGATE to 0 and all others set PROPAGATE to 1, just as in the ripple network. The PROPAGATE is 0 in all links between this processor and the root of the tree, which will be called the separator chain, because the PROPAGATE rootward is the AND of the leafward PROPAGATE. All other links that are not in the separator chain have their PROPAGATE signals equal to 1. The selected processor puts its data on the GENERATE, and all other processors put 0 on their GENERATE lines. These data flow up the GENERATE line on the left side of the separator chain, because all the PROPAGATE that are not in the separator chain are 1. The data then flow down the C line from each node in the separator chain and flow to the left and right sons on the C line because the PROPAGATE is 1 in all these links that are not in the separator chain. The data are broadcast to all nodes to the left of the selected node. We always feed the GENERATE that emerges from the top of the root node into the C that is sent into the root node, as explained earlier. Thus the data, emerging from the GENERATE above the root, are sent into the C at the root. The data sent into the C at the root will be sent—at each node whose right son is not in the separator chain—to the C output toward both sons, and—at each node whose right son is not in the separator chain—to the C output toward both sons, and—at each node whose right son is in the separator chain—toward the right son. The data are broadcast to all nodes to the right of the selected node, including the selected node. The important point, again, is that the lookahead network is functionally equivalent to the ripple network when used as a broadcast bus.

The same analysis can be used for the simplex broadcast link and the fixed-priority and round-robin priority systems. The important conclusion is that the lookahead network is functionally equivalent to the ripple network. We now can attend to the more significant issue of pruning.

A lookahead tree can be pruned at any link. Figure 4 shows an example of pruning at link a. The subtree below a will form

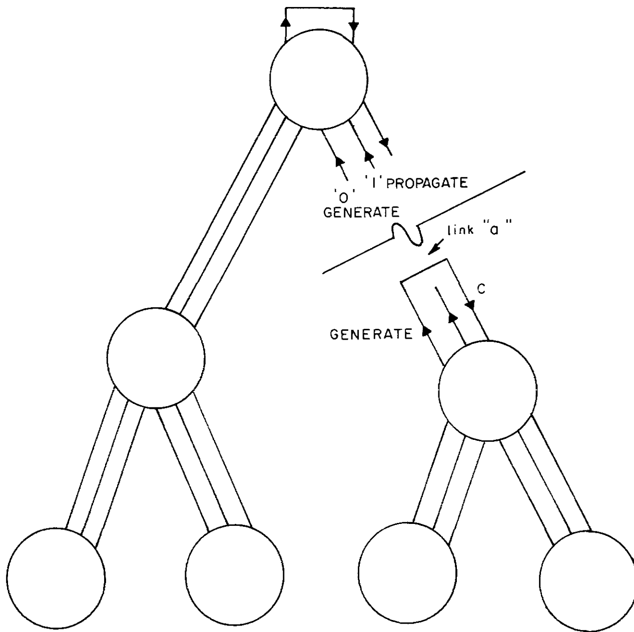


Figure 4—Pruning a lookahead network

an independent tree capable of implementing a shift register ring, a broadcast bus, a simplex broadcast link, or a fixed or round-robin priority circuit while the rest of the tree is capable of independently doing one of these things. To separate the subtree from the tree, the GENERATE, PROPAGATE, and C signals merely have to be forced into a state that they would be in if no signal appeared from the subtree or into the subtree from the main tree. This is easily accomplished. The GENERATE is forced to 0, so the subtree will not generate data into the rest of the tree; the PROPAGATE is forced to 1, so that the subtree will not sever the rest of the tree in a separator chain; and the C into the subtree is forced to be the GENERATE out of the subtree, so that the subtree will have its own "end-around carry" to implement the various functions. The pruning operation is very simply implemented.

The pruning operation can prune faults. Some thought has to be given to the control of the network that selects the link to be cut lest the control itself use the faulty node to prune the faulty node. We discuss the selection of the link to be pruned, together with the most demanding application of pruning, in the next two paragraphs. The nonleaf nodes are not always processors, but are usually essentially lookahead generator chips. They can be selected and set to disconnect the links above them or below them in one of two ways.

One way uses the standard addressing technique in trees. In a binary tree, a binary number selects the nodes at the level of the tree that equals the length of the number; the most significant bit chooses the right (0) or left (1) son of the root, the next most significant bit chooses the right (0) or left (1) son of that node, and so on, until the least significant bit is sent down. At that time, the node just selected by the least significant bit is marked and modified. The leftmost processor can send the signals, bit-serially most significant bit first, through its GENERATE line, to the root of the subtree that it is in. The signal at the root is sent down the C lines, which allow the signal to pass only along the path established by the prior address bits and to extend the path one link each cycle, as described above. A simple sequential machine is needed to implement these operations in each node.

The other way uses the intersection of chains like the separator chain used in the broadcast bus function (see Figure 5). Suppose that we wish to select a particular node N. Then let P1 be any processor in a subtree below the right son of N, and let P2 be any processor in the subtree below the left son of N. If P1 establishes a chain to the root node, and P2 then establishes a chain to the root node, the node N will be where these two chains meet. That node can be selected and modified. Although the latter technique cannot prune out a single broken node in a binary tree, because the broken node must participate in the pruning technique, it can work in a tree with fanout greater than 2. Two good processors could select the node above the bad processor and command it to disconnect the link to the bad processor. Although either technique can be used, we think the latter is simpler, and we will show how it can be implemented in the link protocol.

The pruning technique can be used to create separate loops, busses, or rings that do not communicate to each other. Using this mode, we can optimize the size and the protocol of each subtree to the application. The size and protocol selected for each subtree could be adjusted by a technician upon installation of the network or upon modification or expansion of it, or it could be done dynamically by an operating system. The latter technique appears fairly complicated but promises substantial rewards. In either case, the tree structure poses some constraints. Wired in a natural way between rooms, the subtrees would contain the processors in one room or a floor, and so on. However, processors of one type, scattered throughout the local area, may require one protocol, whereas processors of another type, scattered throughout the local area, may require another protocol. If we establish different trees for each type of protocol and tie their roots together to form a single tree, we can accommodate these requirements; but the simple wiring scheme that is natural for trees becomes messy:

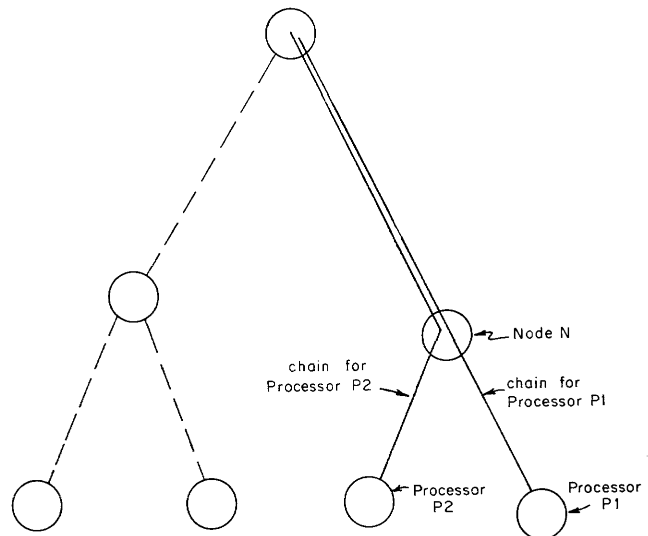


Figure 5—Selection of nonprocessor nodes

we have a thicket rather than a tree. This disability, incidentally, is equally true for busses or rings. Multiple busses or multiple rings would have to be wired around a local area if different processors had to be attached to different rings (or busses) and the processors attached to the same ring (or bus) happened to be scattered throughout the local area. The lookahead network allows these to operate separately or to be joined into a single ring (or bus) if that is useful.

If two loops are created, a bridge is a port on one loop and a port on the other loop in the same node that permits data from one loop to the other in which the frame size and protocol is not altered, but the signaling rates and access mechanisms may change.[12] A repeater is simpler than a bridge: it does not permit the signaling or the access mechanism to be modified at all. A gateway is more complex, allowing any change of the protocol.[12] Bridges and repeaters between loops are reasonably straightforward. We consider the three protocols that have been studied earlier in the paragraphs below.

A bridge between two broadcast busses could be implemented this way. Requests from a processor would be handled by the priority implemented in the bus that they are connected to. When granted use of the bus, the processor would select the bridge, command it to request the next bus, and test the result to see if the request was granted. If the request was granted, the processor could repeat the process to acquire bus by bus until it reached all the destinations that it intended to reach. When it had acquired all the resources it needed, it could send the data. If the request was not granted, it might continue to try to form the next bus for a period of time; but after the period was over, it would abort the request and try again later so that it would not tie up the local bus and so that it could thus avoid deadlock. The period would be determined by analysis of the load and the loss due to blocking the local busses. In order to enhance the chances of completing the circuit, the priority network for each bus should not be strictly round-robin, but should favor the bridge over the lower (leafward) level nodes of the subtree that realizes the bus. Nevertheless, the other nodes should be treated fairly. This observation is forwarded to the discussion of the frame structure in the next section.

A shift register ring can be bridged to another shift register ring without difficulty. In fact, the Pierce loop[4] is a static implementation of this protocol. Figure 6 shows how the "perimeter strings" would appear for both loops where the node below the pruned link implements a bridge between the loops.

A DLCN can be bridged to another DLCN, provided that each node where a bridge can be set up has a buffer memory that is larger than the longest frame. We are currently studying this mechanism, and we are using simulation to determine the size of the buffer needed to reduce the number of frames that are rejected at the bridge.[13] If the rootward loop is lightly loaded, the buffer appears to have to be twice the maximum packet length to insure that more than 90% of the frames are successfully transferred.

Interfacing among loops that have different protocols is still under study. However, the transmission of a fixed-size small frame, adequate for control signaling, appears to be fairly simple between the protocols. The frame size has to be limited to the frame size of the shift register ring, which is fixed to the

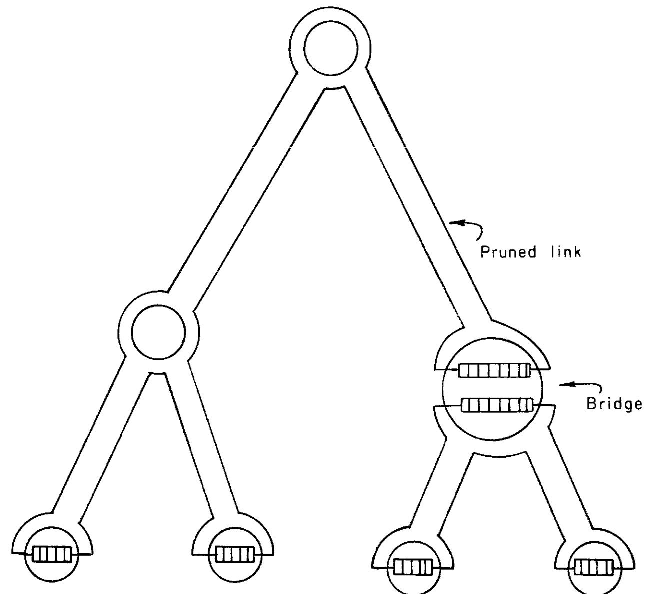


Figure 6—A bridge in a shift register ring that has been pruned

size of the shift register implemented in hardware. Since such a frame would not have to be reformatted, the rootward nodes of the lookahead network would only need to handle the physical levels of the protocol and not the link and transport levels. If complete generality is required, as in a gateway, then the upper-level nodes where a gateway may be implemented would have to be intelligent processors with considerable memory. Note, however, that nodes where a repeater, bridge, or gateway will never be implemented can be built without either buffer memories or processor intelligence.

It appears that a lookahead network with carry lookahead generator logic, logic to select and prune links in the tree, and local memory capable of holding two maximum-sized frames is a very powerful distributed system. A more economical system, deleting the memory buffer, would not be able to interconnect DLCN loops, but would use nodes that are just a little more complex than the carry lookahead generator chip, as described earlier. Moreover, a system built with the less costly chips could be upgraded to the more general system by replacing the chips with modules that have memory or processors and memory, and this could be done in some but not all nodes. Users can upgrade systems to fit their needs.

5. TIMING AND FRAME STRUCTURE

This section discusses the question of the structure of frames in the link level of the protocol, which will be compatible with the modes of operation of the lookahead network. The most complex protocol is that for bus operation. To permit multiple subtrees to use different protocols, but to allow bridges between them, we can implement the simple shift register ring and protocols of the DLCN type, using the same frame structure as that for the bus operation. Timing for the entire network could be synchronized frame by frame. The other protocols would allow time for the signals needed by the bus protocol, but would do nothing during that time. Alterna-

tively, each subnetwork could use its own frame structure tailored to the protocol used in it. Though bridging would be more complex, efficiency within each subnetwork would be improved. In this case, the frame protocol would delete the parts needed only by the bus protocol. In either case, a study of the bus protocol frame structure leads to the other frame structures by removing the special signals needed by the bus protocol.

We have established that we need to send data, evaluate requests by means of a priority circuit, send a frame priority error signal, send commands to nonleaf nodes, and use wire-OR techniques to determine whether any processor has some specified condition. We now consider their timing and then their placement in a frame.

The data should constitute a very large percentage of the signals, or there is something wrong with the protocol we are using. Data are sent by a single source in the lookahead network. Thus they can be self-clocked, using a variation of the well-known Manchester clocking scheme used on disks. In this technique, a clock pulse is sent every nth data pulse (n is usually 1), and the receiver uses a phase-locked loop to extract the clock and reestablish the timing needed to extract the data. For n about 10, the mechanism used in the UART uses the clock pulse as a start bit. It then uses a faster clock, synchronized to the start bit, to extract the data. If n is 1, only half the available bandwidth of the line is used for data. For larger n, the efficiency is improved, but a phase-locked loop with a smaller capture window is needed. Some n has to be fixed to define the physical-level protocol; but this depends on the tracking of the transmitter and receiver oscillators and the probability of missing data, so we are not ready to do so now. Whatever the case, if the clock is sent with the data, the wires themselves appear to form a pipeline, and data can be sent through at the capacity of the wire. In particular, optical cables will allow very high bandwidths for data. The clock interval for this type of transmission will be termed a local clock tick, and its rate will be called the local clock rate. The local clock tick can be derived from a phase-locked loop circuit in each node.

The priority mechanism, the verify pulse, the wire-OR mechanism, and the signals selecting and controlling the nonleaf nodes do not have the property that the signal must come from one source, so they will not usually complete their operation in a local clock tick. In the priority circuit, two or more processors could request the use of the bus in the same interval of time. The entire network, or at least the subtree in which the processor is located, must settle down to a steady state before any processor can look at the data on its C input. The time to settle down is proportional to the logarithm of the number of processors plus the physical length of the longest line that is proportional to the cube root of the number of processors. This time must elapse from the time when the last processor might change its GENERATE or PROPAGATE inputs until any processor may examine its C input. We will call it the system clock tick, and its rate will be called the system clock rate. It can be implemented by setting the system clock tick to n local clock ticks, for some fixed n, and using a divide-by-n counter out of the local oscillator in each node.

A multilevel priority system, where each level is handled by the fair round-robin priority technique, can be implemented by using the same hardware in successive time slots. All highest-priority devices would send requests in the first time slot. The round-robin technique would be used in each time slot. If one wins the grant, the other levels may be omitted or else executed and ignored. A time slot is needed to determine whether a request was successful. In this time slot each processor that has the priority level sets GENERATE to 1 if it has requested service and to 0 if it does not request service, but all set PROPAGATE to 1. The requests are ORed together, using the equivalent of the wire-OR bus discussed earlier, and the OR of the requests is sent to the C input of each processor. If no processor claims the bus in the first time slot, then second-priority-level devices compete in the third time slot, and so on. In particular, at least a three-level priority system is useful in a pruned tree so that repeaters and bridges can have higher priority than other processors and voice traffic can have priority over data traffic; but the other processors would be given fair and equal priority by means of the round-robin technique. This mechanism not only improves the performance of bridges and repeaters; it makes possible a mixed data-voice network, with prompt response to voice communication.

The TENET system[19] also uses priority slots before transmitting a frame on a contention bus. However, packets in a particular priority slot can still collide, introducing random delays. The multilevel priority scheme on the lookahead network selects a packet within one time slot, giving a guaranteed upper bound on the response times.[20]

It will be noted in the next section that the PROPAGATE wire ought to be deleted to reduce cost. This can be done because the PROPAGATE changes once per frame rather than once per data bit. It changes after the winner of the arbitration is selected, for the winner sets up the separator chain to send data, and that same separator chain becomes the place where the carry is inhibited in the next priority evaluation. Finally, the output PROPAGATE can be computed on the sole basis of PROPAGATE inputs. The PROPAGATE can be sent before the GENERATE signal.

The frame structure for a bus protocol now appears this way (see Figure 7). The broadcast bus or simplex broadcast link will begin with one or more priority operations, and each priority operation will be followed by a checking operation, using the wire-OR technique to see whether a processor is selected. Each priority operation or checking operation is one system clock tick long. The next system clock tick is used to send the propagate signal P over the same lines as the generate G, but ANDing the son's Ps to produce the P out rootward from each node and storing the Ps in each link in a flip-flop. Data will be sent at the local clock rate and will include some structure like a control block, source and destination addresses, a data block, and an error-checking block. It must never send a frame, recognizable as a null frame, to allow the round-robin priority circuit to recover from the loss of a grant. The SDLC, ADLC, and HDLC protocols would be suitable, because the null frame could be eight ones, which is prohibited in the data frames sent by these protocols. The frame priority error signal would then have to be sent at the end of the frame for a system clock tick. The frame structure for protocols other than the bus protocol can omit the priority operations and checking operation as well as the priority er-

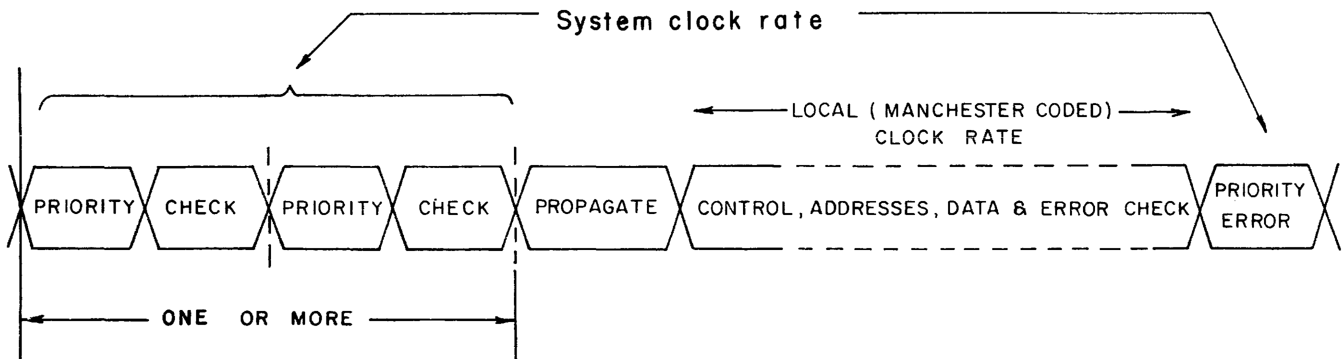


Figure 7—The format of a lookahead frame at the data-link level

ror signal at the end of the frame. Alternatively, other protocols can allow time for these signals and do nothing during them. The latter is recommended for frames in which the extra time (about six system clock ticks) is small compared to the total time for the frame, because bridges and repeaters would be easier to construct.

The control over nonleaf nodes should not be put in the normal frame structure, in our opinion, because a large amount of time would be required for the system clock tick needed to communicate with upper-level nodes in each frame, and this would be done infrequently. The control of nonleaf nodes can be implemented with simple hardware and low signal overhead if the upper-level nodes are able to detect the null frame and to respond to the PROPAGATE inputs in the two system tick periods after it occurs. We assume that such commands will be infrequent and that all the processors in the tree or subtree can be interrupted to cooperate in this operation. Under normal recovery from a loss of a grant, only one processor will set PROPAGATE to 0 to reestablish the priority mechanism, by means of the fixed-priority technique. (If that processor fails, reconfiguration will be necessary to prune it out of the bus.) The pair of processors that intends to command a nonleaf node can send a null signal on the bus to force an error condition on the processors and signal the nonleaf nodes to listen. The processors wait until the fixed-priority circuit is established, and then in the second system clock tick after the null signal was sent, the two processors send PROPAGATE = 0 while all others send PROPAGATE = 1. The PROPAGATE = 0 signals travel up the chain to the root of the tree and meet at the node that is to be selected. Once a node is selected, commands to connect or disconnect its links toward its father or any of its sons can be sent from one of the processors. This mechanism will be able to prune a stuck-at-zero failure. Though this procedure is not able to handle a stuck-at-one fault because the stuck-at-one processor cannot be counted on to cooperate, it would nevertheless be possible to use a bootstrap fault diagnosis scheme.[15] In a bootstrap scheme, we reset all links to the disconnected state by applying a power-on reset signal, then merge the small trees (initially single leaf nodes) into larger trees until the largest tree that does not include the failure is grown. When we grow a tree that is too large and includes a failed node, we start over and grow a tree that just excludes this node.

6. COMPARISON OF THE LOOKAHEAD TO THE ETHERNET AND THE DLCN

The lookahead network can be compared to the two popular networks, the Ethernet and the full-duplex DLCN, on the basis of how it handles failures and on timing and throughput. These comparisons are discussed below.

The ability to prune subtrees enables the operation of parts of the network to continue while other parts have failed. The full-duplex DLCN network[5] is capable of doing this too. The DLCN network can allow arbitrary connection among any pair of nodes when one node fails and can allow fail-soft capability, where a node can communicate to about $1/n$th of the nodes on the average if n nodes fail. The Ethernet can also be partitioned into smaller busses by means of repeaters.[12] These gates introduce a delay that is linear with the number of processors, and they allow about the same fail-soft capability as the DLCN, but not the fault-tolerant capability. A processor can expect to reach about $1/(n + 1)$th of the network if n nodes fail. The lookahead network is very tolerant of failures in leaf nodes, which can be pruned away as subtrees of height 0, and allows full communication among nodes if any number of leaf nodes fail. It is susceptible, however, to failures in the nonleaf nodes. For example, a failed root node will degrade the network into a mode where each processor can communicate to half of the other processors. This is comparable to the expected fail-soft capability of the Ethernet experiencing a single failure, or a DLCN with two failures. Nevertheless, we expect that most failures will occur in leaf nodes. This network should be superior to the Ethernet and the full-duplex DLCN in fault-tolerant and fail-soft capability.

According to the reliability analysis performed by Goyal,[20] the theoretical optimum fanout for the lookahead network is 3. The reliability graphs of various networks with respect to a single node network are given in Figure 8. Curve A shows the degradation in the reliability of a simplex ring as the number of nodes is increased. However, bypassing on node failures, as done in DLCN, can improve the reliability, as shown in Curve B. A logical ring, formed on the lookahead networks with $f = 3$, has better reliability characteristics (Curve C) than the DLCN for the same number of transmitters. The full-duplex DLCN (or DDLCN[5]) with bypassing and the bidirectional lookahead ring[20] has reliability curves D and E respectively. The full-duplex DLCN is superior to the lookahead network,

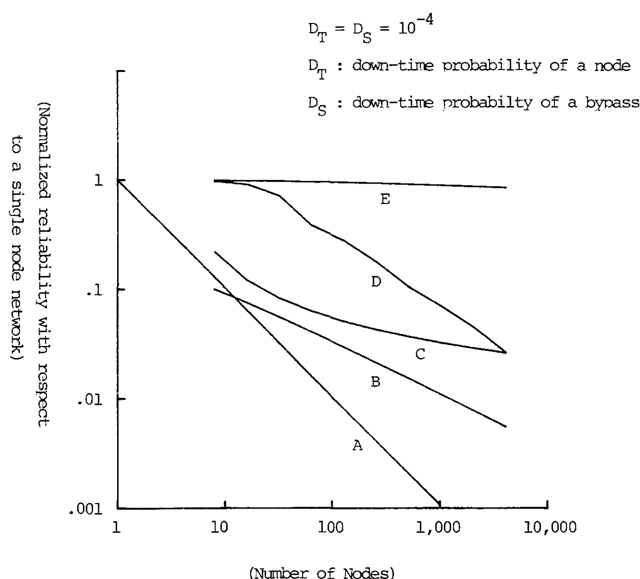


Figure 8—Reliability curves for A = ring without bypass, B = DLCN with bypass, C = lookahead ring, D = DLCN with bypass, E = bidirectional lookahead ring

but a variation of the lookahead network is nearly perfect for the selected values of D_s and D_t. This shows that by using two transmitters per node, the reliability of the lookahead network can be made better than the DDLCN.[5]

The cost of the lookahead network, where the PROPAGATE line is sent on the same line as the GENERATE and is then saved in a flip-flop, is comparable to the full-duplex DLCN, requiring $4*n$-2 high-speed lines between n processors, whereas the DLCN requires $2*n$ high-speed lines. This cost is reduced by time-multiplexing the PROPAGATE signal on the GENERATE line, as discussed in the last section. A similar network to the lookahead network where all nodes, including nonleaf nodes, are processors has been described by Lipovski.[6] That network would actually have slightly lower cost than a full-duplex DLCN. The protocols developed in this paper would apply to that network without modification. Finally, the ease of extendability is comparable with the other networks.

7. CONCLUSION

A network has been presented that is upward compatible to both the Ethernet and the DLCN networks. This lookahead network features faster operation, which may become a factor in very-high-speed networks that use light pipes; and it features the ability to prune faults and to run different protocols in different subtrees, or the same protocol in different subtrees, to increase the throughput. It can evaluate multiple-level priorities; therefore, the modularity, expandability, ease of diagnosis, reliability, and reconfigurability of the lookahead network make us believe that it is the best network so far defined for local-area, or establishment, networks.

ACKNOWLEDGMENTS

We wish to thank Jim Browne for his encouragement and insights. We acknowledge the financial support from a BMD grant for the early work done on this paper, and we would like to thank the staff of the BMD Advanced Technology Center for their suggestions. We also acknowledge a great deal of practical advice from the IBM staff at Austin, and the advice and encouragement of Fred May in particular. The preparation of this paper was also supported by a grant from IBM. However, the opinions expressed in this paper do not necessarily reflect the policies and strategies pursued by the U.S. Army or by IBM.

REFERENCES

1. Metcalfe, Robert M., and David R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM,* 19(1976), pp. 395–404.
2. Liu, Ming T., and Cecil C. Reams. "The Design of the Distributed Loop Computer Network." International Computer Symposium 1975, Taipei (Taiwan), August 1975, pp. 273–283.
3. Farmer, W. D., and E. E. Newhall. "An Experimental Distributed Switching System to Handle High-Speed Aperiodic Computer Traffic." *Proc. ACM Symposium on Problems on the Optimization of Data Communication Systems,* October 13–16, 1969.
4. Pierce, J. R. "Network for Block Switching of Data," *Bell System Technical Journal,* 51 (1972), pp. 1133–1145.
5. Wolf, J. J., and M. T. Liu. "A Distributed Double-Loop Computer Network (DDLCN)." 7th Texas Conference on Computing Systems, November 1978, pp. 6-19–6-34.
6. Lipovski, G. J. "An Organization for Optical Linkages Between Integrated Circuits," *AFIPS, Proceedings of the National Computer Conference* (Vol. 46), 1977, pp. 227–236.
7. Goyal, A., and G. J. Lipovski. "Reconfigurable Hierarchical Rings." Distributed Data Acquisition, Computing, and Control Symposium, Miami Beach, December 1980, pp. 3–10.
8. Goyal, A., and G. J. Lipovski. "Scheduling on a Light Pipe Simplex Ring," The 1st International Conference on Distributed Computing Systems, Huntsville, Alabama, October 1979, pp. 116–123.
9. Tanenbaum, A. S. "Computer Networks." Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
10. Bokhari, Shahid H. Personal communication.
11. Arnold, R. G. and E. W. Page. "A Hierarchical, Restructurable Multi-Microprocessor Architecture." *Proceedings of 3rd International Symposium on Computer Architecture,* January 1976, pp. 40–46.
12. Clark, David D., K. T. Pogran, and David P. Reed. "An Introduction to Local Area Networks." *Proc. of IEEE,* 66 (1978), pp. 1497–1517.
13. Goyal, Ambuj. "Bridge Buffers and Probability of Lost Packets." Technical Report TR-5 at the Department of Electrical Engineering, University of Texas, Austin, August 1981.
14. Lipovski, G. J. "On Designing with Gray Boxes." Workshop on Computer Description Languages, Rutgers University, New Brunswick, New Jersey, 1973.
15. Chang, H. Y., E. Manning, G. Metze, "Fault Diagnosis of Digital Systems." New York: John Wiley and Sons, 1970.
16. "Interface Between Data Terminals Operating in the Packet Mode on Public Networks," CCITT Recommendation X.252. Available from D. J. Horton, Assistant Vice President, Bell Canada, 11th Floor Executive, 160 Elgin St., Ottawa, Canada ($10 a copy in advance).
17. Jensen, E. D. "The Honeywell Experimental Distributed Processor—An Overview." *Computer,* January 1978, vol. 11, pp. 28–38.
18. Motorola IC MC6854. "Advance Data Link Controller (ADLC)." Motorola Semiconductor Co., 3501 Bluestein Blvd., Austin, Texas 78721.
19. Lam, Simon S. "A Packet-Network Architecture for Local Interconnection." Conf. Record Int. *Conference on Communications,* Seattle, Washington, June 1980, pp. 39.2.1 to 39.2.6.
20. Goyal, Ambuj. "Lookahead Networks." Ph.D. dissertation, in preparation, Department of Electrical Engineering, University of Texas at Austin.

Reconfigurable multicomputer networks for very fast real-time applications

by CARL DAVIS
Ballistic Missile Defense Advanced Technology Center
Huntsville, Alabama
SVETLANA P. KARTASHEV
University of Nebraska, Lincoln
Lincoln, Nebraska
and
STEVEN I. KARTASHEV
Dynamic Computer Architecture, Inc.
Lincoln, Nebraska

ABSTRACT

This paper introduces concurrent reconfiguration techniques that perform fast reconfiguration of a multicomputer network into the following network structures: K-rooted trees, stars, and rings with selectable periods. These structures prove to be very efficient for high-speed, real-time applications. The techniques introduced are based on shift register theory and are performed by special shift registers residing in each network node and called *shift registers with variable bias.*

The technique discussed in this paper are implemented in the system with dynamic architecture that is now under construction by Dynamic Computer Architecture, Inc.

The time of the network reconfiguration equals that of one clock period, since to perform reconfiguration into a new network structure, each network node should execute only two logical operations—one-bit shift and mod 2 addition.

INTRODUCTION

Real-time systems compose a class of extremely stressing computational problems.[1,3,5] These systems may be characterized as having a sensing device that collects data, a processing device to extract information from the data, and a reactive device to make use of the information. In the case of weapons systems the problem is complicated by the extremely short times in which a system must respond as well as the large dynamic variations of data to which a processing system must be able to react. This places a premium on the processing system's having high throughput as well as a great degree of flexibility.

A typical real-time application is described in Fig 1., where the four stages indicate a sequence of processing stages performed by the system.

The main system sensor is a phased array radar that can generate a pencil beam that changes directions in a few microseconds.

The objective of the second stage is to eliminate most non-threatening targets from consideration via crude first checks. This may be done through special-purpose hardware dedicated to signal processing.

After this initial filtering operation, the radar objects that are potentially threatening are reported to a computer that initiates a separate file on each object. The computer communicates with the radar by making requests to collect additional information about every object.

Each target is then followed; and the computer computes its velocity, trajectory, current position, and other important information. After the target has been in precision track for a short interval, discrimination routines are used to discriminate among targets.

Computation Requirements

As indicated by Arnold, Berg, and Thomas,[5] each stage of Figure 1 requires the following type of processing. Since computer processing starts with Stage 2, it receives input prepared by Stage 1, which ends with the arrival of multiple data streams requiring the same operation over multiple data items. Thus, Stage 2 is characterized by the MIMD mode of operation since it is specified by several instruction streams, each of which operates over a vector of data items.

Once most of the unthreatening objects are filtered out from consideration, the objects that are left are subjected to a more detailed check. Most of the typical computations at Stage 3 are sorting and arithmetic pipelined computations. Sorting is aimed at forming classes of threatening objects, whereas pipelining is directed at quick organization of a separate data file for each object of the same class. These files are then updated with new information received from the radar.

Finally, Stage 4 is aimed at very precise computation of trajectories of several high priority objects that are still threatening after Stage 3 and at predicting the points of their interception with the killer mechanisms that are going to be launched. The most effective computations of Stage 4 are of

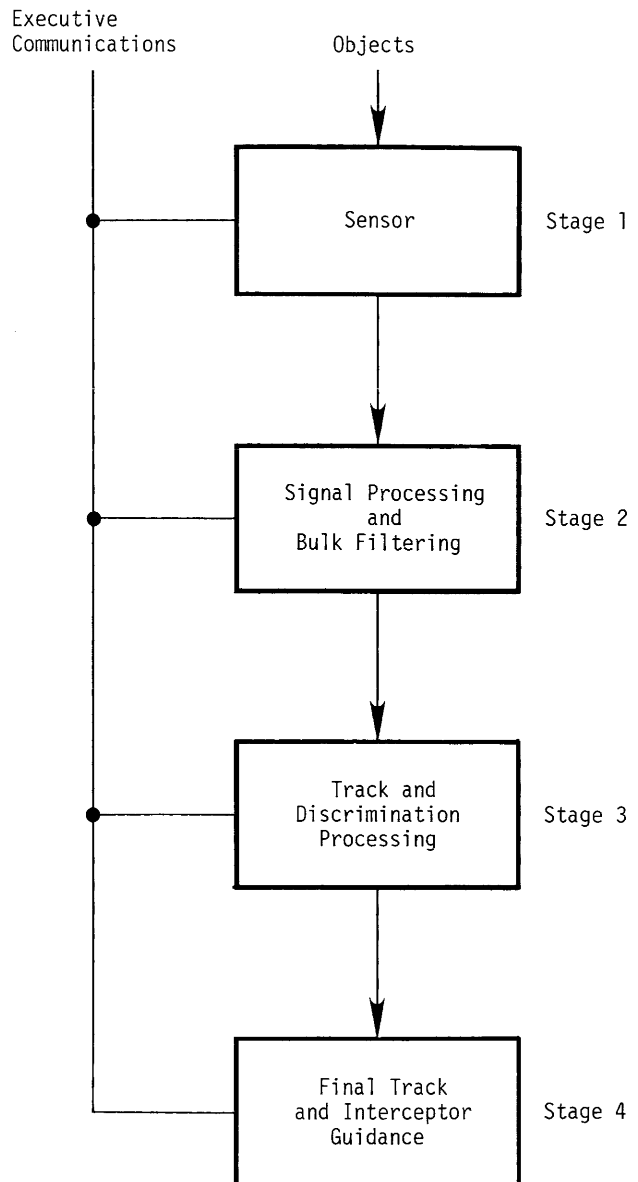


Figure 1—A typical sequence of fast radar real-time processing

an asynchronous MIMD type, performed over data words with long sizes to maintain a required level of accuracy.

Most Useful Configurations of a Multicomputer Network

As was shown by Davis and Couch,[1] a heavy computer involvement in a computational process of real-time application begins with stages 3 and 4. The nature of the application supports distributive and reconfigurable computer resources that are understood as a collection of computer elements, interconnected by an interconnection network.[2–4]

Let us show which are the most effective configurations that can be assumed by the resources.

Stage 3 involves in (a) sorting and discrimination aimed at partitioning the incoming data items into categories, and (b) pipelining aimed at performing identical computations over the items of the same category.

During execution of sorting and discrimination routines the most effective configurations of the system are either binary trees or stars. Once all the necessary classes of data words have been formed, the system must execute pipeline computations over the data words belonging to the same class. The most effective configurations for this are rings of different periods, whereby each node of a ring fetches one operand from its local memory and receives the second operand from its predecessor in the ring. The task of reconfiguration is thus reduced to forming rings with various periods in order to organize pipelined computations of various complex arithmetic expressions, requiring a variable number of arithmetic operations.

Because tracking and discrimination processing (Figure 1), realized via sorting and pipelining, are performed over continuously arriving input data words, trees, stars, and rings should coexist in the same system (Fig. 2a).

Since binary trees are characterized by two outcomes for each nonleaf node, they are useful configurations for tests with two outcomes: true, T, and false, F. If the number of true outcomes is greater than 1, then it is more cost efficient for the network to assume a star configuration.

Trees and stars can be singly or multiply rooted. For a multirooted tree or star, roots form a ring and specify the number of real-time classes that must be distinguished. Each class is specified by an m-step test procedure, where $m = 1, 2, \ldots$, and each step is described by a specific sorting algorithm aimed at distinguishing specific qualifications of the class. The tree or star structure will then contain m levels, where the root(s) is of the highest level and the leaves are of the lowest level (0). The task of the leaves is to accumulate data words with identical test qualifications.

Concurrency is understood here as the continuous acquisition of real-time data streams by the roots $R_1, R_2, \ldots, R_k$, which execute concurrent test algorithms. A tested data word is either forwarded to the true node of the next m-1 level distinguished by the test result T_i, or to the next root R_{j+1} (mod k) if the result of the test is F. Similar procedures are performed by all other lower level nodes except leaves that perform no tests and merely maintain object files, where each file stores data words with identical test outcomes.

Fig. 2(a) shows a 6-rooted 2-level binary tree configuration assumed by the multicomputer network performing concurrent 2-step sorting of continuously arriving data into six classes, R_1 thru R_6. True data words for each class are then accumulated in nodes A_1, A_2, A_3, A_4, A_5, and A_6, respectively. Suppose that the data words for class R_1, accumulated in a node A_1, are then pipelined by an arithmetic pipeline consisting of four stages (A_1, B_2, B_3, B_4) and forming a ring with period 4. Thus, we will have the coexistence of a ring, with period 4, and a tree structure in the same network. Figure 2(b) shows a 3-rooted star configuration assumed by the network that has to sort the data into tree classes, R_1, R_2, and R_3, so that a sorting is two-step and each step of sorting produces

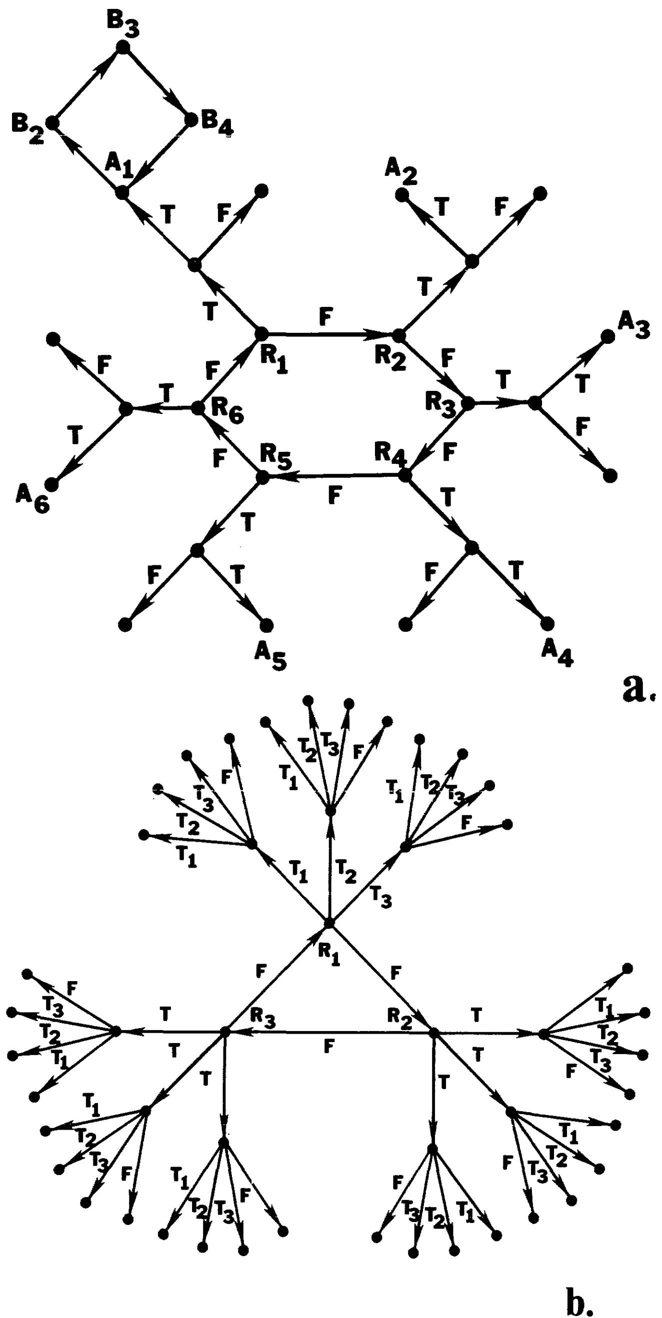


Figure 2—(a) Co-residence of six-rooted binary tree and four-period ring, (b) Three-rooted star

three true outcomes, T_1, T_2 and T_3, and one false outcome, F.

Grouping of data in accordance with the results of the tests is performed by the leaf nodes of the star. The first step is described by the algorithms executed by the roots R_1, R_2, and R_3, and the second step is executed by the nodes of the first level in the hierarchy. The leaves are the nodes of the 0th level and they execute no sorting algorithms.

As follows from this analysis, a multicomputer network performing very fast reconfigurations into various types of k-rooted trees, stars (where $k \geq 1$), and rings becomes extremely cost-effective in sorting and evaluation of data words that specify k real-time classes. Each such class can be described by d different data files, so that each data file contains data words with the same test characteristics.

Paper Composition

This paper introduces concurrent reconfiguration techniques that perform fast reconfigurations of a multicomputer network into the following network structures: k-rooted trees, stars, and rings with selectable periods. These structures have proven to be very efficient for high-speed real-time applications. The techniques are based on shift-register theory and are performed by special shift registers residing in each network node and called shift registers with variable bias (SRVB's). These registers were first introduced in Kartashev and Kartashev (1981).[15]

The time of the network reconfiguration equals that of one clock period, since to perform reconfiguration into a new network structure each network node should execute only two logical operations—one-bit shift and mod-2 addition.

DESIRABLE CHARACTERISTICS OF MULTICOMPUTER NETWORKS FOR REAL-TIME COMPUTATIONS

In order to provide a multicomputer network with very high flexibility and to reduce the amount of data traffic among its nodes, the network must be provided with the following characteristics:

C1. Minimal Time of Reconfiguration from One Structure to Another. Implementation of this requirement will allow the network to realize the maximal speed advantage to be gained by computing in a new and more suitable network structure, NS_d, in as much as it will introduce a minimal reconfiguration overhead—when no computation can be performed—in making transition $NS_f \rightarrow NS_d$ from a current network structure, NS_f, to the next network structure, NS_d.

C2. Multifunctional Node. This is conceived as the capability of each node to be connected into any of the network structures we have introduced and for trees and stars to function as a root, leaf or nonleaf.

This characteristic will eliminate all reconfiguration constraints, minimize the amount of traffic among nodes, and minimize the number of idle resources. Indeed, a programmer will be capable to balance computations among nodes, to minimize idle resources not involved in computations, and to eliminate traffic bottlenecks that might be created in the network because of dedication of its nodes.

C3. Variable Word Size of a Network Node. To increase the network flexibility each network node must be provided with the capability to change its word size. This will minimize the amount of resource interconnected into a particular network configuration and allow computation of additional programs using the same resources. The advantages of such computations are coincident with those performed by dynamic architectures in general. These have been treated extensively by Kartashev and Kartashev.[7–9]

A multicomputer network that is provided with properties C1, C2, and C3, and performs reconfigurations into the structures introduced above can be organized using the Dynamic Computer Group structure described in Kartashev and Kartashev (1980).[7] In the next section we will briefly repeat this material to make the paper self-contained.

DESCRIPTION OF THE DYNAMIC COMPUTER GROUP STRUCTURE

The Dynamic Computer Group (DC group) contains n computer elements (CE), and one monitor V with memory M(V) (Fig. 3). One CE processes h-bit words in parallel. Each CE includes an h-bit processing element (PE), an h-bit-wide memory element (ME), and a GE, equipped with small memory M(GE). Assume $h = 16$.

All functional units of the DC group (PE, GE, and V) are assembled from a universal module UM having 64 pins. Basic concepts of such a module were described elsewhere.[9]

The processor and I/O resource is connected with the memory resource by a reconfigurable memory-processor bus containing two types of connecting modules: address connecting element (ASE) and memory connecting element (MSE). The ASE elements connect each PE with all ME's and broadcast the effective memory addresses and READ and WRITE signals (w_1, w_2) from this PE to all MEs. The number of ASE elements assigned to each PE is n, the number of MEs in the resource, so that ASE_i transfers an address and two signals from a specific PE to ME_i ($i = 1, \ldots, n$).

The MSE elements are connected to each ME and exchange 16-bit bytes between this ME and the PEs. Each ME is assigned n MSEs, where n is the number of PEs in the resource, so that MSE_i exchanges 16-bit bytes between this ME and PE_i. Both the MSE and the ASE connecting elements may be implemented on a universal module considered in Kartashev and Kartashev (1979).[9]

In the DC group the memory-processor bus may broadcast two types of information between any pair of PE (or GE) and ME:

1. an effective memory address from PE to ME, and
2. a 16-bit data byte between PE and ME.

Indeed, if PE_i performs a 16-bit information exchange with ME_j, it uses its local connecting element ASE_j to broadcast the effective memory address E from PE_i to ME_j. A 16-bit data byte to be written in or read out from this address is then broadcast via local connecting element MSE_i attached to ME_j. Therefore, two connecting elements, ASE_j, one local with PE_i and MSE_i, one local with ME_j, accomplish a complete 16-bit

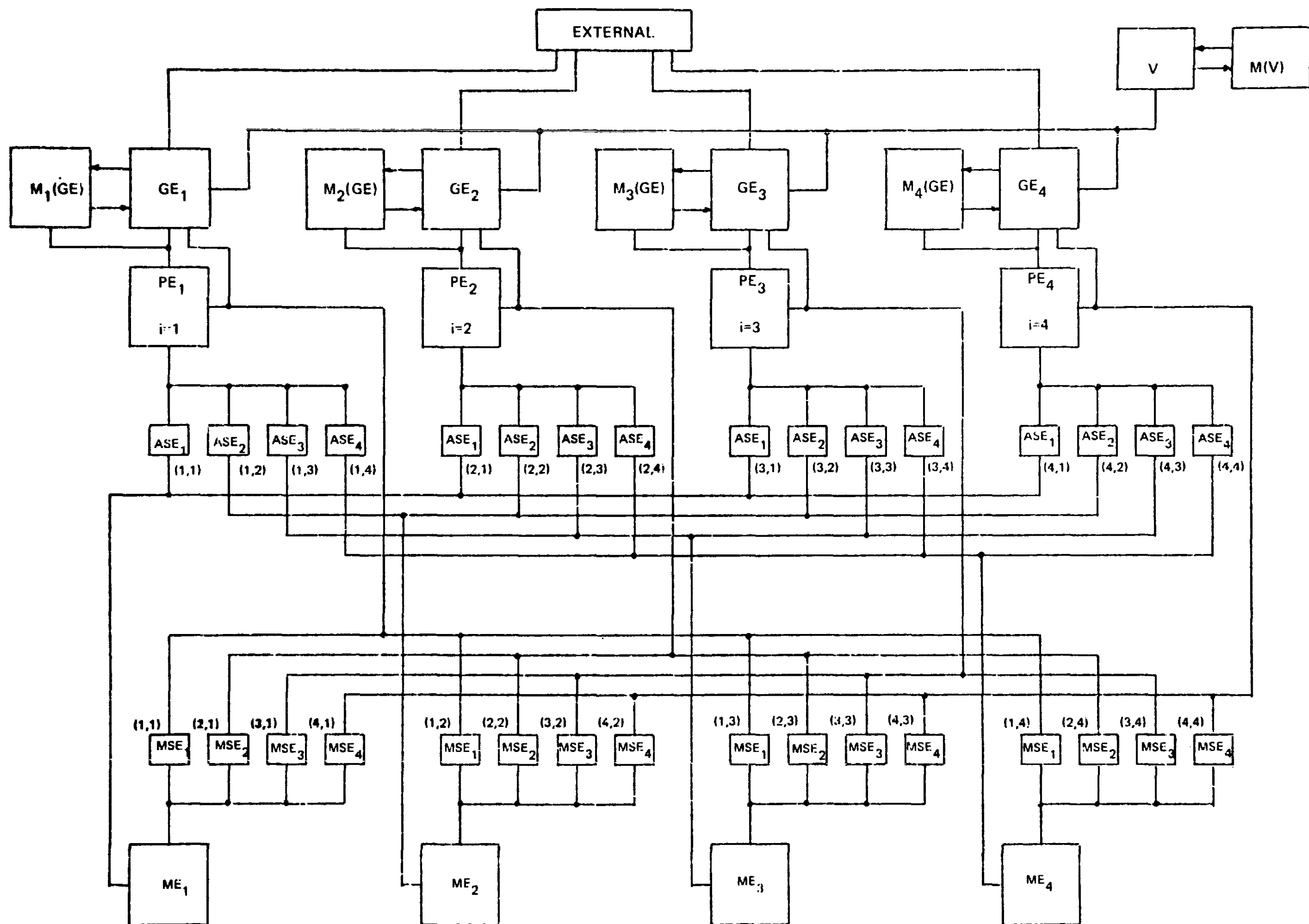


Figure 3—DC-group with four computer elements

data exchange between PE_i and ME_j, otherwise called (PE_i-ME_j) exchange.

(PE_i-ME_j) Exchange

Since this exchange may proceed in two directions—from PE_i to ME_j or from ME_j to PE_i—one has to connect these two connecting elements ASE_j and MSE_i with a two-line connection (i, j) for READ signal w_1 and WRITE signal w_2, respectively. Using the w_1 line of this connection, the ASE_j element belonging to PE_i activates the MSE_i element belonging to ME_j into a 16-bit byte broadcast from ME_j to PE_i. Using the w_2 line, the same ASE_j element that broadcasts the address from PE_i to ME_j activates the MSE_i element into a 16-bit byte broadcast from PE_i to ME_j. It then follows that to synchronize address and 16-bit byte broadcasts, such (i, j) connections have to be established for all pairs of ASE and MSE connecting elements that may be formed in the DC group.

Therefore, each PE_i-ME_j exchange is reduced to execution of the following phases of information broadcasts in the memory-processor bus:

1. Address phase: PE_i sends the E address to ME_j via its local ASE_j;
2. Synchronization phase: ASE_j selected by PE_i activates MSE_i local to ME_j into 16-bit byte broadcast; and
3. Data phase: MSE_i transfers a 16-bit data byte between PE_i and ME_j.

(PE_i-PE_j) Exchange

The memory-processor bus supports individual data exchanges (PE_i-PE_j) between different PEs through the path made of a pair of two MSE's assigned to one ME that may pass a 16-bit byte in the opposite directions. To activate such a path, the first PE_i should activate the (PE_i-ME_j) exchange with one signal (READ or WRITE) and the second PE_j should activate local (PE_j-ME_j) exchange with an opposite signal (WRITE or READ).

Indeed, when PE_i activates the (PE_i-ME_j) exchange, it selects the MSE_i of ME_j and ASE_j of PE_i. When PE_j activates the PE_j-ME_j exchange, it selects MSE_j of ME_j and ASE_j of PE_j. Further, two activated MSE belong to the same ME and are activated in the opposite directions. Since MSE_i is at-

tached to PE_i and MSE_j is attached to PE_j, these two connecting elements form a data path between PE_j and PE_j.

In activation of (PE-ME) exchanges, each PE executes the same sequence of phases: address, synchronization, and data. The only distinction is that the address phase for this exchange is simplified, since each PE sends only w signals for activating the (i, j) or (j, j) connections, respectively. None of the PE send the E address.

(ME_i-ME_j) Exchange

The memory-processor bus supports data exchanges between a pair of different ME's. To speed up this exchange, it is assumed that a generation of addresses for both ME_i and ME_j is performed by PE_i that is local with ME_i. This also releases PE_j local with ME_j from address generation and allows it to work autonomously on other computations requiring no memory access when its local ME is accessed by PE_i.

Since PE_i must generate two E addresses for ME_i and ME_j, respectively, and send them via the same address channel A with the width E, it generates only one E address at a time. This will lead to the two time intervals required for (ME_i-ME_j) exchange. During the first interval, PE_i sends the E address and w signal to ME_j and opens the MSE_i connecting element local with ME_j for a data broadcast in one direction. During the second interval, PE_i sends another E address and an opposite w to its local ME_i, and opens the MSE_i connecting element local with ME_i for a broadcast in the opposite direction.

RECONFIGURATION OF THE DC GROUP INTO A MULTICOMPUTER NETWORK

A multicomputer network is understood as a collection of CE's interconnected through the bus structure introduced in the previous section. Each CE is a node N in the network. To organize data broadcast among a pair of nodes, N and N*, interconnected with this bus, it is sufficient for the network node N to generate the position code of N*. The data path between N and N* may belong to the following types: PE-ME*, PE-PE*, ME-PE* and ME-ME*, where the first element of each pair needs the exchange and the PE and ME belong to a single CE identified with the network node N.

*Network Transition N→N**

The activation of the data path between N and N* will be denoted as transition N→N*, meaning that

1. N will generate the position code of N*;
2. N will establish a data path between N and N*;
3. A data path between N and N* can be made bidirectional—either from N to N* or from N* to N, and it can be one of the four types: PE-ME*, PE-PE*, ME-PE*, ME-ME*, where the first element belongs to node N and the second element belongs to node N*.

(Note that we assume that each node is equivalent to one CE. An extension of the results accomplished to a dynamic computer, C(k), assembled of k CE, can be done very easily by assigning the same position code to all its CEs.)

Since the bus structure is based on the idea of a cross-bar switch, the bus minimizes communication delays in data-word propagation. Advantages of this bus for very fast real-time applications were treated extensively in Kartashev and Kartashev (1980).[7]

To minimize the time of reconfiguration, it is reasonable to assume that for each network structure, a rule of succession N→N* will be maintained during reconfiguration such that each N will have the least possible number of immediate successors of N* in this structure. Indeed, since each reconfiguration N→N* will take one time interval T, it will take time pT to establish p data paths between N and its p successors. Therefore, the minimal number of successors for each N in the given network structure will mean the minimal reconfiguration time that must be spent to establish this structure. For rings, the rule of minimal number of successors is trivial, since each N in a ring has a unique successor N*.

For trees and stars, to hold this rule during reconfiguration requires that the direction of succession be maintained from the leaves to the root(s). This will transform trees and stars into *single successor structures*, since each N will have only one successor N* in this structure. Therefore, it takes only one time interval to establish transition N→N*, if one derives concurrent reconfiguration algorithms whereby all transitions N→N* are established concurrently. For this case, the entire network reconfiguration into a new network structure will take the time T of only one network transition.

APPLICATION OF SHIFT-REGISTER THEORY

In this paper trees, stars, and rings will be generated with the use of concurrent reconfiguration algorithms based on the shift-register theory. Since trees, stars, and rings are single-successor structures, the entire time T of the network reconfiguration into any of these structures will take the time of one network transition N→N*.

The contribution of the shift-register theory to reconfiguration techniques that are developed is that the entire time T of network reconfiguration will take the time of two logical operations executed sequentially: 1-bit shift and mod-2 addition. This is possible because the following technique is used to activate each transition N→N* belonging to a network structure.

Assume that each network node N is provided with a special shift-register of length n that stores its position code N, where n is the size of the code (Figure 4). Suppose that in the given network structure, N should be connected with node N* via a PE-PE*, PE-ME*, ME-ME*, or ME-PE* data path. Then for each type of communication between N and N*, N generates position code N* using a left-shifted shift-register that generates N* as follows:

$$N^* = 1[N] \oplus B, \quad (1)$$

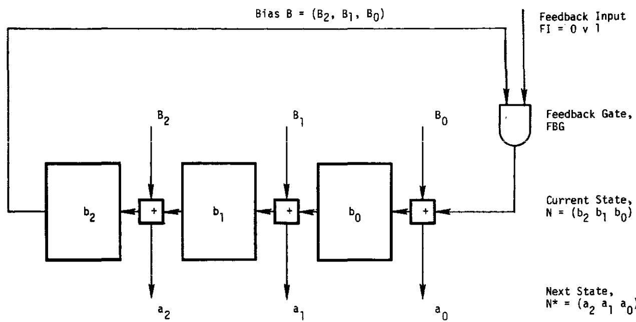


Figure 4—Three-bit shift-register with variable bias $B = (B_2, B_1, B_0)$

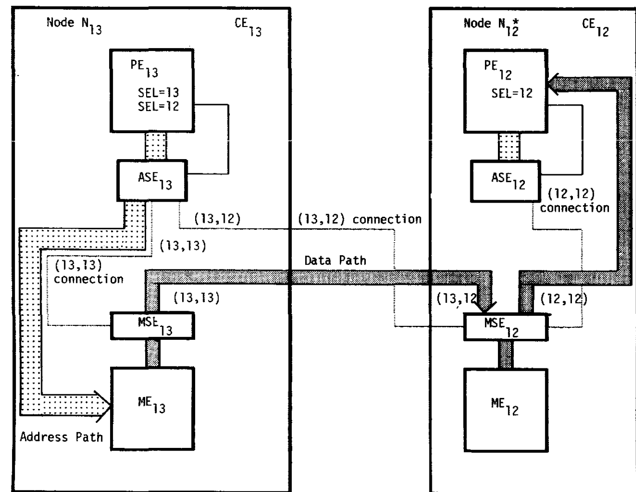


Figure 5—The ME_{13}-PE_{12} data exchange between network nodes N_{13} and N_{12}

where $1[N]$ is a one-bit shift of N to the left and B is an n-bit reconfiguration constant brought with the reconfiguration instruction to all network nodes that are requested for reconfiguration. Reconfiguration constant B is called *bias* and the shift register of Figure 4 is called a *shift-register with variable bias* (SRVB).

Suppose that $N = 1101 = 13$, $B = 0111 = 7$. This gives $N^* = 1[1101] \oplus 0111 = 1011 \oplus 0111 = 1100$. Therefore, network node N_{13} generates position code $N^* = 1100 = 12$ of its successor in the given network structure; $N^* = 12$ is used to activate a given data path (PE_{13}–PE_{12} or PE_{13}–ME_{12} or ME_{13}–ME_{12} or ME_{13}–PE_{12}) between CE_{13} and CE_{12}, identified with nodes N_{13} and N^*_{12}. Figure 5 shows the activation of the ME_{13}-PE^*_{12} data path between nodes N_{13} and N_{12}.

The gate FBG in Figure 4 is called a *feedback gate*. Introduction of the FBG gate allows a shift register, SRVB, to perform two types of shifts: (a) *circular* $1[N]_1$, when *feedback input* $FI = 1$; and (b) *noncircular* $1[N]_0$, when $FI = 0$.

If $FI = 1$, concurrent shift registers generate rings; if $FI = 0$, they generate trees. Certainly, the meaning of FI is brought to each node with the reconfiguration instruction.

However, different network structures depend not only on the value of bias B, and feedback input FI, but also on the type of the SRVB activated in each node.

To this end SRVB can be *single* and *composite*. A single SRVB has a unique feedback gate (FBG), which connects its most significant bit (MSB) with its least significant bit (LSB). A composite SRVB is formed from k ($k > 1$) single shift-registers each having a unique feedback gate, FBG_i. Therefore, a composite shift-register has k feedback gates where $k > 1$. For instance, Figure 6 shows a composite shift-register with three feedback gates, FBG_1, FBG_2, and FBG_3.

Generally, in a shift-register with variable bias, each bit can broadcast its value via one of two alternative paths: (a) a unique *shift-path* by which it is shifted left to the next more significant bit and (b) a unique *feedback path* by which it is sent right to some less significant bit (Fig. 7).

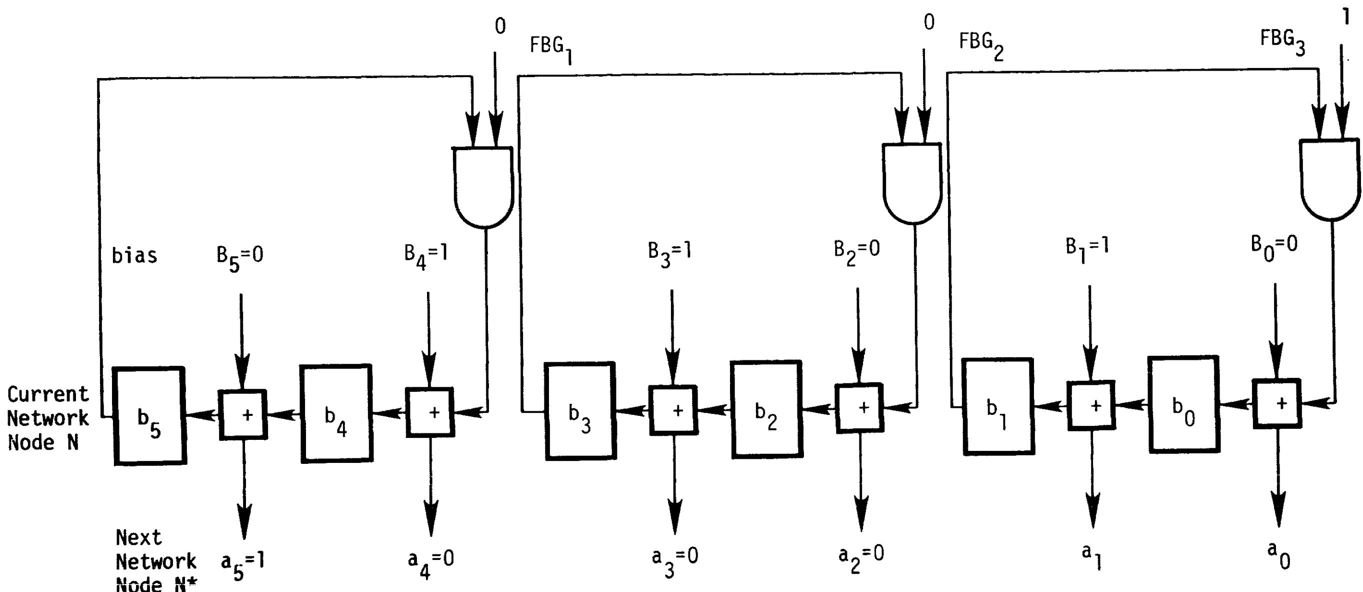


Figure 6—Composite SRVB with three feed-back gates: FBG_1, FBG_2 and FBG_3

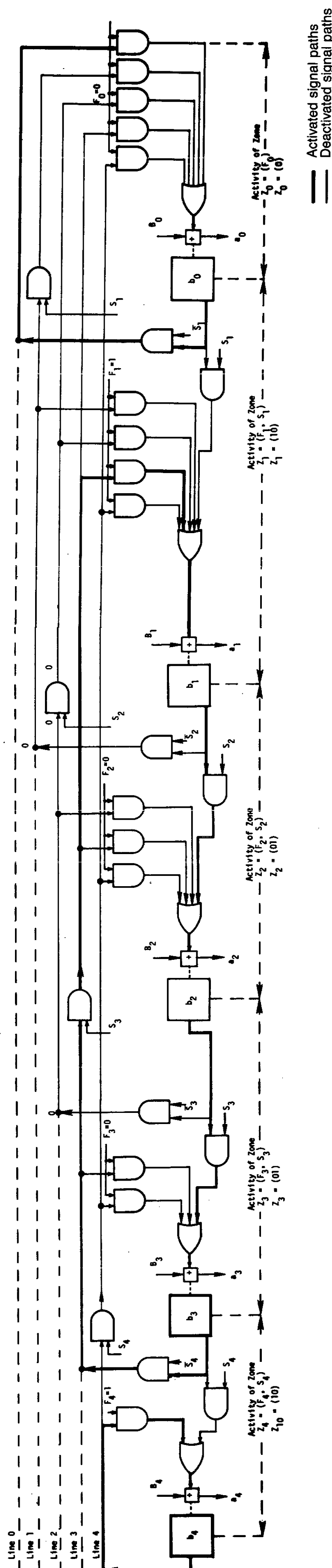
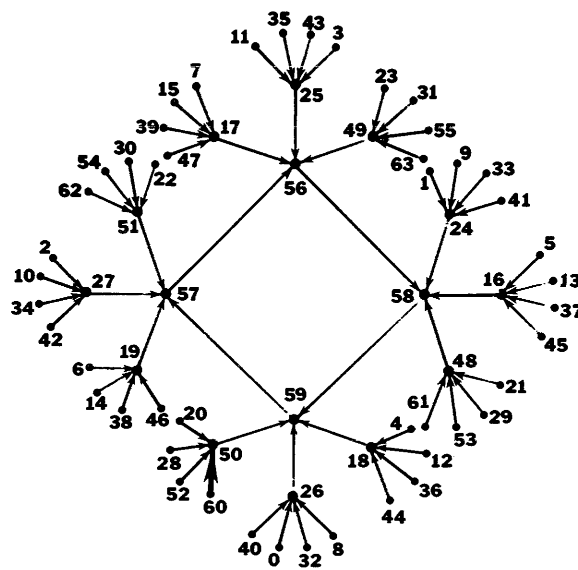


Figure 7—Reconfiguration of the SRVB into composite format $[1, 3]_1 \times [1]_0$

For instance, in Figure 6, each bit b_5, b_3, and b_1 broadcasts its value via the feedback path; the remaining bits activate their shift paths to the next more significant bits. Activation of either a shift or a feedback path for each bit can be made by a special reconfiguration code (RC) stored in the reconfiguration instruction that performs reconfiguration into a given network structure. This instruction also brings to each node the same B that forms the position code of the CE* identified with the N* that succeeds N in a given network structure. The same B received by the PE of N is conceived of as an address of the instruction stored in a local ME that initiates a subroutine of communication between nodes N and N*.

For instance, if the reconfiguration instruction stores B = 011010 and reconfigures the SRVB of each network node N into a composite one shown in Figure 6, then the network structure formed is as shown in Figure 8. As seen, it consists of a 4-rooted star. For each N, identified with CE, its PE uses B = 011010 as the base address (direct or indirect) of a subroutine fetched from local ME that shows what type(s) of exchange between N and N* must be assumed during execution of the algorithm in the given network structure.



N = 60 = 11 ¦ 11 ¦ 00   B = 01 ¦ 10 ¦ 10

$NP_1 = 11$ $NP_2 = 11$ $NP_3 = 00$ $BP_1 = 01$ $BP_2 = 10$ $BP_3 = 10$

$N^* = NP_1^* \, NP_2^* \, NP_3^*$

$NP_1^* = 1[11]_0 \oplus 01 = 10 \oplus 01 = 11$

$NP_2^* = 1[11]_0 \oplus 10 = 10 \oplus 10 = 00$

$NP_3^* = 1[00]_1 \oplus 10 = 00 \oplus 10 = 10$

$N^* = 110010 = 50$

Figure 8—The 4-rooted star generated by the composite SRVB of Figure 6

As an example, let us show how the shift-register of Figure 6 forms a unique successor N^* of the position code N stored in it. As seen from Figure 6, a composite shift-register partitions each position code N stored in it into three codes, NP_1, NP_2, and NP_3, where NP_1 consists of bits b_5, b_4 ($NP_1 = b_5b_4$); $NP_2 = b_3b_2$; $NP_3 = b_1b_0$; that is, $N = NP_1NP_2NP_3$. The B is also partitioned into $BP_1 = B_5B_4$; $BP_2 = B_3B_2$ and $BP_3 = B_1B_0$. Thus, each successor N^* also consists of three portions: $NP_1^* = a_5a_4$, $NP_2^* = a_3a_2$, $NP_3^* = a_1a_0$; thus, $N^* = NP_1^*NP_2^*NP_3^*$. The shift-register rule $N^* = 1[N] + B$ is applied to each code, NP_i, to give NP_i^* ($i = 1, 2, 3$), where NP_1^* and NP_2^* are obtained via noncircular shifts and NP_3^* is obtained via a circular shift. If $N = 60 = 111100$, then $NP_1 = 11$, $NP_2 = 11$, and $NP_3 = 00$ (Fig. 8); for the bias $B = 011010$, its codes are $BP_1 = 01$; $BP_2 = 10$; $BP_3 = 10$. Therefore, the shift register in node N_{60} generates the following successor $N^* = NP_1^*NP_2^*NP_3^*$:

$$NP_1^* = 1[11]_0 \oplus 01 = 10 \oplus 01 = 11$$
$$NP_2^* = 1[11]_0 \oplus 10 = 10 \oplus 10 = 00$$
$$NP_3^* = 1[00]_1 \oplus 10 = 00 \oplus 10 = 10.$$

Therefore, $N^* = 110010 = 50$. (In Figure 8 this transition is shown with a thick arrow.) Similarly, one can obtain any other single successor N^* of the given node N.

As shown, reconfiguration into the structure of Figure 8 is performed during the time of one 1-bit shift and mod-2 addition executed concurrently by all the network nodes that receive the same $B = 011010$ and the same RC to reconfigure each SRVB into the composite register shown in Figure 6.

CONTRIBUTION TO THE ONGOING RESEARCH

The contribution of the reported research to the current state of the art on network reconfiguration is twofold:

1. Original, simple, and elegant techniques of analysis and synthesis on network reconfiguration into the structures that have proven to be convenient for a large class of real-time computational and control algorithms. The time for such reconfigurations approaches the theoretically minimal boundary.
2. A shift register theory described elsewhere[10–14] is further expanded as follows.

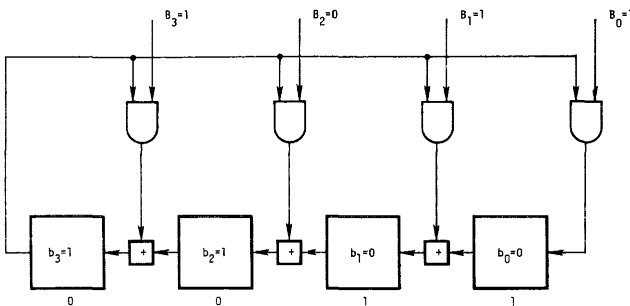


Figure 9—Four-bit linear shift-register

In the literature the shift-register studied is shown in Figure 9. Here each gate fed with B_i means connection if $B_i = 1$ and disconnection if $B_i = 0$. Thus $B = (B_{n-1}, \ldots, B_0)$ is conceived of as the same bias as was introduced earlier for the SRVB shown in Figure 4. The difference between these two registers is that Fig. 9 shows a *linear shift-register* that broadcasts to each mod-2 adder the meaning of its MSB, provided $B_i = 1$. In the linear shift-register each next state N^* generated can be obtained via matrix multiplication $N^* = N \cdot A$ where N is a current state stored in bits $b_{n-1}, b_{n-2}, \ldots, b_0$, and A is the canonical shift-register matrix

$$A = \begin{vmatrix} B_{n-1}, & B_{n-2}, & B_{n-3}, & \ldots, B_2, & B_1, & B_0 \\ 1 & 0 & 0 & \ldots 0 & 0 & 0 \\ 0 & 1 & 0 & \ldots 0 & 0 & 0 \\ 0 & 0 & 1 & \ldots 0 & 0 & 0 \\ & \cdot & & & \cdot & \\ & \cdot & & \ldots & \cdot & \\ & \cdot & & & \cdot & \\ 0 & 0 & 0 & \ldots 1 & 0 & 0 \\ 0 & 0 & 0 & \ldots 0 & 1 & 0 \end{vmatrix}$$

For instance, if $B = 1011$ and current state $N = 1100$, then the next state N^* to be generated by linear shift-register is

$$N^* = 1100 \begin{vmatrix} 1011 \\ 1000 \\ 0100 \\ 0010 \end{vmatrix} = 0011.$$

On the other hand, the SRVB having the same N and B generates the following N^*:

$$N^* = 1[1100]_1 \oplus 1011 = 1001 \oplus 1011 = 0010.$$

(We assume that the SRVB performs circular shift, since this was the condition applied to the linear shift-register.)

As follows, an SRVB and a linear shift-register generate different next states for the same current state and bias B, inasmuch as in the SRVB each mod-2 adder receives bit B_i, but in a linear shift-register each mod-2 adder receives the MSB of the register if $B_i = 1$. As a result, different network structures are generated by these two types of shift-registers.

In particular, a linear shift-register is essentially incapable of generating stars. It can generate only rings and trees, whereas the SRVB can generate rings, trees, and stars.

Another fundamental peculiarity of a linear shift-register is that it always generates a next state $N^* = 0000$ if a current state $N = 0000$, since $N^* = 0 \cdot A = 0$. This means that 0 always generates a cycle of period 1, no matter what B is fed to it. On the other hand, SRVB maps 0 onto B, that is, if $N = 0$ it is succeeded by $N^* = B$. Therefore, if $B \neq 0$, then 0 belongs to a network structure other than a cycle of period 1. For instance, for the SRVB shown in Figure 10(a) if $B = 0111$, and $FI = 1$, then 0 belongs to the first ring of period 8 shown in Figure 10(b). If this shift-register stores $N = 0101$, then it generates the second ring shown in Figure 10(b).

As a matter of fact, this network structure cannot be obtained with 4-bit linear shift-registers no matter what B is

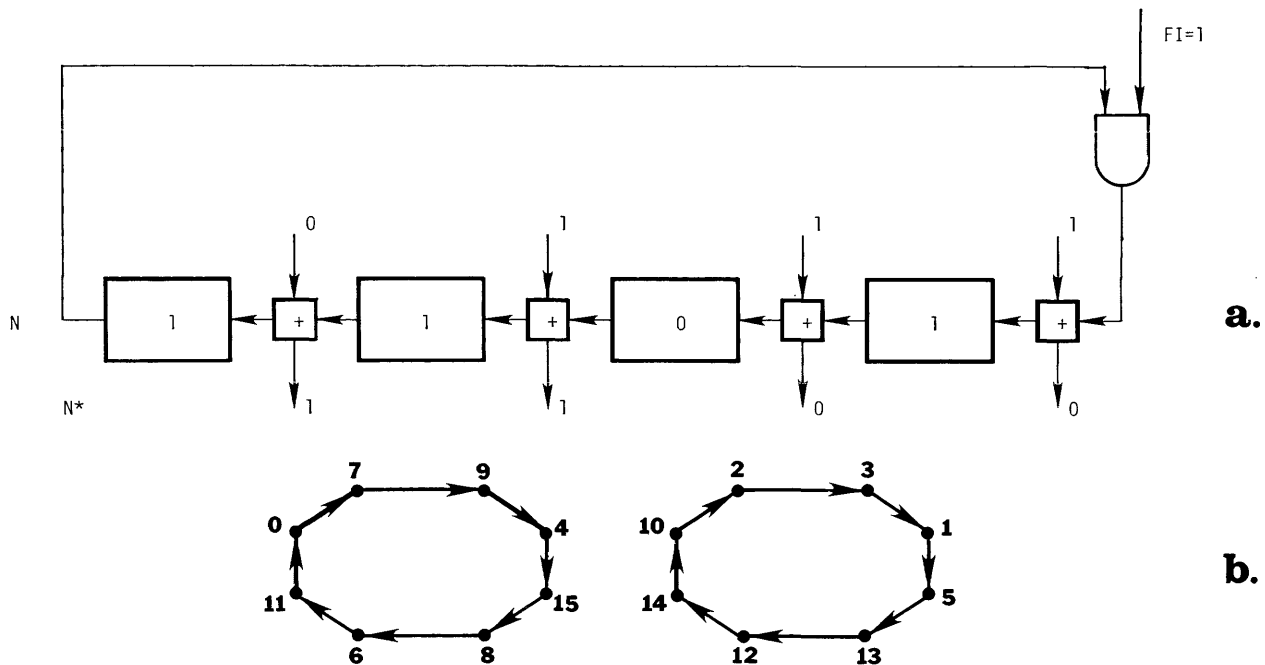


Figure 10—Four-bit circular SRVB and the ring structure generated with it

selected, since linear shift-registers always map 0 onto 0 and the remaining 15 nodes cannot be formed into two rings of period 8 each.

As follows from this, the network structures generated by SRVB and linear shift-registers are not equivalent. Furthermore, a fundamental drawback of a linear shift-register is that the techniques for finding the network structures that can be generated are very laborious and complex, since they are based on finding the periods of polynomials over a Galois field.[10–13] The complexity of these techniques grows exponentially with an increase in n, the number of bits in a shift-register. However, for complex multicomputer networks having a large number of nodes, the size n of a code that identifies each node may become significant ($n = 10$ and more). Thus, it becomes prohibitively difficult to utilize elegant results of linear shift-register theory to analyze different rings and trees that may be generated in an n-dimensional binary space.

On the other hand, all the network structures generated by SRVB can be described with very simple formulas that can be used by the programmer performing various reconfigurations in the multicomputer networks. Another attractive feature is that the complexity of these techniques remains constant and does not depend on n, the size of the position code N. Thus, the analysis techniques for describing network structures generated by SRVB are applicable to complex multicomputer networks, inasmuch as they allow obtaining simple and fast reconfiguration algorithms and simple formulas of various network structures that can be generated in the network.

It should be noted that the only area of equivalence among linear shift-register and SRVB is when $B = 0$. If both registers perform noncircular shift (whereby their MSB are not sent to LSB) and receive bias $B = 0$, they generate the same binary tree (Fig. 11(a, b, c)). If both registers perform circular shift and receive bias $B = 0$, they are transformed into the same circulating shift-register that was extensively studied in the literature[13] (Figure 11(d, e)).

PROBLEMS OF NETWORK ANALYSIS AND SYNTHESIS

The following problems are of importance for the reconfigurable networks generated by the SRVB's:

Network analysis. Given: a type of SRVB and a B. Find the network structure that is generated with this SRVB.

Network synthesis. Given: a network structure. Find the SRVB and B that can generate this structure.

It should be noted that the network analysis and synthesis are complementary inasmuch as the solution of the analysis problem provides an insight into all possible network structures that can be obtained. This will give a programmer an invaluable answer on how many structures are nonisomorphic and what biases must be selected to generate nonisomorphic structures only.

Thus, the solution of the analysis problem will lead to a minimization of the total memory space required to store a table of different network structures and the different SRVB and biases that can generate them.

On the other hand, solution of the synthesis problem is of extreme practical necessity to a programmer, because what is given to him or her is a particular network structure obtained from the analysis of a complex algorithm. It is the task of the programmer to reconfigure a multicomputer network into this

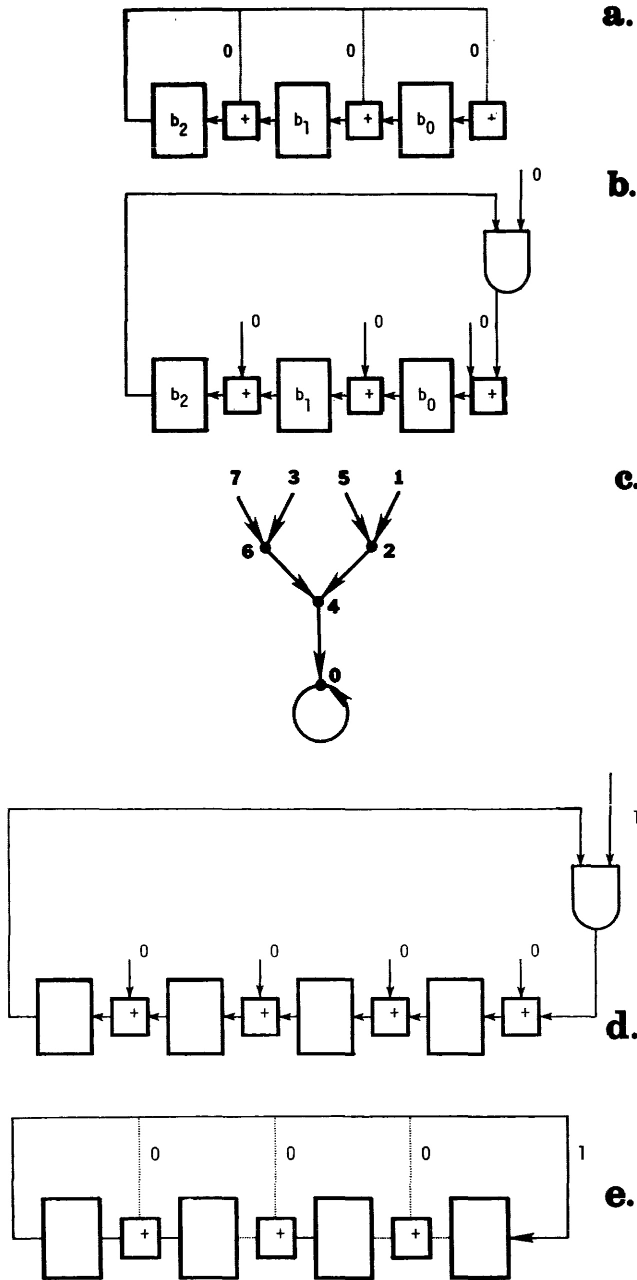


Figure 11—The area of equivalence among SRVB and linear shift-registers: (a) Linear noncircular shift-register with B = 0; (b) Noncircular SRVB with B = 0; (c) Three-level tree generated by noncircular linear shift-register and noncircular SRVB receiving bias B = 0; (d) Circular SRVB receiving bias B = 0; (e) Circulating (linear) shift-register (receiving bias B = 0)

structure. To do so he or she must select the type of SRVB and the B that generate this structure. Thus, the programmer needs the solution of the synthesis problem.

CONCURRENT ALGORITHMS OF NETWORK RECONFIGURATION

As was shown above for each N→N* transition, one can activate four types of data exchanges (PE-ME*, PE-PE*, ME-PE*, ME-ME*). Therefore, each exchange can be identified with a 3-bit code of data exchange (COE), in which the addition of one bit beyond the minimal 2-bit COE required to encode four data exchanges is caused by the fact that 0 words cannot be used since 0 means absence of exchange in the reconfiguration instruction that performs the network reconfiguration.

Global and Local Data Exchanges

Reconfiguration of the network is performed by a special reconfiguration instruction (RIN) that may have two modifications—global and local.

If all the transitions N→N* maintain the same type of data exchange in the network structure obtained with the RIN, then the COE should be stored in the RIN. For this case, RIN executes its *global modification* consisting of two steps:

Step 1. Generate position code N* of the N→N* transition where N* succeeds N in the given network structure.
Step 2. Activate global exchange specified by the meaning of COE code stored in the instruction.

However, during the existence of the given network structure, some of its transitions may maintain different local exchanges, or one transition may be described by a sequence of data exchanges that it will assume while the network keeps a current network structure. Thus, it is expedient for this case to store each COE or a sequence of COE's locally in the network node N that needs communication with N*.

A data exchange specified by COE stored locally in the network code N will be called *local.* Thus for this case the RIN will store no COE or will have 0 word in COE, and execute its local modification consisting of Step 1 only.

Following execution of RIN, each N starts to execute the next instruction stored locally aimed at fetching a local COE and establishing a required data exchange between N and N* as specified by this code.

Reconfiguration Instruction

If a program needs a new network structure, NS_d, for execution of its tasks, it contains a special RIN. An RIN can be executed in an array or even in a single CE.

Codes stored in the instruction RIN

RIN stores the following codes:

1. Code RR of the requested resource, which encodes the positions of computer elements requested for reconfiguration. This code is used in determining whether or not a requested resource is ready for reconfiguration.
2. A reconfiguration code RC, which reconfigures the SRVB of each requested network node N into the type that generates the network structure NS_d. (The techniques for finding RC are given in the next section.)

3. The bias B that is fed to each SRVB, reconfigured by the RC, in order to generate the position code N* that succeeds N in the given network structure. Techniques for finding B are not discussed in this paper.
4. A program user code NP that is used in the global priority analysis to be performed by the system monitor on determining the priority of the program to perform network reconfiguration.
5. For global modification of the RIN it stores the COE, provided all requested nodes will maintain the same type of exchange (PE-ME*, PE-PE*, ME-ME*, or ME-PE*).

Let us define the sizes of the codes used. The size of RR matches the number of network nodes, K, since if a node is requested for reconfiguration it corresponds to 1 in RR; otherwise it corresponds to 0 in RR. For large networks containing 64 nodes and more, to store RR will require several memory cells, inasmuch as the width of the bus in dynamic architectures is 16 bits and computer sizes are formed in 16-bit increments. Thus if RR is stored in k memory cells, to transfer this code to the system monitor takes k time intervals.

As for RC, its size is $2n - 1$, where n (the size of SRVB of each network node) is $n = \log_2 K$. The bias size is n bits; the size of the COE is 3 bits; the NP code size depends on how many programs are executed concurrently. If the network executes P programs then $NP = \log_2 P$. Therefore, the total number of bits, #(RIN), that must be assigned for all these codes is

$$\#(RIN) = K + 2\log_2 K - 1 + \log_2 K + \log_2 P + 3$$
$$= K + 3\log_2 K + \log_2 P + 2,$$

where K is the number of network nodes and P is the number of concurrently executed programs.

If $K = 32$, $P = 64$, then

$$\#(RIN) = 32 + 3\log_2 32 + \log_2 64 + 2$$
$$= 32 + 3 \cdot 5 + 6 + 2 = 55 \text{ bits.}$$

Since one instruction word takes 16 bits, to store all these bits will take 6 words and their fetch will take 6 time intervals.

Selection of the Reconfiguration Code

As follows from the above, to organize a network reconfiguration into the network structure NS_d a programmer must be capable of the correct selection of the following important codes:

1. RC, which reconfigures the SRVB into the type that can generate NS_d.
2. The bias that must be fed to this SRVB to generate NS_d.

This section will introduce a technique for selecting the RC. The techniques for selecting B are not discussed here. Before finding an actual algorithm on forming the RC, it is necessary to introduce classification on different SRVB convenient for a programmer, who will then be able to tabulate various SRVB and select RC's that can form these SRVB.

Types of SRVB shift-registers

Each SRVB can be reconfigured either into a single one or a composite one, depending on how many feedback gates are activated with the reconfiguration instruction. Since reconfiguration of an n-bit SRVB into a composite one is understood as its partitioning into p single shift-registers, each composite SRVB will be described with a so-called *arithmetic format,* $AF = [k_1, k_2, \ldots, k_p]$, where k_i is the size of each single shift-register. Obviously, $n = k_1 + k_2 + \ldots + k_p$.

For instance, $AF = [2^3]$ describes a composite shift-register (Fig. 6) that includes three single shift-registers of size 2, $n = 2 \cdot 3 = 6$.

Since the formulas that describe generated network structures depend on the number, r_i, of shift-registers having size i, it will be convenient to represent arithmetic formats in the following form: $AF = [1^{r_1}, 2^{r_2}, \ldots, n^{r_n}]$, where r_i ($r_i \geq 0$) will be called the *multiplicity* of the i-bit shift-register. Thus $n = r_1 + 2r_2 + \ldots + n \cdot r_n$.

Since each single shift-register of AF may perform either circular shift (FI = 1) or noncircular shift (FI = 0), AF may be divided into the following categories:

1. *Circular* AF_1, when all its single shift-registers perform circular shifts;
2. *Noncircular* AF_0, when all its single shift-registers perform noncircular shifts;
3. *Mixed* AF_{10}, when single shift-registers described by it perform circular and noncircular shifts.

It will be convenient to represent mixed AF_{10} as a combination of circular and noncircular AF, that is $AF_{10} = AF_1 \times AF_0$, where AF_1 includes all circular single shift-registers and AF_0 includes all noncircular ones. For instance, if $A_{10} = [3_0^1, 4_1^2, 5_1^1, 2_0^2]$, then $A_1 = [4^2, 5^1]$ and $A_0 = [3^1, 2^2]$, i.e., $A_{10} = A_1 \times A_0$.

Reconfiguration activities performed by reconfiguration code

Reconfiguration of the SRVB into any given arithmetic format will be performed with RC. RC is stored in the reconfiguration instruction and described as follows. It is a $(2n - 1)$-bit code, where n is the size of each SRVB. It consists of $(n - 1)$ 2-bit zones, Z_i, each including two bits, S_i and F_i, and one 1-bit zone, Z_0, including only one bit, F_0. Thus, $RC = (Z_{n-1}, Z_{n-2}, \ldots, Z_1, Z_0)$, where $Z_i = (S_i, F_i)$ if $i \neq 0$ and $Z_0 = F_0$ (Figure 7).

Each zone Z_i encodes, respectively, feedback and shifting paths for the two bits, b_i and b_{i-1}, of the SRVB, where b_i is more significant than b_{i-1}. For instance, in Figure 7, for $n = 5$, $RC = (Z_4, Z_3, Z_2, Z_1, Z_0)$, and zone Z_4 encodes the feedback path ending in b_4 (bit F_4) and the shift path from b_3 to b_4 (bit S_4); zone Z_3 encodes the feedback path ending in b_3 (bit F_3) and the shift path from b_2 to b_3 (bit S_3), and so on. Finally, the least significant zone, Z_0, has only the one bit F_0, which is associated with the feedback path routed to bit b_0, and no shift path from the next less significant bit, since b_0 is the LSB of the SRVB.

Therefore, for each zone, $Z_i = (F_iS_i)$, the values of F_i and S_i show what type of path is activated for every pair of consecutive bits, b_i and b_{i-1}. If $F_i = 1$, the appropriate FI is activated. This means that b_i receives circular feedback information controlled by F_i and receives no shift information controlled by S_i from the next less significant bit, b_{i-1}. If $F_i = 0$, feedback input is deactivated. This means that either bit b_i receives no feedback controlled by F_i and only shift controlled by S_i, or bit b_i receives a noncircular feedback (for trees and stars).

Bit $S_i = 1$ of zone Z_i stands for left shift from b_{i-1} to b_i and $S_i = 0$ stands for no shift from b_{i-1}. Therefore, together F_i and S_i show what type of path is activated between b_i and b_{i-1}; shift path ($S_i = 1$, $F_i = 0$) or feedback path to b_i and no shift from b_{i-1} ($S_i = 0$, $F_i = 0 \vee 1$). For $F = 0 \vee 1$, 0 means noncircular feedback, 1 means circular feedback. Thus, shift and feedback paths are mutually exclusive; for zone Z_i, if $S_i = 1$, $F_i = 0$; if $S_i = 0$, $F_i = 0$ or 1.

Since $S_i = 1$ means that bits b_i and b_{i-1} belong to the same shift register and $S_i = 0$ means that they belong to different shift registers, each $\bar{S}_i$ is sent to activate a new feedback path initiated in b_{i-1}. Likewise, each S_i is sent not only to a shift path between b_i and b_{i-1}, but also to the feedback path initiated in b_i, either to maintain both paths ($S_i = 1$) or to block unwanted transfer either of b_i to less significant bits of shift-register or of b_{i-1} to b_i ($S_i = 0$).

Example. Figure 7 shows a software controlled reconfiguration of the SRVB into mixed arithmetic format $AF_{10} = [1^1, 3^1]_1 \times [1^1]_0$. This is accomplished with

$$RC = \underset{Z_4\ Z_3\ Z_2\ Z_1\ Z_0}{10\ 01\ 01\ 10\ 0}.$$

Consider the activity of each zone $Z_i = (F_i, S_i)$ during this reconfiguration. Since $Z_4 = (F_4, S_4) = 10$, F_4 activates the feedback path ending in b_4, and $\bar{S}_4$ initiates a new feedback path from b_3. Since $S_4 = 0$, b_4 is blocked from sending its value to other bits but itself, likewise b_3 is blocked from shifting to b_4. This leads to the formation of circular 1-bit shift-register $[1^1]_1$.

For zone $Z_3 = (F_3, S_3) = 01$, $F_3 = 0$ deactivates all the feedback paths ending in b_3 and $S_3 = 1$ maintains two paths: the shift path from b_2 to b_3 and the feedback path from b_3 to other less significant bits. Similar activity is performed by zone $Z_2 = (F_2, S_2)$.

For zone $Z_1 = (F_1, S_1) = 10$, $F_1 = 1$ activates all the feedback paths ending in b_1. However, since only line 3 was activated, F_1 completes a unique feedback path initiated in b_3. This path is ended in b_1. Thus, another circular 3-bit register, $[3^1]_1$, has been formed. Therefore, bit b_0 initiates its own noncircular feedback path that ends in b_0. This results in forming $[1^1]_0$. It then follows that RC = 100101100 has reconfigured the SRVB into $[1^1, 3^1]_1 \times [1^1]_0$.

THE PROBLEM OF THE NETWORK ANALYSIS

This section will describe some nonisomorphic network structures that can be generated by single and composite SRVB. Based on this study, future reports will introduce economical tabulation techniques necessary to store information on all the nonisomorphic network structures in the system monitor that performs network reconfiguration.

In all, a general objective of this section is to give results of the following problem:

Given: B and AF. Find the network structure that is generated.

The solution of this problem will provide a programmer with information on what network structures he or she can form. On the other hand, solution of the synthesis problem will provide a programmer with an answer on how to generate a given network structure. Thus, this section dealing with the analysis of multicomputer networks acquires extreme significance for all users of such networks who would like to apply them for their computational needs in view of some attractive features that these network structures possess. These features are (a) minimal reconfiguration time, (b) very fast data exchanges, (c) multifunctional properties of each computer node, and (d) the ability of each node to change its word sizes.

Classification Among Network Structures: Overview and Examples

Before attacking a general analysis problem introduced above, one must establish a classification among the network structures that can be generated by SRVBs. Since arithmetic formats describing SRVB can be (a) single and composite, and (b) circular, noncircular, and mixed, the classification below is based on attributes (a) and (b) and is shown in Figure 12.

In all, the network structures generated by SRVB can be divided into the following categories.

1. *Single ring structures* (SRS's) generated by single SRVB with circular arithmetic formats $AF_1 = [n^1]$.
2. *Single tree structures* (STS's) generated by single SRVB with noncircular arithmetic formats, $AF_0 = [n^1]$.
3. *Composite ring structures* (CRS's) generated by composite SRVB with circular arithmetic formats, $AF_1 = [1^{r_1}, 2^{r_2}, \ldots, n^{r_n}]$.

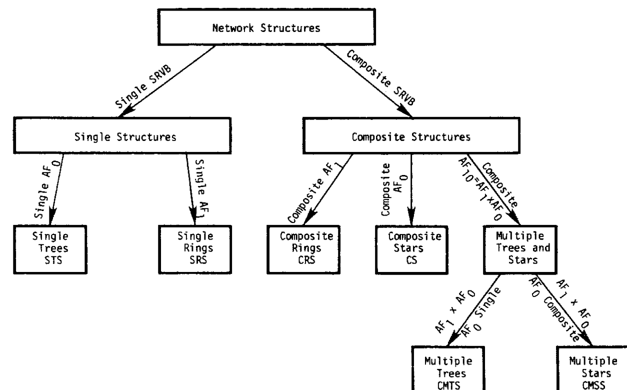


Figure 12—Classification among network structures that are generated by SRVB

4. *Composite star structures* (CSS's) generated by composite SRVB with noncircular arithmetic formats, $AF_0 = [1^{r_1}, 2^{r_2}, \ldots, n^{r_n}]$.
5. *Composite multiple tree structures* (CMTS's) generated by composite SRVB with mixed arithmetic formats, $AF_{10} = AF_1 \times AF_0$, where AF_0 describes single noncircular SRVB, $AF_{10} = [1^{r_1}, 2^{r_2}, \ldots, n^{r_n}]_1 \times [p^1]_0$.
6. *Composite multiple star structures* (CMSS's) generated by composite SRVB with mixed $AF_{10} = AF_1 \times AF_0$ where AF_0 describes composite noncircular SRVB, $AF_{10} = [1^{r_1}, 2^{r_2}, \ldots, n^{r_n}]_1 \times [1^{k_1}, 2^{k_2}, \ldots, p^{k_p}]_0$.

Before attacking a general case of arbitrary AF, consider the so-called *single network structures* produced by single shift-registers, that is, those identified by $AF = [n^1]$, circular or noncircular.

Single Network Structures

These can be of two types, rings and trees, which are specified by circular and noncircular arithmetic formats, respectively. Rings will be described first.

Single ring structures

A *single ring structure,* SRS, is a set of rings that is generated by single shift-registers available in network nodes. To define SRS means to define the following:

1. A set of periods, SP = {T}, where T is the period of a ring generated in the SRS, and
2. The number D(T) of rings having the same period, T.

Therefore, we define SRS as $SRS = \{D(T): T \in SP\}$.

Example. Let a multicomputer network have eight nodes, N_0 through N_7. Suppose that each node receives RC = 01 | 01 | 1 (that identifies circular $AF_1 = [3^1]_1$) and B = 2 = 010 (Fig. 13). The network reconfigures into the SRS consisting of two rings with periods T = 2 and T = 6, respectively; that is, for $AF_1 = [3^1]_1$, and B = 2, set of periods SP = {2,6}, D(2) = 1; D(6) = 1 and SRS = {1(2),1(6)}. Indeed, each successor N* of N is defined as $N^* = 1[N]_1 \oplus B$. For N = 101, $N^* = 1[101]_1 \oplus 010 = 011 \oplus 010 = 001$. For N = 010, $N^* = 1[010]_1 \oplus 010 = 100 \oplus 010 = 110$. For N = 011, $N^* = 1[011]_1 \oplus 010 = 110 \oplus 100$, and so on.

As follows from the above, to identify SRS's one has to identify the set of periods, SP, and the number of rings, D(T), having the period T. The techniques for finding SP and D(T) were introduced in Kartashev and Kartashev (1981).[15]

Single tree structure

As was indicated above, a single shift-register with noncircular AF generates a single-rooted binary tree. We will call a single-rooted binary tree that is generated by a single noncircular SRVB with $AF_0 = [n^1]$ a *single tree structure,* STS.

The difference between different STS's is in the relative positions of their roots, leaves, and nonleaves, although struc-

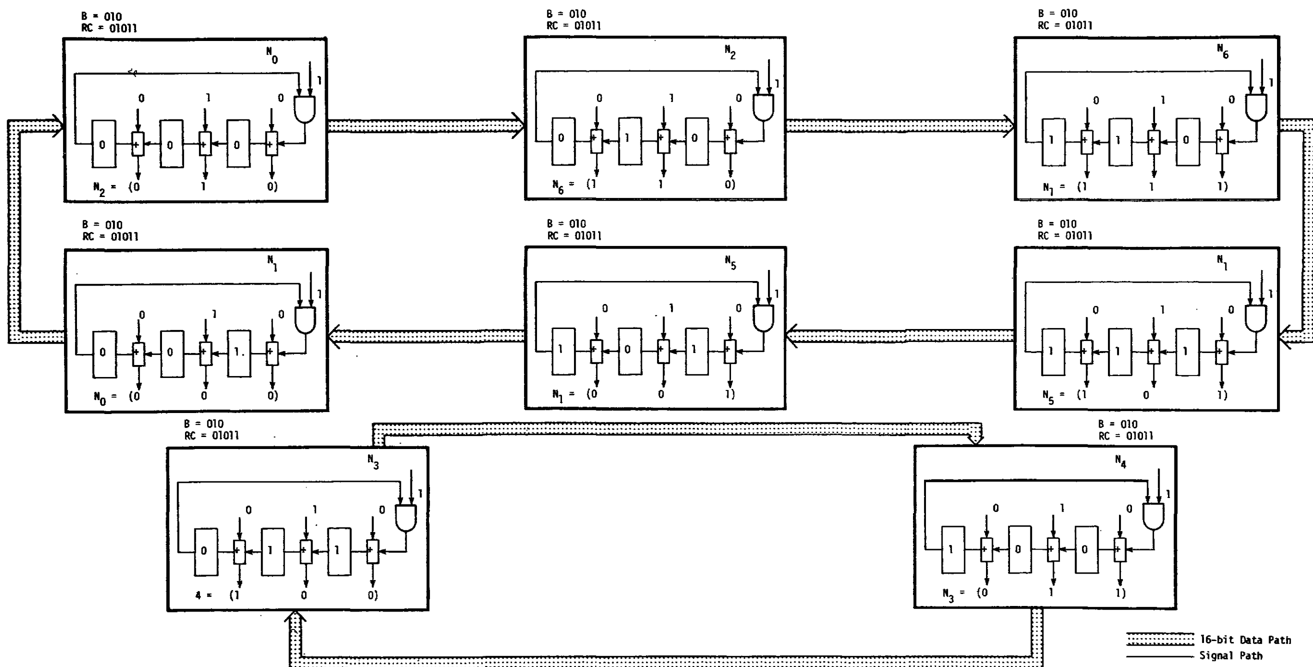


Figure 13—Single ring structure generated by 3-bit SRVB receiving B = 010 and RC = 01011

turally all STS's specified with the same arithmetic format, $AF_0 = [n^1]$, are isomorphic to each other. Figure 14 shows all the STS structures generated by the SRVB specified with $AF_0 = [3^1]$.

As was indicated earlier, to minimize the time of reconfiguration, it is assumed that in the STS each node N has only one successor, N*, inasmuch as it then takes the time of one mod-2 addition to reconfigure the network into the STS. This leads to a direction of succession from the leaves to the root, R, which then succeeds itself by forming a cycle of period 1.

For tabulation purposes, we will use the following symbols for different single tree structures: if STS is generated by a noncircular shift-register with $AF_0 = [n^1]$, then STS $= [n^1, \vec{1}]$, shows that the tree has n levels, that is, n^1, and the root R by performing $N^* = 1[R]_0 \oplus B$ will generate $N^* = R$, thus succeeding itself by forming a cycle of length 1, that is, $\vec{1}$.

For instance, all the STS's of Figure 14 are described as STS $= \{3^1, \vec{1}\}$since these trees are single, they have three levels, that is 3^1, and each root forms a cycle of length 1, that is $\vec{1}$.

Composite Network Structures: Overview and Examples

As follows from the above, composite network structures are generated by the SRVB's described by composite arithmetic formats, $AF = [1^{r_1}, 2^{r_2}, \ldots, n^{r_n}]$. As was indicated earlier, there are three types of composite arithmetic formats:

1. Circular AF_1, when all its single shift-registers perform circular shifts;
2. Noncircular AF_0, when all its single shift-registers perform noncircular shifts;
3. Mixed $AF_{10} = AF_1 \times AF_0$, when shift-registers described by it perform circular and noncircular shifts, where all circular SRVB are included in AF_1 and noncircular SRVB are included in AF_0.

Depending on what type of format AF is used, composite network structures are divided into the categories listed below. In order that the reader be capable of having an intuitive feeling about the characteristics that distinguish the categories, each category will be illustrated with one example.

1. *Composite ring structures* (CRS's) generated by composite SRVB with circular arithmetic format, AF_1 (Fig. 15). Figure 15 shows a CRS generated by SRVB's described with circular $AF_1 = [4^1, 3^1]$ and fed with bias B = 0000 | 001; that is, bias B = 0000 is fed to the 4-bit single SRVB and B = 001 is fed to the 3-bit single SRVB (Figure 16).
2. *Composite star structures* (CSS's) generated by composite SRVB with noncircular arithmetic formats, AF_0. Figure 17 shows a CSS generated by SRVB described with noncircular $AF_0 = [2^1, 3^1]$ and fed with B = 00101 (Fig. 18).
3. *Multiple tree structures* (MTS's) generated by composite SRVB described with mixed arithmetic formats, $AF_{10} = AF_1 \times AF_0$, where AF_0 is a single and noncircular format. Figures 19 and 20 show an MTS and a composite shift-register that generates this MTS, respectively. It is described by mixed format $AF_{10} = [2^1, 1^1]_1 \times [2^1]_0$. As seen, this MTS contains two four-rooted trees.
4. *Multiple star structures* (MSS's) generated by composite SRVB described with mixed arithmetic formats $AF_{10} = AF_1 \times AF_0$, where AF_0 is a composite and noncircular format. Figure 8 shows an MSS generated by

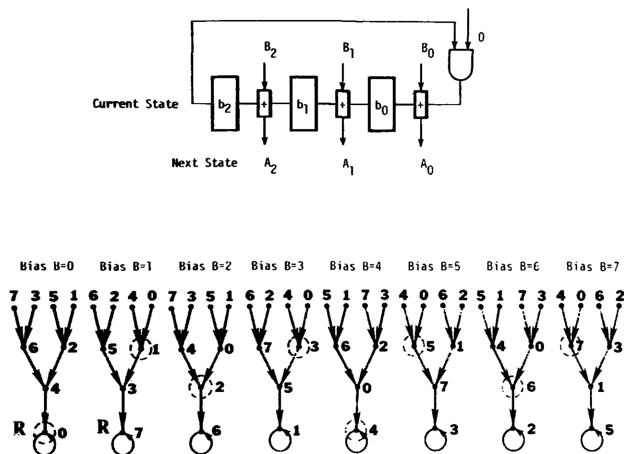


Figure 14—Single tree structure generated by 3-bit SRVB

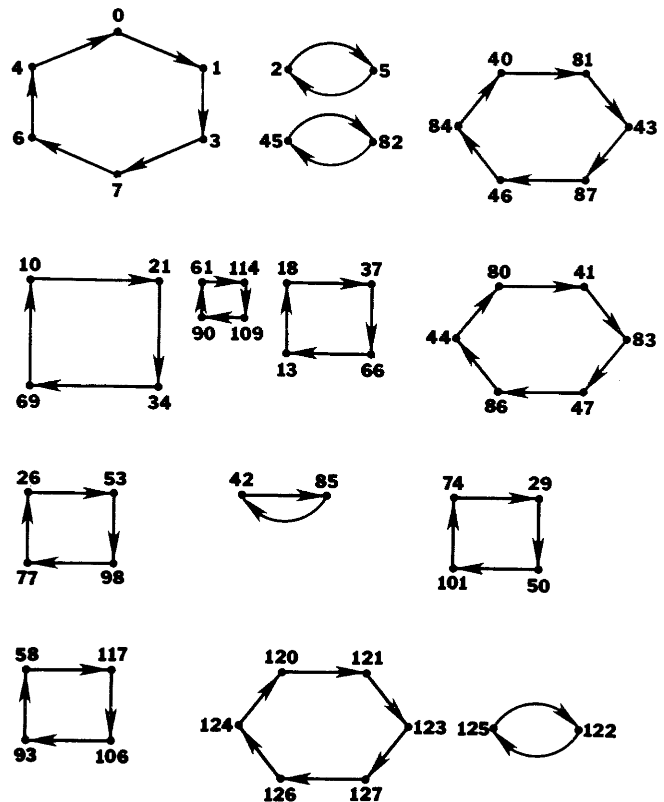


Figure 15—Composite ring structure described with $AF_1 = [4^1, 3^1]$

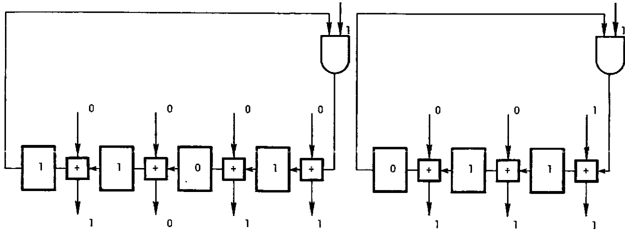


Figure 16—Composite SRVB described with $AF_1 = [4^1, 3^1]$

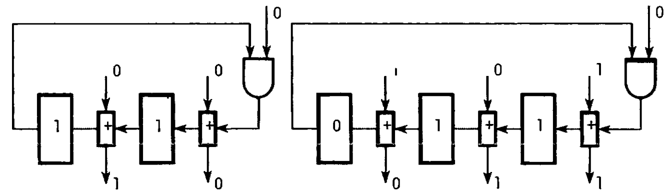


Figure 18—Composite SRVB described with $AF_0 = [3^1, 2^1]$

$AF_{10} = [2^1]_1 \times [2^2]_0$. As seen, to generate MSS, it is required that each SRVB be described with mixed format AF_{10} and a noncircular portion of A_{10} be composite.

CONCLUSIONS

This paper has presented recent research results on reconfiguration of multicomputer networks dedicated to very fast real-time applications. It is shown that there exists a close match between a typical structure of a real-time algorithm and the structures that are assumed by the networks. These structures are rings and single-rooted and multirooted trees and stars. Thus, the capability of the networks to perform very fast reconfigurations from one structure to another is indispensable property for solving fast application algorithms.

This property is accomplished through special reconfiguration algorithms presented in this paper. Two major features of the newly introduced techniques of reconfiguration are

1. They are described by a one-step algorithm performed concurrently by all network nodes requested for reconfiguration.

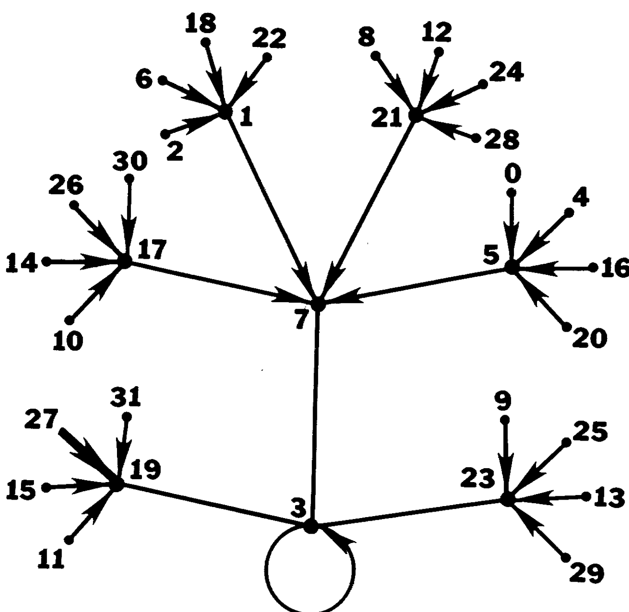


Figure 17—Composite star structure described with $AF_0 = [3^1, 2^1]$

2. The time of this step is equivalent to that of two logical operations performed sequentially: one-bit shift (one-gate delay) and mod-2 addition (2-gate delay).

Therefore, the total time of reconfiguration into a new network structure equals that of 3 gate delays. This means that a new network structure can be established during the time of one clock period and that the reconfiguration introduced gives practically no time overhead. Thus, it can be performed quickly and efficiently to the benefits of real-time computations, each time creating an ideal match between the application algorithm and the network structure that is assumed.

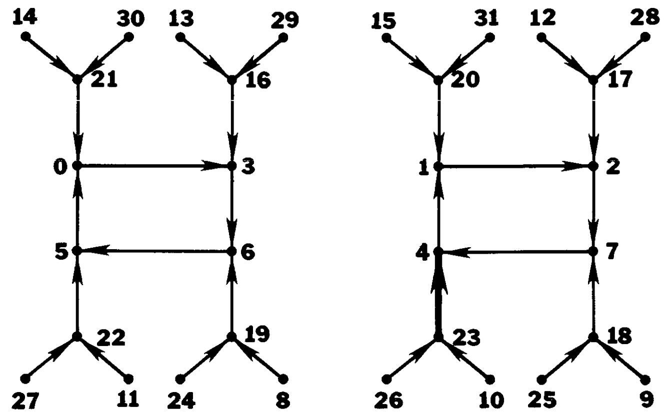


Figure 19—Four-rooted multiple tree structure specified with $AF_{10} = AF_1 \times AF_0 = [2^1, 1^1]_1 \times [2^1]_0$

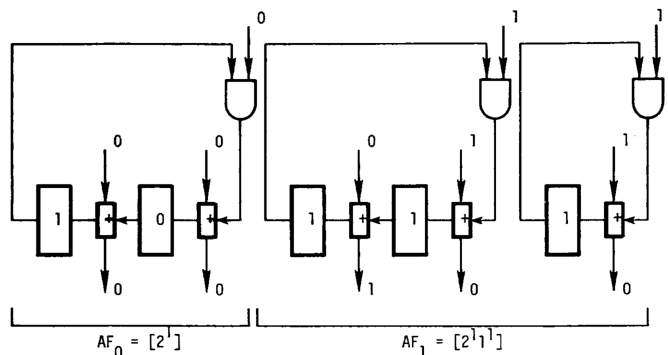


Figure 20—Composite SRVB specified with $AF_{10} = [2^1, 1^1]_1 \times [2^1]_0$

REFERENCES

1. Davis, Carl G., and Robert L. Couch. "Ballistic Missile Defense: A Super-Computer Challenge." *Computer,* 13 (1980), 11, pp. 37–48.
2. Davis, W. A. Jr. "Ballistic Missile Defense Into the Eighties." *National Defense,* September 1979, pp. 55–63.
3. Davis, C. G., and C. R. Vick. "The Software Development System." *IEEE Transactions on Software Engineering* (1977), pp.
4. Vick, C. R. "A Next Generation of Supercomputer From Mainframes to Micros." *Euromicro 80,* London, England, September, 1980.
5. Arnold, R. G., R. O. Berg, and J. W. Thomas. "A Modular Approach to Real-Time Supersystems." Accepted for publication in *IEEE Transition on Supersystems,* May 1982.
6. Vick, C. R., S. P. Kartashev, and S. I. Kartashev. "Adaptable Architectures for Supersystems." *Computer,* 13 (1980), 11, pp. 17–37.
7. Kartashev, S. I., and S. P. Kartashev. "Problems of Designing Supersystems With Dynamic Architectures." *IEEE Transactions on Computers,* C-29 (1980), pp. 1114–1132.
8. Kartashev, S. I., and S. P. Kartashev. "Dynamic Architectures: Problems and Solutions." *Computer,* 11 (1978), pp. 26–40.
9. Kartashev, S. I., and S. P. Kartashev. "Multicomputer System with Dynamic Architecture." *IEEE Transactions on Computers,* C-28 (1979), pp. 704–720.
10. Elspes, B. "The Theory of Autonomous Linear Sequential Networks." *IRE Transactions on Circuit Theory,* (1959), pp. 45–60.
11. Zierler, N. "Linear Recurring Sequences." *J. SIAM,* 7 (1959), pp. 31–48.
12. Kautz, W. H. (ed.). *Linear Sequential Switching Circuits.* New York: Holden-Day, 1965.
13. Golomb, S. W. *Shift Register Sequences.* New York: Holden-Day, 1967.
14. Booth, T. L. *Sequential Machines and Automata Theory,* New York: John Wiley and Sons, 1967.
15. Kartashev, S. P., and S. I. Kartashev. "Reconfiguration of Dynamic Architecture into Multicomputer Networks." *Proceedings of the 1981 International Conference on Parallel Processing,* Bellaire, Michigan, August 25–28, 1981. IEEE Computer Society, pp. 133–141.

MPP: a supersystem for satellite image processing

by KENNETH E. BATCHER
Goodyear Aerospace Corporation
Akron, Ohio

ABSTRACT

In 1971 NASA Goddard Space Flight Center initiated a program to develop high-speed image processing systems. These systems use thousands of processing elements (PE's) operating simultaneously to achieve their speed (massive parallelism). A typical satellite image contains millions of picture elements (pixels) that can generally be processed in parallel. In 1979 a contract was awarded to construct a massively parallel processor (MPP) to be delivered in 1982. The processor has 16,896 PE's arranged in a 128-row by 132-column rectangular array. The PE's are in the array unit (Figure 1). Other major blocks in the massively parallel processor are the array control unit, the staging memory, the program and data management unit, and the interface to a host computer.

ARRAY UNIT

Logically, the array unit contains 16,384 PE's arranged in a 128-row by 128-column square array. Physically, the array unit contains an extra 128-row by 4-column rectangle of PE's for redundancy. Each PE communicates with its four nearest neighbors: north, south, east, and west. Each PE is a bit-serial processor. With a ten-megahertz clock rate and 16,384 PE's operating in parallel the system has a very high processing rate. Each PE can read two 16-bit integers, generate their sum, and store the 17-bit sum in 49 clock cycles, so 16,384 additions are performed in less than five microseconds (more than 3000 MOPS). Floating-point operations are performed at a fast rate even though they are not particularly suited for bit-serial processing. Many different floating-point formats are possible. With a 32-bit format (1-bit sign, 7-bit base-16 exponent and 24-bit fraction) floating-point addition is better than 400 MOPS and multiplication is better than 200 MOPS.

Array Topology

The major application of the massively parallel processor is image processing. Since most of the processing is conducted between neighboring pixels it is natural to connect the thousands of PE's together in a square array with each PE communicating with its nearest neighbors. We investigated the use of other interconnection networks like the multistage SIMD interconnection networks,[1] but with over 16,000 PE's they become unwieldy. The layout of a square array is very simple with no long runs to slow down the transfer rate.

Certain image processing operations like the Fast Fourier Transform (FFT) require communication between pixels or points located far apart in the image. If we store one point in each PE, then the routing time would be severe in a square-array topology. But this is not the best way to do FFT's on the MPP. Each PE can store several points in its random access memory, so the number of PE's required to do an FFT can be reduced to a small compact subarray of the array unit. The processing power of the other PE's is not wasted, since when we want to do one FFT we usually want to do many FFT's so we can divide the array unit up into many compact subarrays, each performing one FFT. For example, suppose we want to do many 5120-point FFT's. Ten points can be packed into each PE, so each FFT can be performed in a 16-row by 32-column subarray of the array unit. Thirty-two such subarrays can perform 32 FFT's in parallel. The longest communication path in each FFT is half the width of the subarray (16 columns), so the routing time can be reduced to a fraction of the computation time.

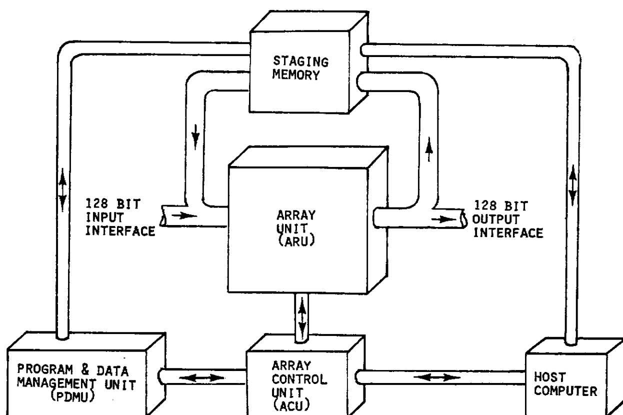


Figure 1—Block diagram of the massively parallel processor

One may ask the question of how the data can be input and output effectively, especially when it has a peculiar layout as in the FFT example. A 5120-point FFT is most easily performed by combining 1024 five-point FFT's with five 1024-point FFT's where the position of any point is a function of its index modulo 5 and 1024. The 5120 points of one FFT have a scrambled layout. The permutations required are akin to the permutations required to change a data array from an item format to a bit-slice format. Some kind of buffer memory will be required in the array unit I/O path to convert data arrays to and from the bit-slice format; if it is properly designed the same buffer memory could perform other permutations as well, such as those required by the 5120-point FFT example.

Thus, in the massively parallel processor we use a simple square array topology in the array unit and insert a buffer memory (the staging memory) in its I/O path to perform the permutations required by particular application programs. The staging memory transforms the bit-serial format of the array unit to the item format of the outside world.

Given a square array with 128 rows and 128 columns what do we do around the edges? Some application programs would like to see a planar topology. Other programs would like to see a cylindrical topology where the PE's on the north edge connect to PE's on the south edge. Also, some programs would rather have the 16,384 PE's connected in one long linear string rather than in a 128 by 128 plane. Thus, the edge connections should be a programmable function.

A topology register is included in the array control unit to allow programming of the edge connections. Between the north and south edges of the array unit one can either stitch them together to make the array look like a cylinder or separate them to make the array look like a plane. Similarly, the east and west edges can independently be stitched together or separated (if both pairs of edges are stitched together the

array looks like a torus). When the east and west edges are stitched together, one can either stitch corresponding rows together or slide the stitching by one row so the west PE of row i communicates with the east PE of row $i + 1$. If one slides the stitching, the rows are connected together in spiral fashion so the array of PE's looks like a long linear string.

Redundancy

One advantage of the rectangular connection network is the easy way it allows faulty PE's to be bypassed. When a faulty PE is discovered, one bypasses all the PE's in its column (or row) so the topology is not disturbed. To add redundancy to the array unit we implement more than 128 columns and insert bypass gates in the east-west routing paths. The array is reduced to 128 columns logically by bypassing some columns. If a faulty PE is discovered we bypass its column and use one of the spare columns instead. Logically, the array will still have 128 columns. Of course, the physical position of many data items will be shifted when the bypassed columns are shifted; but this presents no problem if we don't try to save the data when a fault is discovered. Since the discovery of a fault usually implies the presence of faulty data in the faulty PE and/or its neighbors we should not try to save the data anyway. After the array unit is reconfigured, recovery is accomplished by restarting the application program from the last checkpoint.

We could just add one redundant column of PE's and bypass the 129 columns individually. Instead we divide the array up into 32 four-column groups and add a redundant four-column group so only 33 sets of bypass gates are required instead of 129. When a faulty PE is discovered we bypass all PE's in its four-column group. All outputs from the group are disabled and the east-west routing paths of its two neighboring groups are stitched together. The redundancy of 3 percent is a small price to pay for the ability to reconfigure around any single faulty PE. The scheme does not handle the case of multiple PE's failing, but the probability of this event within a reasonable service interval is miniscule.

Processing Elements

Each PE is a bit-serial element. Initially the PE's had down-shifting binary counters for arithmetic.[2, 3] The PE design was modified to use a full adder and a shift register for arithmetic. Each PE has six 1-bit registers (A, B, C, G, P, and S), a shift register with programmable length, a random-access memory, a data bus (D), a full adder, and some combinatorial logic (Fig. 2). The nominal clock rate of the PE's is 10 megahertz. In each clock cycle all PE's perform the same operations on their respective data streams (except where masked). The basic PE operations are microsteps of the array instruction set. The control signals come from the PE control unit of the array control unit, which reads the microcode from a writable control store. As long as there are no conflicts many PE operations can be combined into one clock cycle.

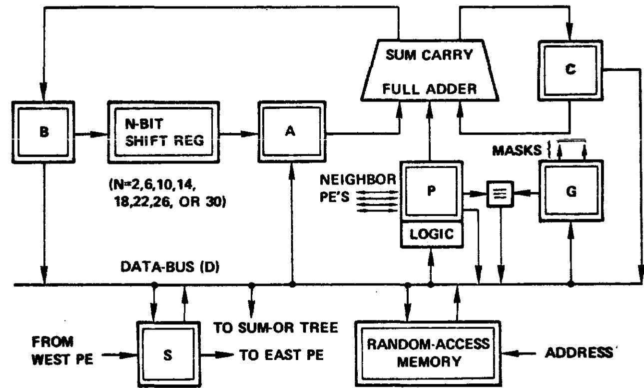


Figure 2—One processing element

Data-bus source selection

The source of the data bus can be the state of the B-, C-, P-, or S-register, the state of a selected bit from the random-access memory, or the output of the equivalence function between P and G ($P \equiv G$ equals 1 if and only if P and G are in the same state). The data-bus state (D) feeds a number of other parts of the PE.

Logic and routing

The P-register is used for logic and routing operations. A logic operation combines the state of the P-register with the state of the data bus (D) to form the new state of the P-register. Any of the 16 Boolean functions of P and D can be selected. A routing operation reads the state of the P-register in a neighboring PE (north, south, east, or west) and stores the state in the P-register. When routing occurs, the 128 by 128 plane of P-registers is shifted synchronously in any of the four cardinal directions.

A logic or routing operation can be unmasked or masked. An unmasked operation is performed in all 16,384 PE's. A masked operation is performed in only those PE's where $G = 1$—the P-register is not disturbed in those PE's where $G = 0$.

Arithmetic

The full adder, the shift register and registers A, B, and C are used for bit-serial arithmetic operations. A full-add operation sums the bits in the A-, P-, and C- registers to form a 2-bit sum that is placed in the C- and B- registers. A half-add operation is similar except that only the bits in registers A and C contribute to the sum.

The shift register has a programmable length. Its length can be set to 2, 6, 10, 14, 18, 22, 26, or 30 bits. A shift operation shifts the register one place to the right, with the state of the B-register entering at the left end of the shift register. If register A is shifted simultaneously, it reads the rightmost bit in the shift register. An operand of length $4n$, where n is an

integer from 1 to 8, can be recirculated around the path formed by register B, the shift register, register A, and the full adder; the shift register length is set to $4n$-2.

Register A has three operations: clear A, load A with the data-bus state D, or load A with the rightmost bit in the shift register (shift A). Register C receives the carry bit in full-add and half-add operations and has two other operations: clear C and set C.

These micro arithmetic operations are combined to perform the array arithmetic instruction set. Addition of two arrays of n-bit integers is performed with each PE treating one pair of integers. Corresponding bits of the integers are fed to the P- and A- registers, respectively, starting with the lowest order bits. They are added with full-add operations with the carry bits recirculating through register C and the sum bits being formed in register B and stored back in the random access memory. It requires $3n + 1$ cycles to read the two n-bit integers from memory and store the $(n + 1)$-bit sum back into memory. Subtraction is performed similarly except that the 1's complement of the subtrahend is loaded into the P-register in place of its true value. Two's-complement subtraction is done by initializing the C-register to 1 instead of 0.

The result of an arithmetic operation can be sent to the shift register instead of storing it to memory. Multiplication is performed by recirculating the partial product through the shift register and adding the multiplicand to it with appropriate shifts. A multiplier bit in the G-register controls the loading of the P-register. Multiplication of an array of n-bit integers by corresponding elements of an array of m-bit integers to produce an array of $(m + n)$-bit integers requires $(m - 1)p + 2(m + n)$ cycles where p is a multiple of 4 equal to n, $n + 1$, $n + 2$, or $n + 3$.

Division uses a nonrestoring algorithm where the partial dividend is recirculated through the shift register and the divisor or its complement is added for each quotient bit.

Floating-point multiplication is an addition of the exponents and a rounded multiplication of the fractions. Floating-point addition is a comparison of the values followed by an alignment of the fractions, addition of the fractions and then a normalization of the result.

Other PE operations

Other PE operations include loading the G-register from the data bus, writing the data bus to a selected bit of the random access memory, loading the S-register from the data bus, feeding the sum-or tree from the data bus, and clearing the memory parity error indicator.

The sum-or tree is a tree of inclusive-or gates with inputs from all 16,384 PE's. The output is fed to the array control unit, which can test and store the result. The sum-or tree is used in maximum value and minimum value searches and in other operations where it is necessary to get a global result from the set of PE's.

Input-output

The S-register in all PE's is used for input and output of array data. Columns of input data are shifted into the S-registers at the west edge of the array unit and shifted across the array until all 16,384 S-registers are loaded. Then the PE processing is interrupted for one machine cycle while the S-register plane is transferred to a selected plane of the random access memories. S-register shifting can run at a 10-megahertz rate, so data can be input at a rate of 160 megabytes per second (128 bits every 100 nanoseconds). Note that PE processing is only interrupted once every 128 columns, or less than 1 percent of the time.

Data output is similar. The PE processing is interrupted for one cycle and a plane of data is transferred from the random access memories to the S-registers. Processing resumes while the output plane is shifted across the array to the east edge, where it is output column by column. Each column is 128 bits long and can be shifted out at a 10-megahertz rate, so the output rate is also 160 megabytes per second. Note that while an output plane is being shifted out an input plane can be shifted into the array unit; so input and output can proceed simultaneously.

At first glance an I/O rate of 320-megabytes per second (160 in and 160 out) would seem to be more than adequate. But the processing rate is so high that some applications may still become I/O bound. When such an application arises (and when fast enough peripherals are available), the array unit I/O scheme can be modified to input and output data at several places in the array instead of just at the east and west edges.

Random access memories

Each PE has a random access memory storing 1024 bits. The address lines of all PE's are tied together so the memories are accessed by planes with one bit of a plane accessed by each PE. Four PE's share one 1K-by-4 RAM chip with an access time of about 50 nanoseconds. The address bus can be expanded up to 16 address lines, so PE memory can be expanded to 65,536 bits per PE or 128 megabytes total. The array unit clock system has enough flexibility to accommodate a wide variety of memory speeds.

Packaging

The PE random access memories use standard RAM integrated circuits. All other components of eight PE's are packaged on a custom VLSI chip. The chip holds a 2-row by 4-column subarray of PE's and 2,112 such chips are used in the array unit. The chip pinout is 52 pins and the complexity is about 8000 transistors.

A 16-row by 12-column subarray of 192 PE's is packaged on one 22-cm by 36-cm printed circuit board. The board contains 24 VLSI chips, 54 memory elements (48 for data plus 6 for parity) and some support circuitry. Eleven boards make up an array slice of 16 rows by 132 columns. Eight array slices (88 boards) make up the array unit and eight other boards hold the topology switches, the control signal fan-out and other support circuitry. The 96 boards are packaged in one cabinet and cooled by forced air.

ARRAY CONTROL UNIT

The array control unit has three subunits: the PE control unit to control processing in the array unit PE's, the I/O control unit to manage the flow of input/output data through the array unit, and the main control unit, which runs the application program, performs any necessary scalar processing, and controls the other two subunits (Fig. 3). This division of responsibility allows array processing, scalar processing and I/O to proceed simultaneously. A queue between the main control unit and the PE control unit can hold up to 16 calls to array processing routines.

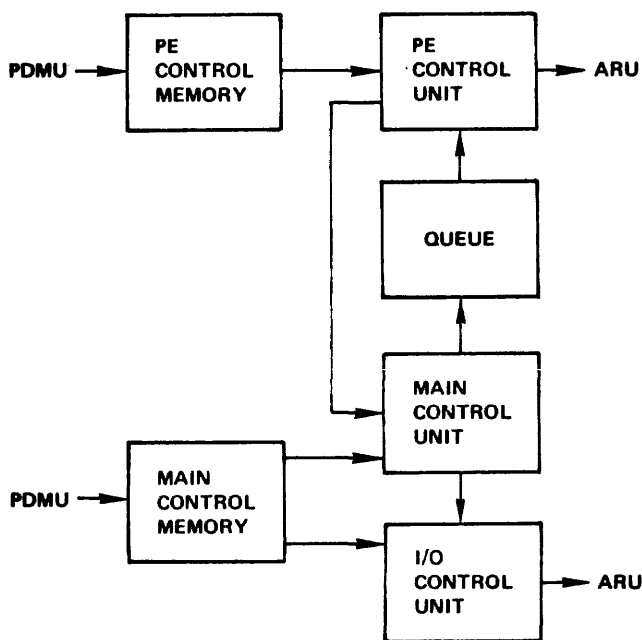


Figure 3—Block diagram of array control unit

PE Control Unit

The PE control unit generates all PE control signals except those associated with I/O and the *S*-registers. The control unit reads 64-bit-wide microinstructions from the PE control memory. The PE control memory holds the standard library of array processing routines plus any routines generated by users, so it is like the writable control store in other computers. When the PE control unit receives a call from the queue, it reads the calling parameters and jumps to the entry point of the called array processing routine. After executing the routine, the PE control unit then processes the next call from the queue.

The PE control unit contains a 64-bit-wide common register to hold the scalar values required by routines that combine scalars with arrays, that search arrays for values, or that generate a scalar from an array.

There are eight 16-bit index registers in the PE control unit. One index register holds the index of a selected bit in the common register. Since array processing is bit-serial, the common register scalar is also usually treated bit by bit. The selected common-register bit *(W)* can be tested by branch instructions, used to select a *P*-register logic function in all PE's, and loaded by the sum-or tree output. Note that using the common-register bit *(W)* to select a *P*-register logic function allows one to select any of the 256 logic functions of three variables—in every PE the selected function between register *P*, the data-bus state *(D)*, and the common-register bit *(W)* is stored in register *P*. This is the mechanism used to broadcast common-register values to all PE's.

The other seven index registers can hold the addresses of array bit-planes in the PE random access memories. Any of the eight index registers can be used to hold the length of an array. Many of the array processing routines are variable length, so they use an index register to hold a loop count and decrement it once for each bit-plane treated.

Other registers in PE control include the topology register to select the array unit topology, a program counter holding the location of the current microinstruction in the PE control memory and a subroutine return stack to facilitate using some array processing routines as subroutines to other routines.

The instruction register is 64 bits wide. Most instructions are executed at a nominal 10-megahertz rate. Several operations can be merged into one instruction; for example, several PE operations, modification of several index registers, and conditional branching. Merging allows most of the control unit overhead to be absorbed so that the PE's are doing useful work on every machine cycle.

I/O Control Unit

The I/O control unit shifts the PE *S*-registers, manages the flow of data in and out of the array unit, interrupts PE control to transfer data between the *S*-registers and the PE memory elements, and can also control the staging memory. Once initiated by main control or the program and data management unit, the I/O control unit chains through a sequence of I/O commands in main control memory.

Main Control Unit

Main control reads and executes the application program from the main control memory. It performs all scalar processing itself and sends all array processing calls to the queue for processing by the PE control unit. Input and output operations for the I/O control unit are either sent directly to the I/O control unit or sent to the program and data management unit for coordination with its peripheral transfers.

Main control has 16 general-purpose registers, some registers to enter calling parameters into the PE control unit queue, and other registers to receive scalars generated by certain array processing routines.

STAGING MEMORY

The staging memory is in the I/O path of the array unit. Besides acting as a buffer between the array unit and the

outside world, the staging memory reformats data so both the array unit and the outside world can transfer data in the optimum format. The array unit sees data in a bit-plane format (one bit from 16,384 different items), while the outside world sees data in an item format (all bits of one item). The staging memory can also rearrange data to match the scrambled layouts of some application programs. The 5120-point FFT example is one such program.

The staging memory comprises a main stager memory, an input sub-stager, and an output sub-stager (Fig. 4). The main stager memory can have 4, 8, 16, or 32 banks of storage with 16K, 64K, or 256K words per bank. Each word holds 64 data bits plus 8 error-correction bits. A fully implemented main stager would hold 67 megabytes of data. Each bank contains 72 dynamic MOS RAM elements. Initially, 16K bit elements are used. When repopulated with 64K bit elements, the storage in each bank is quadrupled. When repopulation with 256K bit elements is feasible the storage per bank can be quadrupled again. Each bank can accept a 64-bit word and present a 64-bit word every 1.6-microsecond cycle time (the cycle time also includes any memory refresh required), so each bank has a ten-megabyte-per-second I/O rate (5-megabyte-per-second input and 5-megabyte-per-second output). A 32-bank main stager can accept and present data at the 160-megabyte-per-second rate of the array unit I/O ports.

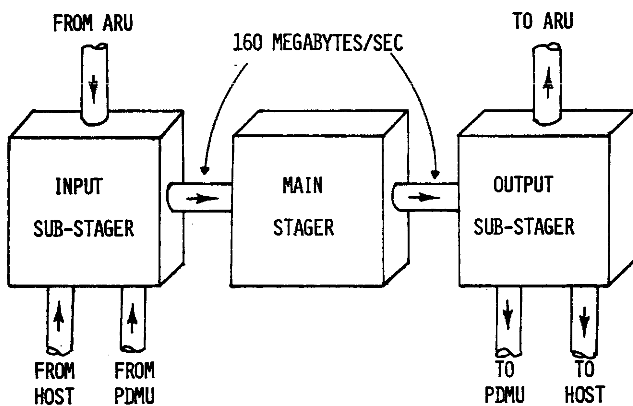


Figure 4—Staging memory

The sub-stagers are fast 128-bit by 1024-bit ECL multidimensional access memories.[4] The input sub-stager accepts data in the format of the source (array unit, program and data management unit, or the host) and rearranges the data to agree with the main stager format. The output sub-stager performs the complementary function of rearranging data from the main stager format to the format of the destination.

Many different main stager formats are possible—a main stager word may contain one bit of 64 different elements, two bits from 32 different elements, and so on. The main stager format is selected according to the data formats in the source and the destination. A software module in the program and data management unit can be used to select the main stager format and program the internal transfers of the staging memory.

PROGRAM AND DATA MANAGEMENT UNIT

The program and data management unit can control the overall flow of programs and data in and out of the massively parallel processor. It acts as a small-scale host when the normal host is not available. The program and data management unit is a DEC PDP-11 minicomputer with a number of terminals, a line printer, disk storage, and a tape unit operating under DEC's RSX-11M real-time multiprogramming system. Custom interfaces provide communication with the array control unit and the staging memory.

The program development software package for the massively parallel processor executes in the program and data management unit. The package includes an assembler for the PE control unit to facilitate developing array processing routines; an assembler for the main control unit to develop application programs; a linker to form load modules for the array control unit; and a control and debug module to load, execute, and debug programs. Much of the software development package is written in FORTRAN to ease the transfer of the package to the host computer.

HOST INTERFACE

The massively parallel processor to be delivered to NASA will connect to a DEC VAX-11/780. The staging memory is connected to a DEC DR-780 high-speed interface of the VAX that can transfer data at a rate of 6 megabytes per second. The staging memory interface is designed to accommodate other devices such as high-speed disks. To allow control of the massively parallel processor by the host, the array control interface can be switched from the program and data management unit to the host computer. Transfer of the software is simplified by writing much of it in FORTRAN.

CONCLUSIONS

The massively parallel processor is designed for high-speed processing of satellite imagery. The typical operations may include radiometric and geometric corrections and multispectral classification. Preliminary application studies indicate that the processor may also be useful for other image processing tasks, weather simulation, aerodynamic studies, radar processing, reactor diffusion analysis, and computer image generation.

The modular nature of the processor allows the number of processing elements and the capacities of its memories to be scaled up or down to match the requirements of the application.

REFERENCES

1. Siegel, H. J., and S. D. Smith, "Study of Multistage SIMD Interconnection Networks," *Proceedings of the 5th Annual Symposium on Computer Architecture,* Palo Alto, Calif., April 1978. IEEE Computer Society, Long Beach, Calif., pp. 223–229.
2. Fung, L. W., "A Massively Parallel Processing Computer," in *High-Speed Computer and Algorithm Organization,* D. J. Kuck et al. eds., New York: Academic Press, 1977, pp. 203–204.
3. ______ "MPPC: A Massively Parallel Processing Computer," GSFC Image Systems Section Report, Sept. 1976, Goddard Space Flight Center, Greenbelt, MD.
4. Batcher, K. E. "The Multidimensional Access Memory in STARAN." *IEEE Transactions on Computers,* C-26 (1977), pp. 174–177.

Optimal design of a distributed supersystem

by DAVID F. PALMER
General Research Corporation
Santa Barbara, California
and
JAMES P. IGNIZIO *and* CATHERINE M. MURPHY
Pennsylvania State University
University Park, Pennsylvania

ABSTRACT

The design of distributed systems, in particular supersystems, requires an extraordinary number of decisions. Designers must define numbers and types of computing components, interconnection networks, software modules, and the allocation of software to hardware.

This paper describes a formal approach to determining complex design decisions. The approach differs from most formal decision techniques as follows: (1) it generates multiple optimal design solutions; (2) it deals with multidimensional criteria; and (3) it has been implemented by an efficient (polynomial growth) algorithm. The approach uses fuzzy clustering, generalized goal programming, and a refined vectormax technique.

INTRODUCTION

Hardware cost, performance, and reliability trends point to a preference for the use of distributed computing systems. This is especially true for the new wave of supersystems. At the same time distributed system design is generally much more complex because of the large number of design alternatives. Designers must define numbers and types of computing components, interconnection networks, software modules, and the allocation of software to hardware. The complexity must be managed to enable achievement of design goals and keep personnel costs from offsetting other advantages. A systematic design approach and automated decision aids are needed to reduce system complexity.

This paper describes a formal approach to determining complex design decisions. The approach differs from most formal decision techniques as follows:

1. It generates multiple optimal design solutions so that the designer can clearly view tradeoffs, e.g., between cost and reliability.
2. It deals with the multidimensional aspect of distributed system design decisions without forcing all objectives to be translated into common terms. (For example, reliability does not have to be assigned a dollar value.)
3. It has been implemented by an efficient algorithm, which enables the total range of decisions to be assessed. In fact, the algorithm can be used for dynamic decisions during system operation.

The approach complements and extends modern design methodologies.

PROBLEM DEFINITION

We assume that design is initiated by defining functional and data elements and their relationships to satisfy requirements of a given application. The elements and relationships are defined without considering specific physical computing resources. A structured design approach, such as described in Yourdon and Constantine,[1] may be assumed.

The results of the initial design phase are best represented by a data flow or data access graph. Here we have typically many nodes representing functional and data elements. The connecting links represent data flow or accessing. Control representation is avoided because it tends to bias selection of physical resources.

Figure 1 illustrates the resource requirements definition problem. Given the data access graph containing functional and data elements, we must define the number and types of physical resources to support the indicated processing. Notice that the functional and data elements must be mapped onto the physical resources at the same time. The mapping is necessary for determining resource requirements.

One of the major difficulties of resource definition and functional mapping is the large number of alternatives. In general, if we have N elements, we need to examine use of $n = 1, 2, \ldots, N$ resources. If all resources are identical, the number of alternatives is

$$\text{Number of alternatives} = \sum_{m=1}^{N} S_N^{(m)}$$

where $S_N^{(m)}$ is the Stirling number of the second kind,

$$S_N^{(m)} = \frac{1}{m!} \sum_{k=0}^{m} (-1)^{m-k} \binom{m}{k} k^N$$

Permutations of solutions must be considered for nonidentical resources. The results are plotted in Figure 2. We see that brute-force techniques can very quickly become computationally impractical because of computing time, storage requirements, and possibly numerical instability.

Our design problem belongs to the class of NP-complete problems, as was recently proven by Malmquist.[2] There is no point in attempting to apply conventional, exact solution algorithms[3-5]—such as branch and bound, implicit enumeration, or dynamic programming—to NP-complete or combinatorially explosive problems, because any exact algorithm will exhibit exponential growth in terms of solution time. For example, a branch and bound algorithm may require about 2^n branches, where n is the number of decision variables. For a problem with just 50 variables, a high-speed computer that

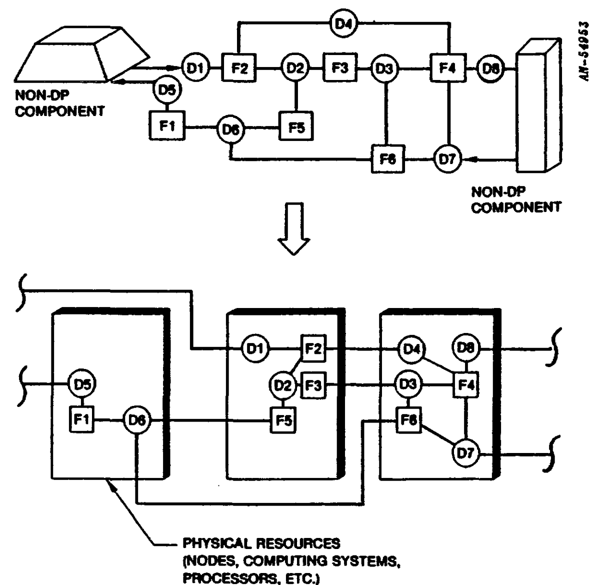


Figure 1—Resource definition problem

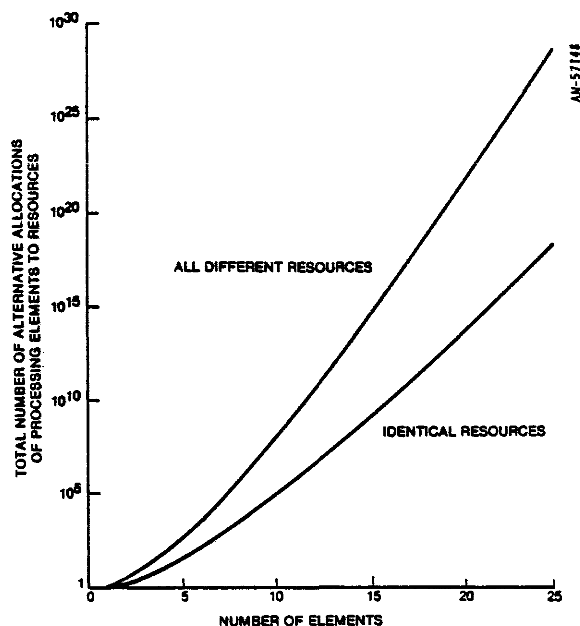


Figure 2—Numbers of alternative allocations

could explore 1,000 decision branches per second would require over *30,000 years* to solve the problem. Consequently, we need an algorithm with polynomially related, say n^2, computational requirements. An n^2 algorithm would need to search only 50^2 branches, requiring only *2.5 seconds.*

The other major difficulty of complex design is the mathematical evaluation of alternatives. We need models relating achievement of objectives (e.g., performance, reliability, cost) to parameters of alternatives (e.g., types of resources, functional assignments to resources). In addition, we need a decision technique that handles multiple, noncommensurable objectives without forcing definition of relative values. For example, is a 0.95-reliable five-million-dollar system better than a 0.93-reliable one-million-dollar system?

SOLUTION APPROACH

Our approach has three phases: (1) fuzzy clustering,[6-9] (2) generalized goal programming,[10-14] and a refined vectormax approach. We outline the three phases below.

Fuzzy clustering

The first step in most (practical) nonlinear optimization procedures is determination of an initial, starting solution about which a numerical search may be conducted. The fuzzy clustering procedure performs this step for distributed computing design very simply and systematically while also greatly reducing the complexity of the overall problem. For example, in a 70-variable problem, we may be able to reduce the search region from 2^{70} possible permutations to, say 2^{20}.

Our fuzzy clustering approach, which is due to Buckles and Hardin,[7] groups functional elements on the basis of tabulated relationships. To illustrate the approach, we use data accessing relationships. Other types of relationships can be handled similarly.

A *data access matrix* is used to define data accessing relationships. A data access matrix contains the total numbers of data access by each function, compiled over an appropriate interval of time. For high-level design the matrix would usually be determined by simulation and the partitioning completed as described here. The approach is simple enough, however, that the procedure might be performed at run time to provide a dynamic allocation.

The procedure is as follows: (1) compute the pairwise proximity of functional entities with respect to commonality of data accessing; (2) produce a transitive relation among functional entities by computing their similarity; (3) apply thresholds of similarity to generate a family of candidate partitions; and (4) select the partition having the best potential implementation.

The proximity of function F_i and F_j is defined to be

$$P_{ij} = P_{ji} = \frac{\sum_k [\mathrm{MIN}\,(A_{ik}, A_{jk})]}{\sum_k [A_{ik}+A_{jk}]/2} \tag{1}$$

where A_{ik}=number of accesses of data D_k by function F_i.

The numerator of Equation 1 is seen to be the total mutual accessing of data by F_i and F_j. The denominator is the total average accessing of data by functions F_i and F_j, i.e., the sum of all data accesses by F_i and all data accesses by F_j divided by 2.

The proximity matrix is not entirely useful for partitioning because it does not provide transitivity of relationships. To gain a transitive relationship we compute the similarity of functions F_i and F_j by the equation

$$S_{ij} = S_{ji} = [\mathrm{MIN}\,(P_{ik}, P_{kj})] \tag{2}$$

where k ranges over all functions. (A useful alternate computation using matrix operations is

$$S = P \vee P^2 \vee P^3 \vee \ldots \vee P^n \tag{3}$$

where V is the maximum operation and products are actually minimum operations.[9]) In computing similarities we find the strongest mutual "friend" for each pair of functions. Then we compute further-removed relationships until complete transitivity is reached.

The next step is to form functional groups by applying thresholds of similarity. At any similarity threshold we group together only functions that are more similar than the threshold. Thus at a similarity threshold of 0 all (n) functions would form a single group, no matter how little similarity they might have. As we raise the threshold, we form more and smaller groups containing functions with higher similarity. Finally, when the threshold is set at unity, all functions become separated; we obtain (n) groups containing 1 function each.

The similarity threshold may be preselected or used as a problem variable to be maximized. (Generally, high values of similarity result in low communication. High similarity also tends to produce functionally cohesive modules, which eases maintenance.)

Generalized goal programming

Goal programming (GP) may be considered just one type of mathematical programming approach. After the development of linear programming, which encompasses but a single objective, the vast majority of the research in mathematical programming dealt with only the single objective model. However, a small number of investigators, primarily Charnes and Cooper,[10] Ignizio, Kornbluth,[17] Lee,[14] and Steuer[18] directed their attention to practical approaches to the modeling and solution of multiobjective problems. In the early 1950s, Charnes and Cooper developed the method of "constrained regressions," later dubbing it as "goal programming" in their 1961 text on mathematical programming.[10]

Goal programming was but one of several approaches to multiobjective mathematical programming that have appeared primarily from the mid-1960s to the present. The vectormax technique, actually the method of nondominated solutions (defined later), has also received attention. One of the drawbacks of this approach is that it is currently not practical for realistic, large-scale problems. More recently, Zimmerman[8] and others have proposed a combination of fuzzy sets and mathematical programming that leads to a methodology known as fuzzy programming.

Generalized goal programming has now evolved into a highly flexible methodology, encompassing several of the attributes of both the nondominated solution method and fuzzy programming, (1) which is applicable to any problem involving either linear or nonlinear functions and continuous or discrete variables; and (2) wherein the objectives and goals within the problem may be weighted, or simply ranked, or worked with by means of some combination of weighting and ranking.

The baseline model is the first step in the construction of the final mathematical model. The general form of the baseline model is as follows:

Find $\bar{x} = (x_1, x_2, \ldots, x_n)$ (the control variable values) so as to

$$\text{maximize: } f_r(\bar{x}) \text{ for all } r \quad (1)$$

$$\text{minimize: } f_s(\bar{x}) \text{ for all } s \quad (2)$$

$$\text{satisfy equalities/inequalities:} \quad (3)$$

$$f_t(\bar{x}) \begin{Bmatrix} \leq b_t \text{ or} \\ = b_t \text{ or} \\ \geq b_t \end{Bmatrix} \quad \text{for all } t$$

where b_t is a constant

where $f_r(\bar{x})$ and $f_s(\bar{x})$ are sets of objectives to be maximized and minimized and $f_t(\bar{x})$ is the set of goals and rigid constraints that are to be satisfied if at all possible. (Note that only *one* of the signs $\geq$, $=$, or $\leq$ will hold for each goal or rigid constraint.) *Any* problem that may be quantified may be placed into a baseline model; but, in general, there are no practical methods for solving a problem in this format. The steps for conversion of the baseline model into a generalized goal programming model, together with methods for solving GP problems and interpreting the output, are described by Ignizio.[11]

Basically, the Phase 2 algorithm, which we call the nonlinear discrete goal programming (NLDGP) algorithm, conducts a search on the unit hypercube of all feasible solutions. The search method is a highly modified version of the Hooke-Jeeves pattern search but differs from the conventional approach in that, as the search proceeds in continuous space, a parallel search is being conducted in discrete (i.e., zero-one) space.

Phase 2 yields a solution to the distributed computing system design problem that is an attempt to satisfy all rigid constraints as well as to come as close as possible to an optimal compromise between conflicting goals such as cost and reliability.

Refined vectormax approach

Phase 3 of the overall solution methodology uses the output of Phase 2 as its input (in conjunction with some additional information). (We also can use the Phase 3 algorithm on any arbitrary guess at the optimal solution.) This phase produces a so-called nondominated solution set. A nondominated solution is a solution to a multiobjective problem that is not dominated by any other solution: Design A dominates Design B if it is equal to or better than Design B in every way —i.e., according to *all* the performance measures under consideration).

Phase 3 thus uses the *single* input design to develop a relatively small collection of nondominated solutions from the many thousands possible. This solution subset, rather than a single solution, is then presented to the designer for consideration. The steps used to accomplish this phase of the overall process are listed below:

Step 1. List the results of Phase 2 for each design objective (i.e., the resultant cost, reliability, similarity), or, alternately, list *any arbitrary* solution. Call this solution the initial solution.

Step 2. Determine the amount of degradation in performance, for each design goal, that you would be willing to give up to significantly improve some other design goal. For example, how much additional cost might you accept so as to significantly improve reliability?

Step 3. Enter the initial solution from Step 1 and the degradation allowances from Step 2 into the exchange search algorithm of Phase 3.

Step 4. Exercise the algorithm. The output is the nondominated designs *within the degradation levels* specified in Step 2.

CONCLUSION

This work has resulted in a practical and viable approach for the modeling, analysis, and solution to the distributed computing design problem. The concept has been proved by experiments with our prototype code. We are now refining the code, mainly by improving user interfaces.

REFERENCES

1. Yourdon, E., and L. L. Constantine. *Structured Design.* Englewood Cliffs, New Jersey: Prentice-Hall, 1979.
2. Malmquist, J. P. "Storage Allocation for Access Path Minimization-Structured Data Bases." Ph.D. Dissertation, Pennsylvania State University, 1979.
3. Ignizio, J. P. "Solving Large-Scale Problems: A Venture into a New Dimension." *Journal of Operational Research Society,* 31 (1980), pp. 217–228.
4. Lewis, H. R., and C. H. Papadimitrion. "The Efficiency of Algorithms." *Scientific American,* January 1978, pp. 96–109.
5. Steen, L. A. "Computational Unsolvability." *Science News,* 109 (1976), pp. 298–301.
6. Anderberg, M. R. *Cluster Analysis for Applications.* New York: Academic Press, 1973.
7. Buckles, B. P., and D. M. Hardin. "Partitioning and Allocation of Logical Resources in a Distributed Computing Environment." In M. P. Mariani and D. F. Palmer (eds.), *Tutorial: Distributed System Design.* Piscataway, New Jersey: IEEE Computer Society, 1979, pp. 247–276.
8. Zimmermann, H. J. "Fuzzy Programming and Linear Programming with Several Objectives." *Fuzzy Sets and Systems,* 1 (1978), pp. 45–55.
9. Negoita, C. V., and D. A. Ralescu. *Applications of Fuzzy Sets to Systems Analysis.* New York: John Wiley and Sons, 1975.
10. Charnes, A., and W. W. Cooper. *Management Models and Industrial Applications of Linear Programming.* New York: John Wiley and Sons, 1961.
11. Ignizio, J. P. *Goal Programming and Extensions.* Lexington, Massachusetts: D. C. Heath and Co. (Lexington Books), 1976.
12. Ignizio, J. P. *Linear Programming in Single and Multiobjective Systems.* Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
13. Ignizio, J. P. "Goal Programming: A Tool for Multiobjective Analysis." *Journal of Operational Research Society,* 29 (1978), pp. 1109–1119.
14. Lee, S. M. *Goal Programming for Decision Analysis.* Philadelphia: Auerbach, 1972.
15. Ignizio, J. P. "The Determination of a Subset of Efficient Solutions Via Goal Programming." Forthcoming in the *International Journal of Computers and Operations Research.*
16. Ignizio, J. P. *Si-II Trajectory Study and Optimum Antenna Placement.* SID-63 Technical Report, North American Aviation, 1963.
17. Kornbluth, J. S. H. "A Survey of Goal Programming." *OMEGA,* (1973), pp. 193–205.
18. Steuer, R. E. "Multiple Objective Linear Programming with Interval Criterion Weights." *Management Science,* 23 (1976), pp. 305–316.

Distributed processing with the NS16000 family

by LESLIE KOHN
Intel Corporation
Santa Clara, California

ABSTRACT

The paper discusses the benefits of microprocessor-based distributed processing systems, which are greater than those of conventional timeshared mini or mainframe systems. It highlights the advantages of the NS16000 microprocessor family for this application, and it explains how the NS16000 operating system supports distributed processing.

INTRODUCTION

Microprocessor-based distributed processing systems are rapidly gaining popularity as a simple and cost-effective way of providing high-performance computer systems. A microprocessor-based distributed system can give better performance at lower cost than a timeshared mini or mainframe system. The NS16000 family is well suited to such distributed systems because of its true 32-bit architecture and demand-paged virtual memory support. This makes it easy to run software that formerly would only fit on a large mini or mainframe machine.

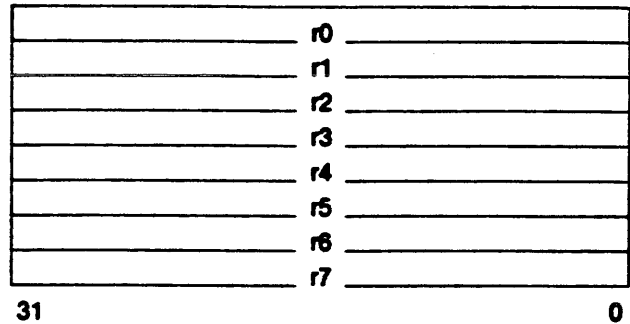
The importance of distributed processing was recognized from the inception of software development for the NS16000 family. The NS16000 operating system supports transparent distributed processing, which allows the tasks of a software system to be placed on different nodes without any change in the tasks. The operating system itself has been modularized into a collection of software tasks. This allows an operating system component such as the file manager to be located far from another operating system component, such as the memory manager. This mechanism is used to implement transparent access to nonlocal network resources such as disks. In this manner expensive hardware resources, such as disk or high-speed printers, may be shared by several NS16000 processors, increasing the cost effectiveness of the system.

NS16000 FAMILY ARCHITECTURE OVERVIEW

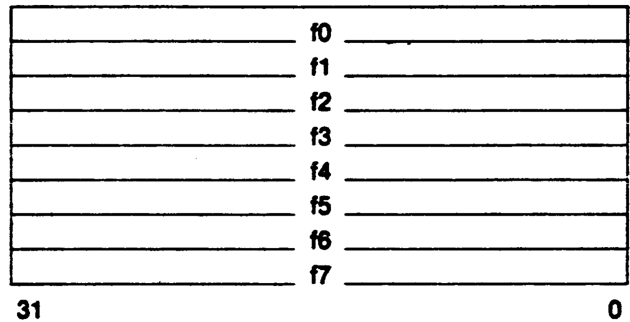
The NS16000 family has been designed to support large software systems. The NS16032 CPU provides a 16-Mbyte uniform address space for efficient access to large programs and/or data structures. The CPU contains eight 32-bit general-purpose registers and seven special purpose registers visible to the programmer (Figure 1). The instruction set provides a symmetric set of operations for 8-, 16-, and 32-bit integers. The instruction set also provides operations for strings, bit and bit field manipulation, packed decimal, and arrays.

The instruction set is designed to be efficient for high-level-language compilation. To eliminate register allocation bottlenecks, the instruction set uses a general two-address format. Any addressing mode may be used for source or destination operands so that memory locations may be used as accumulators or pointers. The nine address modes (Table I) implement the typical variable references of high-level-language programs. The offset constants used by the address mode are frequency-encoded so that small (−64 to 63) values take only 1 byte in the instruction stream, medium values (−8192 to 8191) take 2 bytes, and large values take 4 bytes. By encoding the length in the upper 2 bits of the offset, any offset up to the full address space size may be used without excessively increasing the size of the address mode field. Instructions are

General Registers



Floating Point Registers



Special Purpose Registers

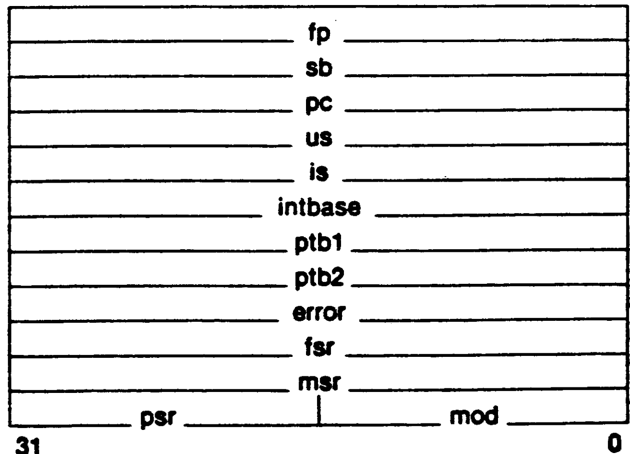


Figure 1—NS16000 Register Set

TABLE I—Address modes

Number	Syntax	Name
0–7	r0 or f0	register
8–15	disp(r0)	register relative
16	disp(disp(fp))	frame memory relative
17	disp(disp(sp))	stack memory relative
18	disp(disp(sb))	static memory relative
19		reserved for future
20	value	immediate
21	disp	absolute
22	ext(disp)	external
23	tos	top of stack
24	disp(fp)	frame memory
25	disp(sp)	stack memory
26	disp(sb)	static memory
27	disp(pc)	program memory
28	mode[r0:b]	index byte
29	mode[r0:w]	index word
30	mode[r0:d]	index double
31	mode[r0:q]	index quad

also byte-aligned and frequency-encoded for compactness.

The instruction set includes operations for implementing the program control constructs found in high-level languages. Procedure entry and exit instructions manage the stack frame and registers for procedure calls. A case instruction implements a multiway PC relative branch. A modular software facility supports structuring of large programs from small manageable modules. Some of the benefits of this system include fully ROMable software and code size reduction, resulting from the smaller addresses needed for referencing within a module. Each module has three components: code, data, and interface. All references to objects in other modules are bound by the interface component. There is no modification of the code component as the module as linked into various programs. The locations of the three components of each module are defined by a module table, which allows a program to be dynamically configured.

The instruction set implemented by the CPU may be extended in a software-transparent fashion through the use of slave processor chips. One slave processor is the NS16081 floating-point unit (FPU). The FPU implements a complete set of operations on 32- and 64-bit floating-point numbers compatible with the proposed IEEE standard. The floating-point unit contains 8 additional 32-bit registers for holding temporary floating-point values. The FPU can perform a 64-×-64-bit floating-point multiply in about 6 microseconds.

Another slave processor is the NS16082 memory management unit (MMU). The MMU implements a demand-paged virtual memory system. Every virtual address emitted by the CPU is translated to a physical address by using the two-level translation algorithm shown in Figure 2. A cache in the MMU holds a copy of the most recently referenced mapping information so that most addresses can be translated without any reference to the page tables contained in memory. If the address is not in the cache, the MMU automatically accesses the page tables and loads the translated address into the cache for future reference. The MMU also provides software debugging aids, including data breakpoints and program flow tracing.

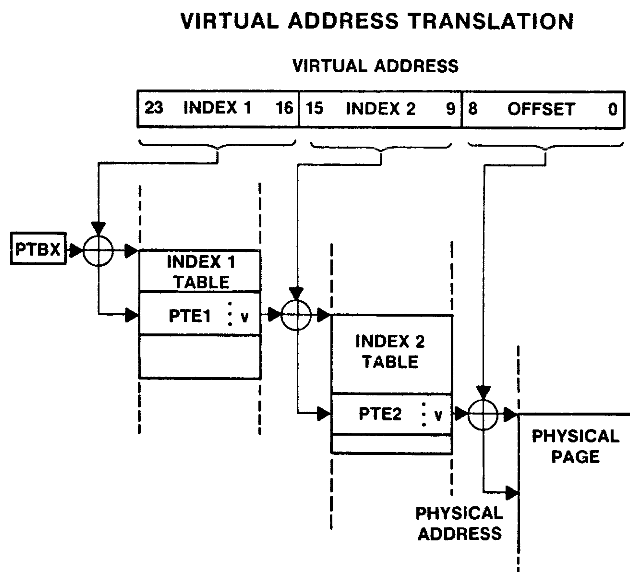


Figure 2—Virtual to physical address translation algorithm

MOTIVATION FOR DISTRIBUTED PROCESSING

In the past it was not possible to run large software applications on low-cost microprocessor-based systems. Microprocessors had limited address spaces. They did not adequately support high-level languages; consequently, it was not possible to compile code that was anywhere close to assembly language efficiency. They did not support fast floating-point operations in hardware; therefore floating-point operations took much longer than they did on a minicomputer or mainframe. They did not support efficient paged virtual memory management. With the NS16000 family, three chips implement a software architecture fully comparable in sophistication to a high-end 32-bit minicomputer. In fact, a complete system built around the NS16000 chip set and selling for $10,000 would have about half the performance of a VAX 11/780, at a cost an order of magnitude lower.

The low cost of a microprocessor means that there is little economic incentive to timeshare a processor between several users. To increase performance and eliminate contention problems, it is desirable for every user to have a dedicated processor in his/her work station. The increased computer power can be readily translated into a more friendly system by supporting better user interfaces, such as graphics and multiple display windows. Connecting these work stations to form

a distributed system can bring two additional significant benefits: access to nonlocal databases and access to high-performance peripherals that are too expensive to justify for a single user. Sharing high-performance peripherals minimizes the need for local peripherals. For instance, a node running a virtual memory operating system need not have a local disk.

OPERATING SYSTEM SUPPORT OF DISTRIBUTED PROCESSING

There are many variations in distributed system configurations. For example, the nodes may be interconnected by an Ethernet or SDLC links. There may be 1 node or 20 nodes in the system. Access to a database may be local or remote. For these reasons it is desirable for the system configuration to be transparent to the application software tasks. The mechanism by which one software task communicates with another task should not depend on whether the tasks are on the same node or how the nodes are connected if they are not on the same node. Furthermore, it is desirable for the operating system itself to be partitioned into multiple tasks so that they may easily be distributed throughout the network by the same transparent communication mechanism. For example, the virtual memory manager task should be independent of the location of the file manager used for swap requests.

The NS16000 operating system supports transparent distributed processing with a modular message-based design. The operating system consists of the kernel and a set of resource manager tasks. The kernel regulates the execution of tasks and low-level access to local hardware resources. Each task generally runs in its own address space so that it cannot interfere other tasks or hardware resources. A resource manager task controls access to a hardware resource for other tasks. A resource manager implements an allocation and protection mechanism that allows multiple tasks to share the resource. It provides a high-level abstraction that is easier to use than the low-level hardware interface. For example, the disk resource is controlled by the file manager, which implements a file abstraction for other tasks to access the disk. The physical memory resource is controlled by the address space manager, which implements a virtual memory abstraction for other tasks.

The kernel implements an interprocess communication mechanism called circuits. A circuit is a path that allows one task to send messages to another task or to the kernel. Each task has a circuit table, managed by the kernel, containing all the currently valid paths by which a task can send and receive messages. The key to supporting transparent distributed processing is that a circuit may be extended transparently across the network. This is done by one of the resource manager tasks called the network manager. The network manager interposes itself between two tasks connected by a circuit when the tasks are on different nodes. When the transmitting task tries to send a message to the receiving task, it is intercepted by the local network manager. The network manager adds the appropriate network protocol and sends the message over the network to the network manager on the node containing the receiving task. The receiving network manager then strips off the protocol and performs a kernel send operation to the receiving task. Thus, all network dependencies are encapsulated in the network manager process.

OBJECT MANAGEMENT IN A DISTRIBUTED SYSTEM

The NS16000 operating system supports a general-purpose object management mechanism. An object is an instance of some user-definable abstraction. The behavior of the object is determined by the set of operations that are implemented by an object manager task for that abstraction. There is no direct access to the internal representation of the object. The object management mechanism provides a uniform way of accessing objects, be they files, directories, devices, or any other objects. It provides the ability to name and locate objects independent of the location of the accessor or of the object itself.

There are two types of object references: short-term and long-term. A short-term reference is lost when the process holding the reference is terminated, and this reference is implemented by a circuit to the object manager. Operations are performed by sending messages over the circuit that are interpreted by the object manager. Since circuits are transparently extended across the network, short-term references are automatically independent of the path from the accessor to the object. A long-term reference to an object exists beyond the life of the task that creates the reference. For example, a file created by a text editor should continue to exist after the edit session is terminated. Since the circuits held by a task are lost after the task is terminated, another way to reference long-term objects is needed. This is accomplished with an object identifier, which consists of two parts: an object serial number and a hint of where the object is located. Each object has a unique serial number, which is used to ensure that no impostor object is located if the referenced object is moved or deleted. The hint is used to guide the system in searching for the object over the network. An object identifier is generated by the system for newly created objects. When a task wants to perform operations on the object, it converts the long-term reference (object identifier) to a short-term reference (circuit) with the system open operation.

The system provides a standard directory manager task that provides mapping between UNIX-style pathnames and object identifiers. The directory system allows arbitrary nonhierarchical directory structures. This is useful in a distributed system, where there may be multiple file servers and no global directory system root. Because objects are referenced independently from their path name, several local naming conventions may be used in the same network.

SUMMARY

An NS16000-based distributed system can be a highly cost-effective alternative to conventional timeshared mini or mainframe systems. The NS16000 architecture effectively supports the large software applications found on large computer systems. The NS16000 operating system has been designed with careful consideration to the issues arising in distributed systems.

ACKNOWLEDGMENTS

The following people made valuable contributions to the NS16000 operating system: Dan O'Dowd, Ross Harvey, Jim Avera, Peter Bishop, Laura Neff, Kee Ely, and Bill Thompson. The NS16000 family was developed by the NSTA design center.

REFERENCES

1. National Semiconductor. *NS16000 Programmers Reference Manual.* Santa Clara, California: National Semiconductor, 1981.
2. Kaminker, Asher, Leslie Kohn, Yoav Lavi, Avraham Menachem, and Zvi Soha. "A 32-Bit Microprocessor with Virtual Memory Support." *IEEE Journal of Solid State Circuits,* SC-16 (1981), pp. 548–557.

SOFTWARE ENGINEERING

Exploiting parallelism for the performance enhancement of non-numeric applications

by DAVID J. DEWITT and DINA FRIEDLAND
University of Wisconsin
Madison, Wisconsin

ABSTRACT

In this paper we examine the design of computer architectures that use parallelism to enhance the performance of non-numeric applications. In particular we examine how the technology of mass-storage devices has affected the design of computer architectures for non-numeric processing.

INTRODUCTION

In this paper we examine the design of computer architectures that use parallelism to enhance the performance of non-numeric applications. While this has been an active area of research for over ten years,[1] no commercial products that exploit parallelism have resulted from these efforts. This is especially interesting when one considers the wide range of products available commercially for numeric applications that use parallelism. At the top-end (in terms of both performance and price) are large scientific processors such as the Cray-1. At the bottom end (in terms of price) are products such as the Floating Point Systems attached pipelined processor.

Why is this the case? There seem to be several possible explanations. The first is that the problems associated with parallelism for non-numeric computation have not received the same amount of attention as the numeric area. Considering the large number of papers on "database machines," this is not a very plausible explanation. There are, however, two plausible ones. The first is that while there has been a constant demand for "more cycles" for numeric computations from a variety of user groups (e.g., geologists at oil companies doing seismic data analysis, physicists at Lawrence Livermore Laboratory solving large sets of simultaneous differential equations), a similar market has never developed for non-numeric applications. Another feasible explanation is that the recent advances in technology have not really helped make highly parallel non-numeric processors economically viable.

In this paper we intend to present and discuss a number of alternative architectures that have been proposed for non-numeric computation. While we will not present a performance evaluation of these architectures, we have performed a number of such studies recently.[2,3,4,5] Instead, we intend to summarize the results from these studies in describing the problems associated with the architectures that have been proposed. We begin by characterizing those operations that must be efficiently supported by a non-numeric processor (regardless of whether parallelism is used). The impact of limited I/0 bandwidth on parallel architectures for non-numeric processing is discussed in the third section. In the fourth and fifth sections, we describe a number of architectures that have been proposed to increase the total I/0 bandwidth in the system. A class of architectures referred to as "on-the-disk" machines are described first. Processing "complex" non-numeric operations such as sorting requires a different type of machine. We discuss several alternative organizations for machines of this type in the fifth section. Finally, we present our conclusions and suggestions for future research.

OPERATIONS OF NON-NUMERIC PROCESSING

Non-numeric databases can be naturally divided into two classes: (1) those that contain unformatted documents such as law cases and (2) those that contain formatted data such as inventory records. In this section we will describe the types of operations that must be supported by non-numeric processors for databases of both types.

Unformatted data—text retrieval systems

Processing unformatted data is both simple and complex. It is simple in the sense that what is required is to make a sequential pass through a database searching for those records (documents) that have a certain property. For example, in an online medical database containing all research reports dealing with cancer, a researcher might ask a text retrieval system to retrieve all documents that mention the word "liver."

Processing queries can become significantly more complex, however. Consider the query "retrieve those documents that mention the liver and contain a reference to either the chemical 'carbon tetrachloride' or 'perchloroethylene'." Processing this query is difficult because the reference to one of these chemicals might appear either before or after the reference to the liver and, in general, the references may be separated by an arbitrary amount of text. Furthermore, the system must be aware of a number of different delimiters (e.g., paragraphs, chapters, etc.), so that it does not retrieve a document when the query is satisfied by strings outside of the appropriate context.

The number of documents that must be searched to process a query can be reduced by maintaining secondary indices[6] on certain key words or phrases such as "organ type." However, simply storing these indices can become prohibitive when indices must be maintained for a large number of different keywords. Unless users access the database using only a limited number of key words (i.e., organ names or chemical names), secondary indices are not viable. Thus, since processing a query involves primarily a linear scan of the database, application of parallel processing techniques appears to be a very promising approach to providing fast access to very large unformatted databases. In the section on search machines, we will describe several architectures that are well suited for processing unformatted queries.

Formatted data—database systems

While a number of different data models have been proposed for describing the logical organization of a formatted

database,[6] most researchers who have designed special purpose architectures for enhancing database system performance have assumed the relational data model. This has occurred primarily because of the regularity of the relational data model for specifying information on both entities (e.g., parts, suppliers) and relationships between entities (e.g., which parts a supplier supplies).

A relational database[7] consists of a number of normalized relations. Each relation is characterized by a fixed number of attributes and contains an arbitrary number of *unique* tuples. Thus, a relation can be viewed as a two-dimensional table in which the attributes are the columns and the tuples are the rows. In a relational DBMS, relations are used to describe both entities and relationships between entities. Figure 1 shows a simple database that describes information about suppliers (relation *S*), parts (relation *P*), and the association between suppliers and the parts they supply (relation *SP*).

The operations supported by most relational database systems can be divided into two classes according to the time complexity of the algorithms used on a uni-processor system. The first class includes those operations that reference a single relation and require linear time (i.e., they can be processed in a single pass over the relation). The most familiar example is the selection operation which selects those tuples from a relation that satisfy a simple predicate (e.g., suppliers in "New York").

The second class contains operations that have either one or two input relations and require nonlinear time for their execution. An example of a relation in this class that references one relation is the projection operation. Projecting a relation involves first eliminating one or more attributes (columns) of the relation and then eliminating any duplicate tuples that may have been introduced by the first step. Sorting [which requires $0\ (n \log n)$ time] is the generally accepted way of eliminating the duplicate tuples. The join operation is the most frequently used operation from this class that references two relations. This operation can be viewed as a restricted cross-product of two relations. A join would be used by a user to find the name and address of all suppliers who supply a particular part (see Figure 1).

S RELATION

S#	SUPPLIER-NAME	ADDRESS
17	JONES	MADISON

P RELATION

P#	PART-NAME	COLOR	WEIGHT
36	STOVE	YELLOW	300

SP RELATION

S#	P#	QUANTITY-ON-HAND
17	36	13

Figure 1—Supplier-parts relational database

The problem of size

The very size of both formatted and unformatted data sets introduces two important problems. First, operations on both types of database are almost always I/0 and not CPU intensive. For certain complex search conditions on both unformatted and formatted databases, a rather fast CPU may be required to process the query at the speed of the mass-storage device. However, as we will discuss in next section, during the last ten years improvements in technology have had a much more dramatic effect on performance of low-cost processing units (e.g., the Motorola 68000) than the bandwidth of I/0 devices. We contend that even this type of query should be considered to be I/0 limited rather than CPU limited.

The second effect of very large data sets is that the algorithms used must be "external" algorithms. For example, in designing a machine that can rapidly sort very large data files, one must never make the assumption that enough main memory will be available to permit the whole file to be brought into main memory and sorted. For a single processor, the algorithm normally used is an external merge sort.[8] It is important to realize that the requirement that external algorithms be used has the side effect of increasing I/0 activity. Later we will examine several architectures that employ parallelism for processing complex operations in order to minimize the number of I/0 operations performed.

The algorithms for processing selection operations on formatted or unformatted databases are naturally external algorithms as the normal mode of execution is to read the next block of data from mass storage and then apply the search condition to it. Since these queries can always by processed in one pass through the database, they are, when measured in terms of I/0 activity, significantly simpler than sorting a very large data file. Thus a different class of architectures that exploit parallelism are appropriate. These architectures will be discussed in the fourth section.

THE PROBLEM OF I/0 BANDWIDTH

The key factor in limiting development of parallel processors for non-numeric computation is the lack of sufficient I/0 bandwidth from commercially available mass-storage devices. Consider, for example, a typical (for 1981) mass-storage device such as the IBM 3350. Each track on this device holds 19,069 bytes, and a revolution takes 16.7 ms. Thus, the maximum burst bandwidth of the device is 1.144 Mbytes/second. This represents an improvement of only 46% in I/0 bandwidth when compared with the IBM 3330—a drive that was introduced over ten years ago.

As was discussed briefly in the previous section, executing a sequential search on a formatted or unformatted database is more likely to be CPU limited than executing an external algorithm for a complex operation (e.g., a join) on a formatted database, since the first task requires only *N* I/0 operations (one for each of the *N* blocks of data) while the latter requires, in general, on the order of *N*log*N* I/0 operations. It is thus most appropriate to examine the level of CPU performance necessary to process data at the rate of 1.144 Mbytes/second.

At a data rate of 1.144 Mbytes/second, a processor that is directly attached to a disk controller has approximately 0.87 microseconds to process each incoming byte. If one assumes that it takes two instructions to process each byte (one instruction both to compare the next byte of the incoming data stream with the next byte of the search pattern and to do an autoincrement of both pointers, and a second instruction to test for both loop termination and branch on failure), then the processor must be approximately a 2.3 MIP processor. While this is significantly faster than a conventional microprocessor such as a Motorola 68000 (which is about a 1 MIP processor), it is certainly within the range of a simple processor constructed using bipolar bit-slice technology. If one insists on using "off the shelf" hardware, one might use three 68000 type processors each with an associated track-long buffer to process the incoming stream of data. Since each processor would now have to process only every third track, the resulting performance should be more than adequate.

For formatted databases, every byte of the incoming data stream does not need to be examined, since the offset of the fields to be examined are located at known positions in the incoming records. Thus, a single processor with two track-long buffers would be able to process selection queries at the data rate of an IBM 3350 disk. While the disk is filling one buffer, the processor can be applying the selection criterion to the records in the other buffer. Processing a record involves simply indexing into the record (by adding the offset of the field in the record) and then applying the selection condition to the field. As long as the number of bytes to be processed does not exceed approximately one-third of the record, one processor should be able to process the records as fast as they are read off the disk.

The point that we are trying to make with these examples is that one needs, at most, two to three processors and not tens or hundreds to process data at the rate of present conventional disk drives. Thus, an architecture that blindly uses a lot of processors (no matter how they are interconnected) to process data that resides on few standard disk drives will inevitably be I/0 bound. Advances in disk technology hold only limited hope for a resolution of this problem. Over the past ten years, increased disk capacities have been mainly the result of an increased number of tracks per surface, rather than an increased number of bytes per track. It is only with the recent announcement of the IBM 3380 disk drive that this situation has changed significantly. The 3380, with a revolution time of 16.7 ms. and a track capacity of 47,476 bytes, has a burst data rate (assuming no seeks) of 2.85 Mbytes/second. While this is a significant improvement, it still limits the number of processors that can be effectively used to the range 6–8.

In the next two sections, we will examine several architectures that have been proposed for increasing the effective I/0 bandwidth. In particular, we show that the architectural solutions for processing selection operations are different than those for processing complex operations on formatted databases.

SEARCH MACHINES—"ON THE DISK MACHINES"

Architectures for parallel processing of selection operations on formatted or unformatted data consist of two basic components: (1) a processing element and (2) a storage unit. A number of different storage technologies can be used as the basis of the storage unit, including bubble memories, charge-coupled devices, or conventional magnetic disks. Since the logical organization of the processing elements is independent of the storage medium used, we shall assume that the storage unit is a track on a magnetic disk drive and that records are stored bitwise along the track.

There are two radically different approaches for designing the search engine. The first is to use a "conventional" processor such as a Motorola 68000 or a simple processor constructed using bit-slice components. The query to be executed is first compiled (by some host processor) into the machine language (or microcode) of the processor. Next the compiled query is loaded into the memory of the processor and then executed by applying the query to the storage unit associated with the processing element.

An alternative approach is to construct a special purpose processor[9, 10] that behaves like a finite state automaton (FSA). In this approach the query to be processed is compiled into a state-transition matrix that is then loaded into the memory of the FSA. For each incoming byte, the FSA computes a next state based on its current state and the value of the incoming byte. Since processing any query requires only one transition for each byte in the data stream, processing an arbitrarily complex query can always be done at the speed of the data stream. The main disadvantage of this approach is that the state-transition matrix that represents the compiled query is rather large [number of rows (states) * number of different incoming symbols].

In the following sections we shall describe three ways of organizing processing elements and storage units. For each design, we have assumed that the processing elements are connected to a host processor. This processor serves two important functions. First, it accepts and compiles queries from the users of the system. Second, as we shall discuss below, it can be used to assist in the execution of certain queries that are too complex for the processing elements to handle alone.

Processor-per-track (PPT) machines

In the first class of search machines, each storage unit has its own processing element associated with it as shown in Figure 2. The processing element scans the data as the track rotates and places selected records in an output buffer associated with the head. After a buffer fills, additional logic attempts to place its contents on the output bus for transmission to the host. In the event that the processing logic is not able to output a selected record (because the bus is busy and the temporary storage buffers are full), processing is suspended until at least one buffer is emptied.

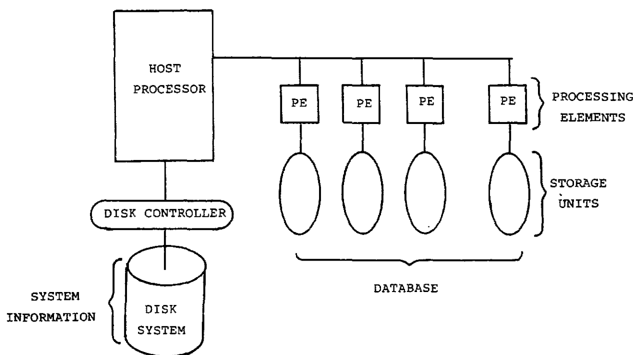


Figure 2—PPT design

PPT machines were pioneered in 1970 by Slotnick,[1] who suggested using first a track of a fixed head disk as the unit of storage and includes machines proposed by Parker,[11] Minsky,[12] and Parhami.[13] Two early database machine designs, RAP[14, 15] and CASSM,[16] also belong to this class. Two versions of RAP, each having two processing elements and two storage units, were constructed using first a fixed head disk and then charge coupled devices.

Because the entire database can be searched in one revolution of the storage units, this architecture is, at first glance, very appealing. However, it has a number of serious flaws. First, if magnetic disk technology is used as the basis of the storage unit, an extremely large number of storage units and cells are needed. Assume, for example, a state-of-the-art track capacity of 50,000 bytes. To have the same storage capacity as a conventional 300 Mbyte disk drive (which can hold only a very small database), 6,000 storage units and processing elements would be needed. The resulting system would most likely be relatively unreliable. A second major problem with this approach is that the bus connecting these 6000 processing elements to the host can easily become a bottleneck. Assume that the bandwidth of the output bus is 10 Mbytes/second. If the storage units have a revolution time of 16.67 ms., the maximum aggregated bandwidth of the processing elements is 18 billion bytes/second. Even if only 0.1% of the data stored in each storage element satisfies the search condition, the processing elements will have an output bandwidth of 18 Mbytes/second and hence the bus will still be a bottleneck.

The PPT architecture appears to be feasible only if the capacity of each storage unit is much larger (with a proportionally longer revolution time). The consequences would be twofold. First, the number of units required to store a "large" database could probably be reduced to a "reasonable" level. Second, a reduction in the number of processing units would ease the output bus bottleneck problem. There are two techniques of increasing storage unit capacities. One is to use a different storage medium such as bubble memories (some of which have a capacity of one million bits each[17]). While the performance of such an approach is excellent,[2] most manufacturers are dropping bubble memory components because their cost per bit has not become competitive. A second approach is to associate each processing element not with a single track of a disk, but rather with an entire surface of a disk. This approach is discussed in the following section.

Processor-per-head (PPH) machines

In the second class of search machines, each processing element is associated with a head of a moving-head disk as illustrated in Figure 3. Thus, the storage unit consists of all the tracks on the surface of a disk instead of a single track. In this class of machines, data is transferred in parallel over 1-bit wide data lines from the heads to the processing elements. Each processor applies the selection criteria to its incoming data stream and places selected records in its output buffer. In such an organization, an entire cylinder of a moving-head disk is examined in a single revolution (assuming no output bus contention). As in PPT organizations, additional revolutions may be needed to complete execution of the query if an output buffer overflows.

The DBC[18] database machine project adopted the PPH approach over the PPT approach as the basis for the design of the "Mass Memory Unit" because PPT devices were not deemed to be cost-effective for the storage of large databases (say more than 10^{10} bytes). Another possible reason for taking this route is the apparent lack of success of head-per-track disks as secondary storage devices. Moving-head disks with parallel readout, on the other hand, seemed an attractive and feasible alternative. The Technical University of Braun-

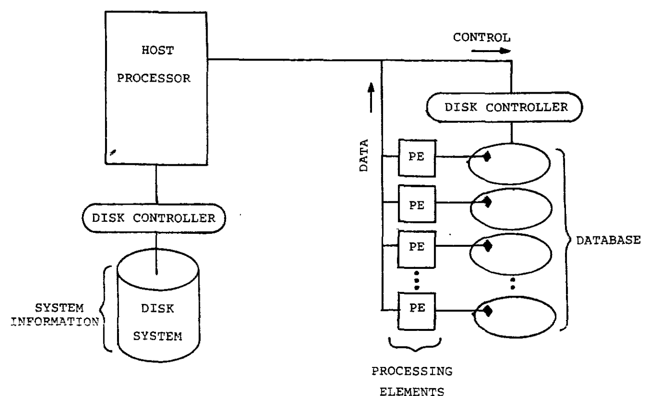


Figure 3—PPH design

schweig, in cooperation with Siemens, has actually built one for use in the Braunschweig search machine SURE.[19] It is the case, however, that parallel readout disks are presently not widely available (a 600-Mbyte drive with a 4-track parallel readout capability and a data transfer rate of 4.84 Mbytes/second is, however, available for the Cray-1 for approximately $80,000 without controller) and may never be cost-effective for the storage of large databases (when compared to "standard" drives).

Processor-per-disk (PPD) machines

Unlike the PPT and PPH approaches, the PPD organization utilizes a standard disk drive. In this organization, a processing element is placed between the disk and the memory device to which the selected records are to be transferred as shown in Figure 4. This processor acts as a filter[9] to the disk by forwarding to the host only those records that match the selection criteria. At first glance, it seems as though this approach is so inferior to the others that it does not merit any attention. However, its advantage is that for a relatively low price one can obtain the same functionality (but not the same performance) as the PPT and PPH designs. In addition, by broadcasting the data stream from the disk to a set of processing elements,[19] one can simultaneously process selection queries from different users over the same database.

Use of Search Machines for Complex Database Operations

Since the only functionality provided by the PPT, PPH, and PPD designs is to process selection queries, each of these designs processes complex operations (e.g., join) on formatted databases by decomposing the query using an algorithm based on Wong's tuple substitution algorithm.[20] Assume that the join operation as specified by the user has the form $R.a = S.b$ and that S contains fewer tuples than R. These designs will process this query by issuing one selection subquery for each tuple in S. The form of each subquery will be $R.a = x$ where x is the join attribute value from the current tuple in S. The result relation is produced by having the host

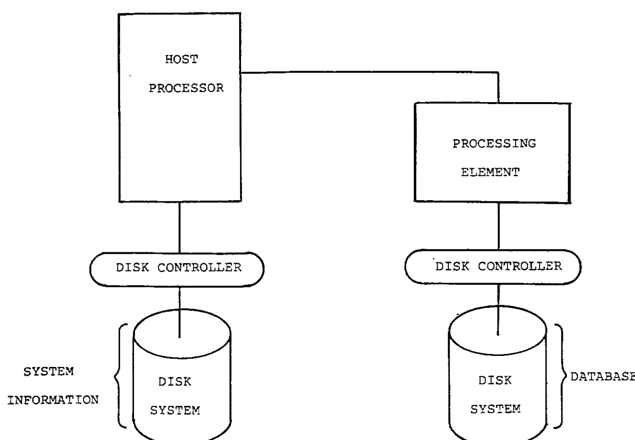


Figure 4—PPD design

processor "join" each tuple in S with all the tuples from R returned by the execution of its subquery. The performance of this approach, however, is very poor.[3] In fact, it is inferior to using only the host processor to perform a traditional "sort-merge" join. In the next section, we will describe several non-numeric architectures that are designed to process complex database operations effectively.

MACHINES FOR COMPLEX DATABASE OPERATIONS

Parallelism can also be employed to enhance the performance of complex database operations. The use of parallel processors to enhance the performance of the relational join and project operations is highly desirable, since both are very time-consuming in a conventional database management system.

As discussed by DeWitt and Hawthorn[3] and Boral, DeWitt, Friedland and Wilkinson,[21] we strongly advocate an "algorithmic approach" to the design of architectures that use parallelism to enhance the performance of non-numeric operations. As an example of a complex operation, we present the relational join operator in the following section. Afterwards, we discuss four alternative building blocks that can be used as the basis for parallel algorithms for complex formatted database operations. Then we describe parallel architectures based on two of these building blocks. Because a performance analysis of these alternative algorithms and architectures is beyond the scope of this paper, the interested reader is encouraged to examine Boral, DeWitt, Friedland and Wilkins[21] and Friedland.[4]

Unlike the selection operation, for which a partition of the data can be searched independently by each processor, the join, projection, and the other complex operations require that the processors exchange data among themselves during intermediate stages of execution. In addition, the volume of I/0 activity is significantly higher, since unlike the selection operation, these operations cannot be performed in a single pass over the operand relations. While the building blocks vary in terms of the degree of interprocessor communication and amount of I/0 activity each requires, the architectural features that are needed to efficiently execute these tasks include a fast interprocessor communication facility and a cost-effective, mass-storage device that provides high I/0 bandwidth.

The relational join operation

In Figure 5 the execution of the join operation is illustrated by the "nested-loops" algorithm. Relations R and S are assumed to have N and M tuples respectively, and fields r of relation R and s of relation S are assumed to be of the same type. Execution of the join requires that the r field from each tuple of R be compared to the s field from each tuple of S. When they match, the two tuples are concatenated. In general, each tuple in R will match an arbitrary number of tuples in S. In the worst case, the join of R and S will produce $N*M$ tuples.

```
for i <- 1 to N do
   for j <- 1 to M do
      if R_i.r = S_j.s then
      begin
         t = join(R_i,S_j);
         output(t);
      end;
```

Figure 5—The nested-loops join algorithm

Building blocks for complex database operations

While the nested-loops join algorithm is adequate for "small" relations, it is a very slow method joining two "large" relations. If indices exist on the join attributes for both relations, then the join can instead be performed on the index entries (instead of the tuples themselves).[22, 23] This procedure substantially reduces the volume of I/0 activity, since the index files are much smaller than the data files. If indices are not available, the join of two "large" relations is usually performed by presorting the operand relations. This process is followed by a modified merge phase in which matching tuples are located and joined. After the relations have been sorted, the actual join step requires only a linear pass through both relations. Sorting is also used by conventional database management systems to efficiently perform the duplicate elimination part in the projection. Alternatively, hashing[24] can be employed to eliminate duplicate tuples in a projected relation.

There appear to be four building blocks that can be used as the basis for parallel algorithms (and parallel machines to execute these algorithms) for complex database operations. They are

1. Indexing
2. Hashing
3. Sorting
4. Broadcasting

Of these four techniques, sorting and broadcasting appear to us to be the most promising. In the following paragraphs we will describe a number of unresolved problems associated with using indices and hashing as basic building blocks. In both cases we shall use the join operation as a vehicle for explaining our objections.

The basic idea of using indexing[25] in a parallel algorithm for the join operation is to have both the indices and the corresponding relations uniformly partitioned among the processors, each of which has a mass-storage device associated with it. During execution of the algorithm, each processor uses its portion of the indices to compute its portion of the result relation. The fundamental flaw of this approach is that it assumes that if you uniformly distribute the tuples from both relations among all the processors, the two tuples from R and S with the same attribute value will end up on the same processor.

Goodman and Despain[25] also propose using hashing as the basis for a parallel join algorithm. Again the two relations are assumed initially to be uniformly distributed among all the processors. The algorithm begins by applying a hash function to the "joining" attribute of both relations. This hash function is used to compute a processor number. After the appropriate processor has been identified, the tuple is sent to the processor to be joined with the tuples with the same attribute value. During the second phase of the algorithm, each processor joins the tuples it receives as the result of the first phase.

There are a number of problems associated with this algorithm. First note that the number of distinct join attribute values will, in general, be much larger than the number of processors available. Thus, each processor will receive tuples with a range of join attribute values. As a consequence, the processor must somehow (e.g., by sorting) determine exactly to which tuples every tuple is to be joined. A second problem is the cost of moving almost every tuple to a different processor from where it originally resided. While both of these problems have solutions, a serious problem that is impossible to resolve remains. The idea of using a hash function in the first place was to distribute the tuples in a manner that would permit all processors to help perform the actual join operation. However, when there are only a few join attribute values, only a few processors will be used to actually perform the join. In general, since the distribution of join attribute values is unknown, the performance of a machine designed around this algorithm will be unpredictable (an unsettling fact).

An architecture based on parallel sorting

Until recently,[4] parallel sorting had not been suggested or investigated as the basis for performing complex database operations. This can be partly explained by the lack of research in the parallel external sorting area. While extensive literature on parallel sorting exists, no algorithms had been developed for sorting in parallel large mass-storage files. When one considers how successful sorting is for performing complex operations on a single processor, it is natural also to examine its use for parallel architectures.

Friedland[4] has examined and analyzed a number of external parallel sorting algorithms and architectures. The algorithms that display the best performance are based on either a two-way external merge step[26] or an extension of Batcher's bitonic sort[27] to permit the sorting of external data files. In general case, the algorithm based on the two-way external merge step, termed the *parallel binary merge* algorithm, has the best performance. In this section we will describe the operation of this algorithm on an architecture that permits the efficient execution of the algorithm. Use of this algorithm for performing the join would occur in two steps. First, both relations would be sorted on the joining attribute. Next, one processor would be used to join the two relations by reading the sorted files in a sequential pass through them. Tuples from the two relations with equal joining attribute values would then be joined.

Execution of the parallel binary merge algorithm is divided into three stages as shown in Figure 6. We assume that the number of pages to be sorted, N, is *at least* twice the number of processors, P. The algorithm begins execution in a suboptimal stage in which sorting is done by successively merging pairs of longer and longer runs until the number of runs is

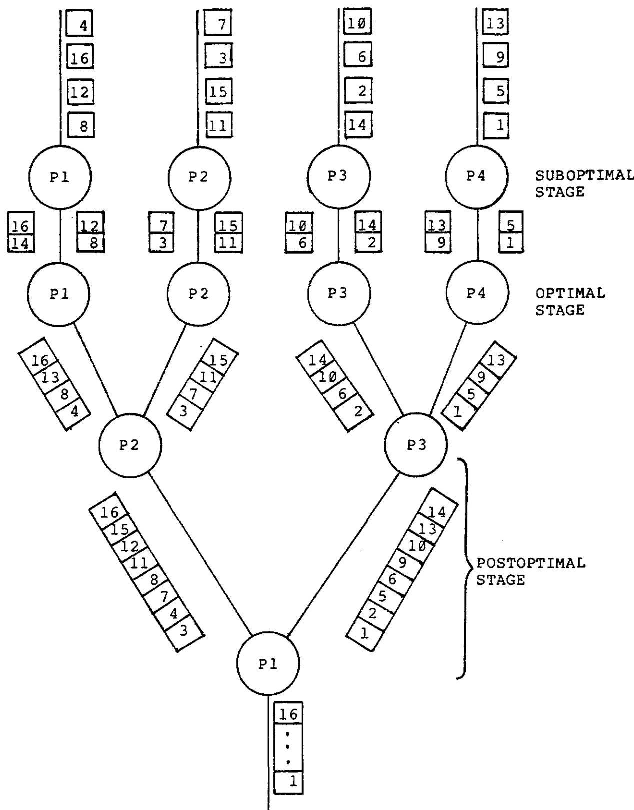


Figure 6—Parallel binary merge with 4 processors and 16 pages

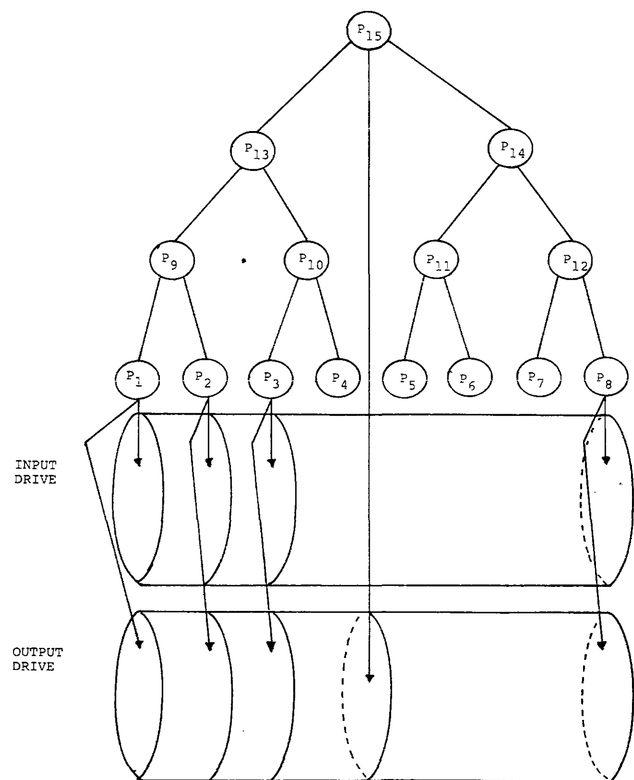


Figure 7—Binary tree interconnection between the processors

equal to twice the number of processors. During the suboptimal stage, the processors operate in parallel but on separate data. First, each of the P processors reads two pages and merges them into a sorted run of two pages. This step is repeated until all single pages have been read. If the number of runs of two pages is greater than $2*P$, each of the P processors proceeds to the second phase of the suboptimal stage in which it repeatedly merges two runs of two pages into sorted runs of four pages until all runs of two pages have been processed. This process continues with longer and longer runs until the number of runs equals $2*P$.

When the number of runs equals $2*P$, each processor will merge exactly two runs of length $N/2P$. We term this phase the optimal stage. At the beginning of the postoptimal stage, the controller releases only one processor and logically arranges the remainder as a binary tree (see Figure 6). During the postoptimal stage, parallelism is employed in two ways. First, all processors at the same level of the tree (Figure 6) execute concurrently. Second, pipelining is used between levels. By pipelining data between levels of the tree, a parent is able to start its execution a single time unit after both its children (i.e., as soon as its children have produced one output page).

The ideal architecture for the execution of this algorithm is a binary tree interconnection between the processors as shown in Figure 7. The mass-storage device consists of two disk drives, and each leaf processor is associated with a surface on both drives as in the PPH organization. This organization permits the leaf processors to do I/0 in parallel while reducing almost in half the number of processors that actually must do input/output. Analysis of the performance of this architecture shows substantial performance improvements (over a single processor) when such a mass-storage device is available. If, however, only two conventional disk drives are available, the performance improvement achieved is very limited, which indicates again the need for more I/0 bandwidth if parallel architectures for non-numeric applications are to be viable.

An architecture based on broadcasting

By using broadcasting as a basic building block, a number of very simple and relatively fast parallel algorithms for complex database operations can be constructed. As an example, consider the nested-loops join algorithm shown in Figure 5 and assume that N and M refer to the number of disk pages occupied by relations R and S, respectively, instead of the number of tuples. Also assume (for the moment) that N processors are available. A parallel nested-loops join algorithm would begin by having each of the N processors read one page from relation R. Next, the M pages of relation S are *broadcast,* one at a time, to all the processors. Upon receipt of a page of S, each processor will apply the nested-loops join algorithm to its page of R and the incoming page of S. If $P < N$ processors are available, the algorithm is executed in N/P stages.

Furthermore, as shown in Figure 8, the architecture required to support these algorithms is straightforward. All that

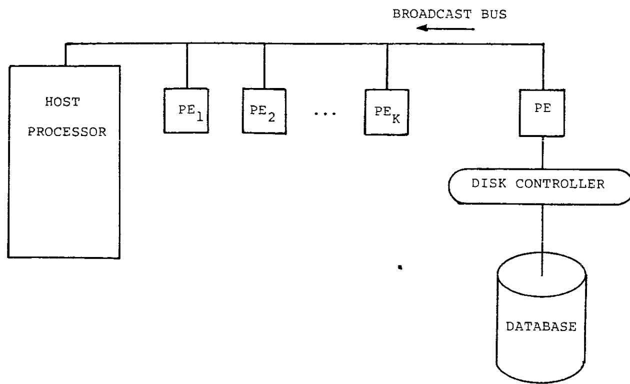


Figure 8—An architecture based on broadcasting

is required is a bus that is capable of broadcasting at the same data rate of the mass-storage device. The main disadvantage of this approach is that the first step of the algorithm in which each processor gets a page of R is sequential. It can, however, be parallelized if R resides on a parallel-readout disk drive.

CONCLUSIONS AND FUTURE DIRECTIONS

From the discussions contained in this paper we are able to draw a number of conclusions about parallel architectures for non-numeric applications. First, manufacturers must place more emphasis on the development of mass-storage devices that have higher transfer rates if parallel architectures for non-numeric applications are to become viable. Second, while we have suggested two different styles of parallel architectures for execute searching and complex operations, database management systems perform both types of operations. Thus, it appears worthwhile to investigate the design of a machine that combines the ability of the "on the disk" machines to process selection queries rapidly with those features that facilitate execution of complex operations on formatted databases. The RDBM[28] represents a first attempt at designing such a machine.

ACKNOWLEDGMENTS

We would like to acknowledge support of this research by the Department of Energy under contract #DE-AC02-81ER10920 and the National Science Foundation under grant MCS78-01721.

REFERENCES

1. Slotnik, D. L. "Logic per Track Devices." In Frantz Alt (ed.), *Advances in Computers* (Vol. 10), New York: Academic Press, 1970, pp. 291–296.
2. Boral, H., D. J. DeWitt, and W. K. Wilkinson. "Performance Evaluation of Four Associative Disk Designs." *Journal of Information Systems,* Vol. 7, No. 1, January 1982.
3. D. DeWitt and P. Hawthorn. "Performance Evaluation of Database Machine Architectures." Invited paper, *Proceedings of the 7th International Conference on Very Large Databases,* September 1981.
4. Friedland, Dina B. "Design, Analysis, and Implementation of Parallel External Sorting Algorithms." Computer Sciences Technical Report #464, University of Wisconsin, January 1982.
5. Hawthorn P. and D. J. DeWitt. "Performance Evaluation of Database Machines." IEEE Transactions on Software Engineering, January 1982.
6. Date, C. J. *An Introduction to Database Systems.* Reading, Mass.: Addison-Wesley, 1981.
7. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM,* Vol. 13, No. 6, June 1970.
8. Knuth D. E. *The Art of Computer Programming—Sorting and Searching.* Reading, Mass.: Addison-Wesley, 1975, p. 160.
9. Bancilhon F. and M. Scholl. "Design of a Backend Processor for a Data Base Machine." *Proc. of the ACM SIGMOD 1980 International Conference of Management of Data,* May 1980.
10. Haskin, R. L. "Hardware for Searching Very Large Text Databases." *Proceedings of the Fifth Workshop on Computer Architecture for Non-Numeric Processing,* March 1980, pp. 49–56.
11. Parker J. L. "A Logic per Track Retrieval System." IFIP Congress, 1971.
12. Minsky N. "Rotating Storage Devices as Partially Associative Memories." *Proc.* 1972 FJCC.
13. Parhami B. "A Highly Parallel Computing System for Information Retrieval." *Proc.* 1972 FJCC.
14. Ozkarahan, E. A., S. A. Schuster, and K. C. Smith. "RAP—Associative Processor for Database Management." *AFIPS Conference Proceedings* (Vol. 44) 1975, pp. 379–388.
15. Sadowski P. J. and S. A. Schuster. "Exploiting Parallelism in a Relational Associative Processor." Fourth Workshop on Computer Arch. for Non-numeric Processing, Aug. 1978.
16. Su, Stanley Y. W., and G. Jack Lipovski. "CASSM: A Cellular System for Very Large Data Bases." *Proceedings of the VLDB,* 1975, pp. 456–472.
17. Bryson, D., D. Clover, and D. Lee. "Megabit Bubble-Memory Chip Gets Support From LSI Family." *Electronics,* April 26, 1979.
18. Kannan, Krishnamurthi. "The Design of a Mass Memory for a Database Computer." *Proc. Fifth Annual Symposium on Computer Architecture,* Palo Alto, CA., April 1978.
19. Leilich H. O., G. Stiege, and H. Ch. Zeidler. "A Search Processor for Data Base Management Systems." *Proc. 4th Conference on Very Large Databases,* 1978.
20. Wong, E. and K. Youssefi. "Decomposition—A Strategy for Query Processing." *ACM Transactions on Database Systems,* Vol. 1, No. 3, September 1976, pp. 223–241.
21. Boral, H., D. J. DeWitt, D. Friedland, and W. K. Wilkinson. "Parallel Algorithms for the Execution of Relational Database Operations." submitted to *ACM Transactions on Database Systems,* October, 1980.
22. Astrahan, M. M. et al. "System R: Relational Approach to Database Management." *ACM Transactions on Database Systems,* Vol. 1, No. 2, June 1976, pp. 97–137.
23. Blasgen M. W. and K. P. Eswaran. "Storage and Access in Relational Data Bases." *IBM System Journal,* Vol. 16, No. 4, 1977.
24. Babb, E. "Implementing a Relational Database by Means of Specialized Hardware." *ACM Transactions on Database Systems,* Vol. 4, No. 1, March 1979, pp. 1–29.
25. Goodman, J. R. and A. M. Despain. "A Study of the Interconnection of Multiple Processors in a Data Base Environment." *Proceedings 1980 International Conference on Parallel Processing,* August 1980, pp. 269–278.
26. Even, S. "Parellelism in Tape Sorting." *CACM,* Vol. 17, No. 4, April 1974.
27. Batcher, K. E. "Sorting Networks and Their Applications." *Proceedings Spring Joint Computer Conference* (Vol. 32) 1968, pp. 307–314.
28. Hell, W. "RDBM—A Relational Data Base Machine: Architecture and Hardware Design." *Proceedings of the 6th Workshop on Computer Architecture for Non-Numeric Processing,* June 1981.

Performance engineering of software systems: a case study

by C. U. SMITH
Duke University
Durham, North Carolina
and
J. C. BROWNE
University of Texas at Austin
Austin, Texas

ABSTRACT

This paper summarizes the concepts of performance engineering in large software systems and illustrates the application of performance engineering techniques to the early design phase of a large database system.

Performance engineering is a methodology for evaluating the performance of software systems throughout their life cycles. The case study given here demonstrates that it is possible to predict resource usage *patterns* of complex software systems even in early design phases of the system, although detailed predictions of resource usage are not likely to be validated. The results presented here show the leverage of considering performance implications in the early design phases of a software project.

1.0 INTRODUCTION

This paper describes a performance engineering study of a large-scale software system. Performance engineering is a practical discipline used throughout the design and implementation stages of a system to ensure that it meets responsiveness and/or throughput goals. Performance engineering is routinely applied to mechanical devices and computer hardware, but it has been largely ignored in the realm of software engineering.

Performance engineering is, however, an emerging discipline. A methodology has been developed for the representation and evaluation of software systems, given specified workloads and specified hardware/operating system environments.[1,2] Smith[3] has defined a software engineering context for performance engineering. It has been established and validated that this methodology can generate accurate representations of the execution behavior of software systems, given implementation specifications and reasonable, measured input data.

This paper demonstrates the application of performance engineering in the early design phase of a large software project. It will be demonstrated that it is possible to assess accurately the principal resource usage patterns of a large and complex software system when only quite preliminary design specifications are available (provided that the target host hardware/operating system is known). The study reported here predicted the bottlenecks in performance that appeared more than a year later (fall 1979—prediction, spring 1981—execution). Detailed prediction of specific performance characteristics cannot be attained at the stage of preliminary design or even detail design. It is often the case that analysis leading to bottleneck predictions will result in design alterations, thus precluding detailed validation of predictions. The case study presented here underwent many substantive changes in design over the 18-month period immediately preceding implementation, but many major patterns of resource usage remained constant and were due to constructs identified in the early design phase of performance engineering. (The major deviations from these early predictions were also identified in the detail design and early implementation phases prior to execution.)

The example in this paper is a large database system intended for the support of integrated CAD/CAM applications. The complete performance engineering study is voluminous.[4,5] We use as an example the analysis of a small query transaction within a given hardware, operating-system, and database environment. It is an essential part of our purpose to establish that performance engineering is a low-effort/high-return enterprise. The work reported herein is a part of a project that was usually staffed by a half-time systems analyst. The part of the work reported here represents approximately one person-month of effort.

2.0 OVERVIEW OF PERFORMANCE ENGINEERING

Performance engineering includes all activities associated with constructing software to meet performance goals. It begins with the analysis of the preliminary software design to determine whether its performance characteristics appear to be satisfactory. It is far better to correct performance problems before code is written than to invest significant development effort in a product that is certain to have poor performance. Performance engineering continues throughout the software life cycle. It is important to monitor software development progress and to assess the performance impact of detail design decisions as well as design and implementation changes. It is important even after the implementation stage, during maintenance. Since maintenance usually includes software redesign to incorporate additional functions, the revised system is subject to the same performance pitfalls as the original design.

A performance engineering project consists of obtaining the necessary data, conducting the analysis, comparing results to performance goals, and evaluating design alternatives. All performance evaluation projects should also be concluded with a validation of the results to evaluate their effectiveness. These topics are discussed in the remainder of Section 2.

Performance engineering of software is most effective when conducted by a team supplying expertise in three areas: the intended use of the software (supplied by the client representative), its design (supplied by the software designers), and software performance analyses (supplied by the performance analysts). The team begins the performance engineering early in the design stage, just after the initial functional architecture is specified. The purpose of the first analysis is to rule out designs that are potentially disastrous, select a suitable one, and identify performance problems. It may also be used for capacity planning to determine the configuration that will be required to support the new product.

2.1 Performance Specifications

Seldom, at any development phase, is there sufficient documentation or written specification available to determine the performance characteristics of software systems. This information must then be obtained through *performance walkthroughs*. A performance walkthrough is similar in structure to the design and code walkthroughs characteristic of current software engineering practice.

The team, therefore, conducts a performance-oriented de-

sign walkthrough to gather the needed information. The client representative begins by describing a *typical* scenario or task that will involve the software from the user's point of view. The software designer then describes the processing necessary for accomplishing the user's task. The performance analyst participates in these two discussions and asks pertinent questions to obtain the data necessary for the analysis. The analyst then summarizes the scenario and software processing, from the analysis point of view, to verify that the information was correctly interpreted. An example of a walkthrough is given in Section 4.

The process is repeated for several representative scenarios; this may require multiple walkthroughs. The initial ones cover most of the necessary processing details; later ones cover only new information.

After the walkthrough, the analyst gathers the relevant information, conducts the evaluation, and presents the results and recommendations to the team. The team then decides on the most appropriate course of action.

Additional walkthroughs are conducted at later stages in the life cycle to update the information for the analyses, to compare the early results to the current results to see whether problems have arisen, and to look at the effect of replacing the usually optimistic assumptions used in early design with more realistic ones.

The preceding discussion addressed the methods of obtaining data for the analyses and following the software throughout the development process. An important component of performance engineering is the analysis *techniques* used to obtain predictions.

2.2 The ADEPT Analysis Technique

"A Design-Based Evaluation and Prediction Technique," ADEPT,[6] was used to evaluate the design in this case study. The following is a summary of the analysis steps:

1. Definition of the performance goals and the workload of the system, from the user's point of view.
2. Definition of the execution environment for the tasks. This may involve the specification of database characteristics and volumes as well as hardware and basic operating-system characteristics.
3. Definition of the system structure from a component or module viewpoint. The hierarchical development of system structure and the incremental resolution of components and modules are accommodated.
4. Mapping the workload onto the system structure to determine execution paths (execution graphs) that characterize the typical processing of this transaction in the given data/hardware/OS environment. Detailed explanations of execution graph concepts can be found in Smith.[6]
5. Specification of the resource requirements necessary for executing the components in the execution graphs for this workload.
6. Evaluation of the elapsed time and resources required for each module or component along the execution path(s). Techniques for evaluation of execution graphs can be found in Smith.[6]
7. Mapping of the execution graph to a queueing network model of the hardware and operating system to allow study of the execution of the workload with multiple users in the context of existing workloads and computer system environments.
8. Evaluation of the results of this analysis.

Alternatives are evaluated by revising the corresponding specifications and repeating the analysis steps.

A unique aspect of the ADEPT strategy is the initial use of a simplistic analysis of the performance in a *best-case* situation. More sophisticated analyses of realistic cases are introduced as more detailed information is available. The initial concern is to identify feasible designs and eliminate potentially disastrous ones. Since the analysis is based on estimates, a complex, costly, time-consuming initial evaluation is not justified.

The rationale for the best-case analysis is to focus attention on designs and potential improvements and to eliminate arguments about situations that are likely (or unlikely) to occur that would result in better (or worse) response times. For example, an analysis might show that the *average* response time is 25 seconds. Arguments could ensue about what particular combinations of input data will cause better (or worse) responses, when they will occur, and similar matters. But an analysis showing that the *best* response obtainable is 20 seconds clearly indicates problems. Any arguments about the likelihood of specific situations is irrelevant, since the response in those cases will be worse.

2.3 Validation Of Performance Engineering

Performance engineering has a unique problem: It will be impossible to demonstrate success, but its failures will be obvious. Successful performance engineering of a software system will lead to an effective and efficient product. Attribution of the success to performance engineering is, of course, unlikely! Products that perform poorly when used are clearly visible as failures. It is clear, therefore, that performance engineering as a discipline may gain more credibility from failures in which poor performance was predicted, but in which the predictions went unheeded, than from its true successes. A successful performance engineering project should be invisible at the conclusion of implementation, but *very* visible during design and development.

It should be clear that performance engineering of a software system can be initiated at early design stages and carried through the life cycle of the software product. The study described here began with early design and followed through until field task execution of code. Predictions developed at the early stage of the system that were unheeded will be shown to be qualitatively valid for the actual system representation that emerged.

3.0 PROBLEM DEFINITION

The subject system of this performance engineering study is the database component called the integrated program for information processing (IPIP) of the integrated program for

aerospace-vehicle design (IPAD) system. Information on IPAD can be found in the *Proceedings of the IPAD National Symposium*[7] and in reports available from NASA and the IPAD prime contract, the Boeing Commercial Aircraft Company. A schematic of the components of this system and their relationships are shown in Figures 1 and 2. The components illustrated in Figure 1 are the executive program, IPEX; the user interface program, IFAC; a user application program, APPL; and the data management system, IPIP. The IPIP data management system is a flexible and powerful system. It implements a multischema data-definition capability similar to that proposed in the ANSI-SPARC Three Schema Data Model.[8] The IPIP/IPAD system is written in Pascal. The early design representation of IPIP, shown in Figure 2, consists of a set of major functional modules. Figure 3 has the description of the functions of some of these modules. It is extremely large and has highly modular code. There are some one thousand Pascal procedures in IPIP itself. The original design called for the recursive use of the database system; i.e., the database system itself would be used to manage at run-time the definitions of data items, schema, and other structural information. This implies that a call to the data management system may involve multiple levels of recursion—first to locate the definitions of the data, then to obtain index directory information before actually going and searching for the data values themselves.

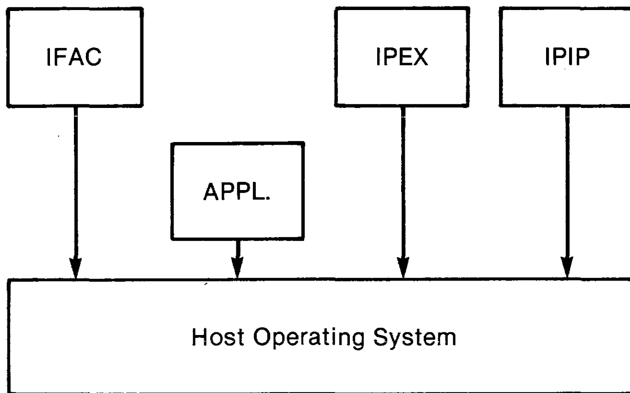


Figure 1—Structure of IPAD system

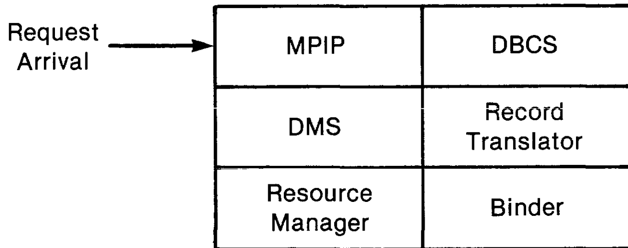


Figure 2—Partial list of IPIP major components at early design

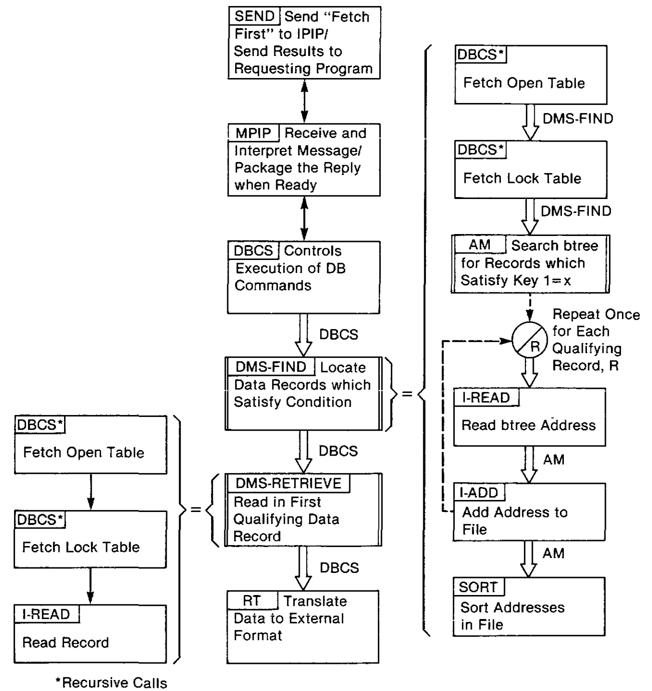


Figure 3—FETCH FIRST

The specific task used here as an example is the execution of a simplified transaction proposed as a part of the demonstration package for the IPAD system. This transaction establishes a list of record occurrences that meets given qualifications and delivers the first qualified occurrences to the user program. The database used here is an indentured parts list. This transaction against an indentured parts list could be part of the processing associated with a cost engineering study or manufacturing setup study. The actual processing steps included are three FIND FIRST commands, which qualify a list of record occurrences from the three record types of the database and retrieve the first occurrence that qualifies. These are followed by a series of FETCH NEXT commands (a total of 10), which deliver the balance of the qualified records. Figure 4 shows the steps of the transaction. The last two FETCH NEXT commands are repeated four and five times, respectively, before the lists of qualified records are exhausted.

This simplified transaction was selected as the basis for presentation of this case study because it is simple enough to be easily presented and because it includes the two most frequent operations of a database system, FIND and RETRIEVE. The fact that it includes both FIND and RETRIEVE operations makes it exercise a large fraction of the modules of the system. It will be seen that the consideration of only this single transaction brings to visibility many of the fundamental resource usage patterns of execution of the entire database system. The simplified transaction described omits from the analysis OPENing and CLOSEing of the schema and other low-frequency processing components.

Selection of workload elements for an analysis can have major impact on the cost/benefit ratio of a performance engineering study. It is important that elements selected be representative of the actual workload in order to obtain accurate predictions of the performance of the actual workload.

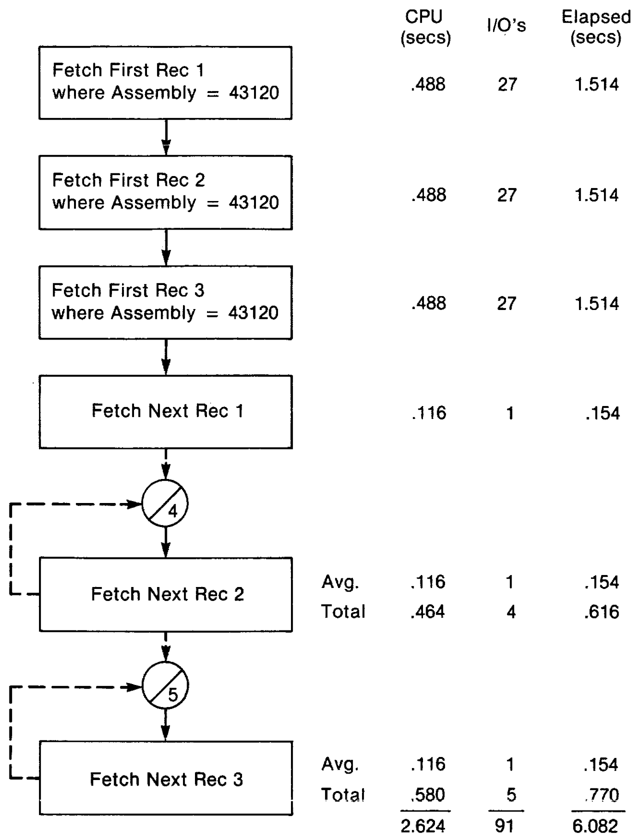


Figure 4—Scenario description and predictions

4.0 EXECUTION OF THE PERFORMANCE ENGINEERING TASK

4.1 Definitions And Specification Of Execution

The elements of the workload, the scenario containing FETCH FIRST and FETCH NEXT commands, were specified in the previous section. The next step is to determine the performance goals for the scenario. The demonstration program is a query that will determine and display a list of parts for the engineer (the user of the system). An engineer may be prepared to wait from 1 to 5 seconds to begin to see results. The hardware/software environment for this study was a Cyber 170 computer executing the NOS operating system.[9] Additional specifications were obtained from a performance-oriented design walkthrough. Figure 5 illustrates the contribution of an engineer who described the scenario. Figure 6 is a description of the processing steps required for the FETCH FIRST command. The resource estimates, obtained through questions posed by performance analysts, are in Figure 4.

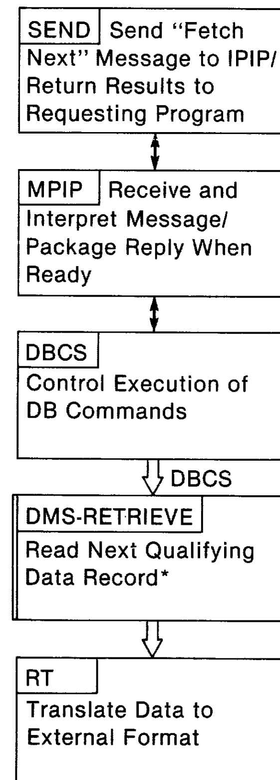
The data collected in the walkthrough were then collected, and the software structure was depicted by execution graphs. Figure 4 illustrates the processing steps in the scenario in the execution graph format, Figure 6 shows more detailed information for the FETCH FIRST command, and Figure 7 shows processing steps for the FETCH NEXT command.

This scenario is a query from a terminal asking for a list of all parts required to build a particular assembly. It contains multiple sub-assemblies each of which contains multiple parts. The query will be used to validate data and obtain information for inventory control.
The user enters the identification number of an assembly and indicates the type of information desired.
The data base is small and has a simple structure. There are three record types. The number of record occurrences that satisfy the queries is much smaller. For this example assume that we are interested in one assembly consisting of two sub-assemblies one of which contains three parts, the other four parts.

Figure 5—Sample contribution of the engineer

The query request is received from the terminal and sent to IPIP. MPIP is then called to interpret the message. Next, DBCS is called to process the request. It first calls DMS-FIND to locate the record occurrences which satisfy the request; then calls DMS-RETRIEVE to read in the first record; and finally calls Record Translator, RT, to convert the data to external format. The results are sent back through DBCS to MPIP for packaging, then to SEND to send the results to the requestor.

Figure 6—Sample contribution of the software designer



*See Graph for DMS-RETRIEVE in Figure 6

Figure 7—FETCH NEXT

The analysis of the graphs yields the predictions shown in Figures 4, 8 and 9. Many optimistic assumptions were made in this first analysis, such as the following:

1. Pre-runtime binding of data items and schema descriptions.

2. Optimal ordering of data items.
3. Processing-cost estimates that did not include memory management overhead.
4. Minimal depth of recursion.

Later analyses incorporated the additional processing required to include the above as the data became available and analyzable. This report focuses only on the analyses that could be made at early design.

Component	CPU estimate (secs)	#I/O's	Elapsed secs
SEND	.014		.014
MPIP	.045		.045
DBCS	.273	18	.957
DMS-FIND	.092	6	.320
DMS-RETRIEVE	.059	3	.173
RT	.005	—	.005
	.488	27	1.514

Figure 8—Prediction for FETCH FIRST

Component	CPU estimate (secs)	#I/O's	Elapsed secs
SEND	.014		.014
MPIP	.045		.045
DBCS	.005		.005
DMS-RETRIEVE	.047	1	.085
RT	.005	—	.005
	.116	1	.154

Figure 9—Prediction for FETCH NEXT

5.0 VALIDATION OF PREDICTED RESOURCE USAGE PATTERNS

The IPAD system has now been implemented. Measurements are available of execution and resource usage behavior on a demonstration closely resembling that on which the analysis of Section 4 was based. The very early runs of the demonstration were totally dominated by memory management overhead, which could not be modeled at the design phase described herein (although they were later analyzed and predicted). The resource usage and response times given in Section 4 display unacceptably long response time because of excessive CPU requirements.

The data in Table I compare the CPU time and the elapsed time for predicted and measured execution with memory management overheads factored out of the measured times and CPU times scaled to CPUs of the same speed. These elapsed times are for a single query executing on a dedicated computer system of approximately 5 MIPS processing power, a CDC Cyber 175. It is clear that our prediction of CPU bottlenecking was validated. In fact, the CPU processing requirements are much higher than our deliberately very optimistic estimates. These discrepancies are primarily due to detail design and implementation considerations not yet resolved at the time these estimates were made. The principal additional causes of CPU overhead included excessive procedure calls and elaborate procedures for allocation of resources. Analysis for this class of problems must await an appropriate stage of design and implementation.

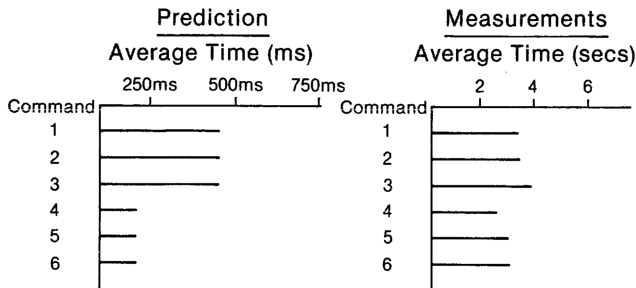
TABLE I—Actual times

Command		Total CPU	Total Elapsed	Without Memory Management Elapsed
Fetch first 1		3.5	13.1	8.0
Fetch first 2		3.5	13.2	8.1
Fetch first 3		3.6	13.2	8.1
Fetch next 1		2.2	5.6	4.0
Fetch next 2	avg.	2.6	7.5	5.0
	total	10.2	29.9	19.9
Fetch next 3	avg.	2.6	7.6	5.0
	total	12.9	38.0	25.2
		35.9	113.0	73.3

Recall that the purpose of this comparison of performance predictions to actual performance characteristics is to demonstrate that it is possible to identify unsatisfactory software design and the elements of the design that introduce problems early in the software development cycle before code is written. An integral part of the methodology used for predictions is the analysis of best-case performance, rather than average performance, because best-case analysis focuses attention on design problems rather than on model assumptions. Thus, actual performance characteristics will vary from predictions, for several reasons: (1) implementation details are not resolved at an early design stage, so optimistic assumptions are made for their resource requirements; (2) the best case is unlikely, so predictions will be low; and (3) many changes are made during the implementation stage that invalidate initial descriptions of the software. For these reasons it is important to monitor software development and continually update the model and to identify critical software components with respect to performance. These critical components should be implemented first and actual performance measurements substituted for early estimates to yield more realistic performance predictions. Note that the histograms of CPU and elapsed times for the scenario in Figure 10 are very similar; the difference is in the scaling factors. This supports the argument that both the critical resource and the critical components were identified at the early design stage.

A performance enhancement project for IPAD was initiated in April-May 1981. The data given here are from that period of the project. The performance enhancement project for IPAD has, in fact, now incorporated most of the recommendations resulting from the early performance engineering project. The data given in Table I actually reflect performance in the early phase of the performance enhancement project, in which many causes of poor performance were present. Performance has since been improved by approximately an

a) CPU Time Comparison



b) Elapsed Time Comparison

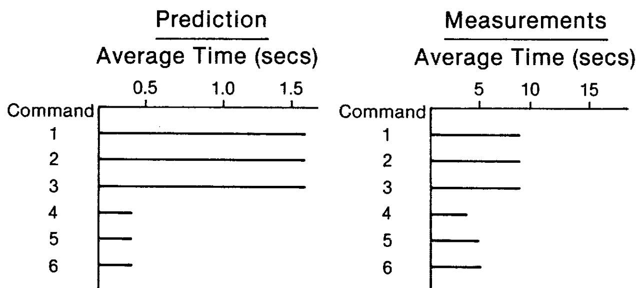


Figure 10—Histograms of predictions and measurements

order of magnitude, but the detail data necessary for comparisons are not available.

It is interesting to note that the principal reason for not correcting the performance problems identified during early design was that it would require effort and cause schedule slippage for delivery of the system. Performance problems that emerged during system integration testing were the primary cause of delays of many months in delivery of the software, because systems tests were slow and because a subsequent effort to improve performance took six to eight months. The problems can be attributed to decisions made both at the early design stage and at the implementation stage. Many of these decisions were analyzed and predicted to lead to performance problems before code was written. Some of the most serious problems, however, were due to low-level implementation decisions that were only detected in walkthroughs conducted during the integration testing stage. This indicates that closer scrutiny is required during implementation. Since the information required at that time is very detailed, an automated tool for gathering information is essential.

6.0 SUMMARY

This paper demonstrates through a case study the applicability and validity of performance engineering in the early design phase of software system development. The importance of carrying performance engineering through detail design and implementation is also stressed.

REFERENCES

1. Smith, C. U., and Browne, J. C. "Aspects of Software Design Analysis: Concurrency and Blocking," *Proceedings Performance '80,* Toronto, May 1980.
2. Smith, C. U. "The Prediction and Evaluation of Software from Extended Design Specifications." Ph.D. dissertation and Report TR-154, University of Texas at Austin, 1980.
3. Smith, C. U. "Increasing Information Systems' Productivity by Software Performance Engineering." *Proceedings XII CMG,* New Orleans, Dec. 1981.
4. Information Research Associates IPAD Report No. 1, October 1979. Information Research Associates, Austin, Texas.
5. Information Research Associates IPAD Report No. 2, August 1980. Information Research Associates, Austin, Texas.
6. Smith, C. U. "Development of a Tool to Support IPAD." Report No. CS-1981-2. Computer Science Department. Duke University. April 1981.
7. NASA IPAD Project Office, Langley Research Center, Hampton, Virginia. *Proceedings of the IPAD National Symposium,* Denver, Colorado, September 1980.
8. Tsichritsis, D., and A. Klug. *The ANSI/X3/SPARC Framework,* Montvale, New Jersey: AFIPS Press, 1978.
9. Control Data Corporation. *NOS Reference Manual.* Publication No. 60445300, CDC, Minneapolis, Minnesota.

A systolic processor for signal processing

by G. A. FRANK, E. M. GREENAWALT, and A. V. KULKARNI
ESL Incorporated
San Jose, California

ABSTRACT

A systolic array is a natural architecture for a high-performance signal processor, in part because of the extensive use of inner-product operations in signal processing. The modularity and simple interconnection of systolic arrays promise to simplify the development of cost-effective, high-performance, special-purpose processors. ESL Incorporated has built a proof of concept model of a systolic processor. It is flexible enough to permit experimentation with a variety of algorithms and applications. ESL is exploring the application of systolic processors to image- and signal-processing problems. This paper describes this experimental system and some of its applications to signal processing. ESL is also pursuing new types of systolic architectures, including the VLSI implementation of systolic cells for solving systems of linear equations. These new systolic architectures allow the real-time design of adaptive filters.

INTRODUCTION

A *systolic array* is a set of identical processing elements (called *cells*) with regular, nearest-neighbor interconnections and fixed data flow patterns. Systolic array architectures have been developed by H. T. Kung.[1,2,3] Systolic array architectures promise to be a cost-effective way to organize a large number of computational elements to make a high-performance data processor. Because they have regular interconnection patterns, performance can be increased at low cost by adding cells. They have fixed data flow patterns, which implies simplicity in control structures; data stream into and out of the boundaries of the array. As the data stream through the array, they are used by every processor that they reach. This implies maximum use of every data item fetched from memory. As a result of the regular data flow, systolic arrays fully exploit the potential for parallel-pipelined processing. Systolic architectures are well suited to VLSI implementation[1,3,4] owing to their regularity and their high performance in comparison to their I/O bandwidth.

The systolic processor developed by ESL is an experimental machine that performs three basic operations: matrix multiply, 1-D convolution, and 2-D convolution. It is programmed to support arbitrary sizes of problems, and it can process many data formats. It is a proof of concept model, so development risks were minimized by employing off-the-shelf components and using simulation to debug software and hardware independently.

SYSTOLIC ARCHITECTURE

The systolic architecture described in this section is a linear array that performs parallel and pipelined inner products as the essential operation. Figure 1 shows the configuration of a linear array and the cell structure.

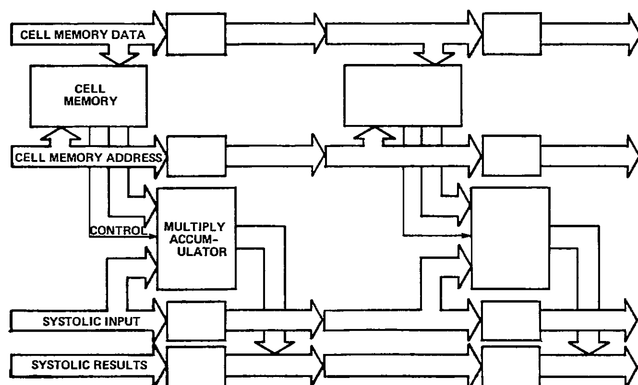


Figure 1—A linear systolic array and its cell

Each cell consists of a *multiplier-accumulator,* a *cell memory,* and three *latch registers,* one for each data stream. Three data streams are used by a cell: *data input, address input,* and *output.* The data input stream supplies one operand to the multiply-accumulator; the address input stream supplies addresses to the cell memory and the tag memory. The cell memory supplies the second operand to the multiply-accumulator. The tag memory supplies control to the multiply-accumulator to reset the accumulator and send results to the output stream. The data input and address input streams are passed through the cells unchanged; the output stream is either passed through intact or overridden with an output from the cell. Kulkarni and Yen[5] show that no valid result is overwritten. Two examples will be given to show how the array operates: matrix multiplication and 1-D convolution.

Matrix Multiplication

Table I is a list of snapshots of the array during the multiplication of the matrices

$$Y = AX$$

where A is a 2×3 matrix, X is a 3×2, and Y is a 2×2 matrix. This problem fits nicely on a 2-cell array, since the number of rows of A is 2. The table shows the data latched into each cell at selected periods during the multiplication. The ith row of A is stored in the memory of cell i. The rows are reused once for each column of X; this is accomplished by recirculating the data in the cell memories. The data in the cell do not move; instead the cell memory address is streamed through the cells systolically. The X matrix is streamed through the array in column order. Results are removed from the array by the output stream. Each result moves through two cells each cycle, so that the results are not overwritten and occur in the correct order.

TABLE I—Snapshots of a small matrix multiply

	Cell 1				Cell 2			
Time	Address	Input	Output	Memory	Address	Input	Output	Memory
0	0	$x1,1$		$a1,1$				
1	1	$x2,1$		$a1,2$	0	$x1,1$		$a2,1$
2	2	$x3,1$		$a1,3$	1	$x2,1$		$a2,2$
3	0	$x1,2$	$y1,1$	$a1,1$	2	$x3,1$		$a2,3$
4	1	$x2,2$		$a1,2$	0	$x1,2$	$y2,1$	$a2,1$
5	2	$x3,2$		$a1,3$	1	$x2,2$		$a2,2$
6	0		$y1,2$	$a1,1$	2	$x3,2$		$a2,3$
7					0		$y2,2$	$a2,1$

Convolution

One dimensional convolution is very similar to matrix multiply. In fact, the only difference if the array is large enough is the loading of the cell memories, and the fact that the input data stream comes from a vector rather than a matrix. Although for a matrix multiply each cell memory is loaded with a different row of the A matrix, for convolution all the cell memories are loaded with copies of the same kernel, appropriately rotated. Note that only four of the five results are generated by the pass described in Table II. A second pass would be required in order to generate the third result. This is part of the decomposition problem, fitting large problems on small arrays.

TABLE II—Snapshots of a small 1-D convolution

	Cell 1				Cell 2			
Time	Address	Input	Output	Memory	Address	Input	Output	Memory
0	0	$x1$		$w1$				
1	1	$x2$		$w2$	0	$x1$		
2	2	$x3$		$w3$	1	$x2$		$w1$
3	0	$x4$	$y1$	$w1$	2	$x3$		$w2$
4	1	$x5$		$w2$	0	$x4$		$w3$
5	2	$x6$		$w3$	1	$x5$	$y2$	$w1$
6	0	$x7$	$y4$	$w1$	2	$x6$		$w2$
7	1				0	$x7$		$w3$
8					1		$y5$	$w1$

2-D and multidimensional convolutions can be performed by loading the cell memories with the kernel weights appropriately shifted[5] and exploiting the multidimensional addressing capability of the local memory. ESL has also developed a scheme that uses multiple data streams for performing multidimensional convolutions.[6]

THE ESL SYSTOLIC PROCESSOR

The systolic processor developed at ESL is designed for experimentation with different applications and algorithms.[7] This requirement implies a great deal of flexibility in both software and hardware. Host software provides the experimenter with a simple view of the systolic processor by decomposing large problems and compiling the user's request into a program of systolic processor instructions. The systolic processor hardware includes programmable address generators that allow a single cell design to support a variety of algorithms including 1-D and 2-D convolution, matrix multiplication, and Walsh and Fourier transforms. The machine accommodates data representations that are widely used in image and signal processing: 8- or 16-bit input formats and 8-, 16-, 32-, or 48-bit result formats. The data can be processed in either of two modes: signed (twos-complement) or unsigned. The cells accumulate full-precision 42-bit results; final rounding or truncation is controlled by the user. The systolic processor instruction set includes operations for determining the most significant nonzero bit of a series' inner products; this information can be used to control rounding or truncation on the next pass.

The Experimental System

The demonstration model systolic processor (Figure 2) works as an attached processor to a host computer and is accessed from the host computer through a collection of FORTRAN subprograms. The *host interface* transfers data and commands to the systolic processor from the host and transfers results and status information from the systolic processor to the host. The *command dispatcher* stores systolic processor instructions in a command buffer and sends the instructions to other subsystems for execution. The *local memory* serves as a buffer to support the high-speed operation of the systolic array. Data are stored in the local memory of the systolic processor for repeated use.

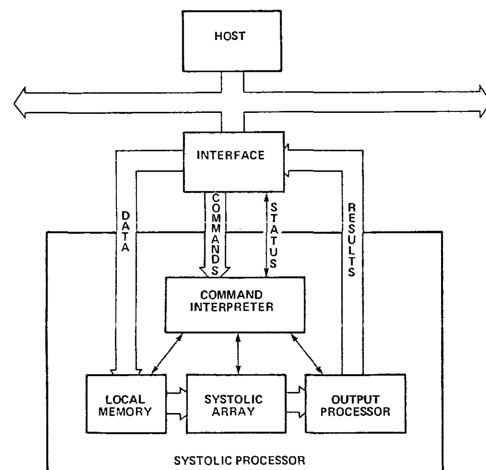


Figure 2—The experimental system

The *systolic array* is the computational unit of the systolic processor. It consists of an array controller and a linear array that can be configured with any number of *cells*. The cells do the computations; the array controller synchronizes the operation of the local memory, the cells, and the output processor. The *output processor* shifts and rounds the results according to the specifications supplied by the user and detects the maximum result value. This maximum value and its address in the result stream are returned to the user as status information that can be used to select the scaling parameter.

The *output buffer* supports the high-speed operation of the systolic array and allows the systolic processor to rearrange data before they are sent back to the host computer by a direct memory access (DMA) transfer. The output is double buffered, which allows the systolic processor to overlap the transfer of results back to the host with computation of the next block of results. Programmable *address generators* provide the address sequences for the local memory and the output buffer.

The Cells

The architecture of a cell is shown in Figure 3. Each cell consists of a *multiply-accumulate* chip; a *cell memory* with 1024 16-bit words; a *tag memory* with 1024 4-bit words; and three *latch registers,* one for each systolic stream that passes through the cell. Each of its cells can perform one 16-bit fixed-point multiplication and one full-precision (42-bit) accumulation every 200 nanoseconds, which gives each cell a maximum computational rate of 10 MOPS. Thus, a systolic array of 20 cells would have a maximum computational rate of 200 MOPS. The number of cells actually used in a computation is selected by the host software to match the needs of the computation. The cells are built with TRW multiply-accumulate chips. The use of multiply-accumulate chips allows cells to be compact, but it also requires that results be accumulated in each cell. The tag memory supplies the control signals to the cells. It is indexed by the same address used to index the cell memory.

System Software

There are three levels of software in the host system: the application programs, the decomposition routines, and the systolic processor driver. The host software routines and the major data structures are shown in Figure 4.

The user's *application program* in the host calls the decomposition routines that perform standard image or signal processing functions. The user program specifies the type of operation (e.g., 1-D convolution, 2-D convolution, or matrix multiply), the size of the problem, the formats (word size) of the inputs and outputs, and the shifting and rounding of the results.

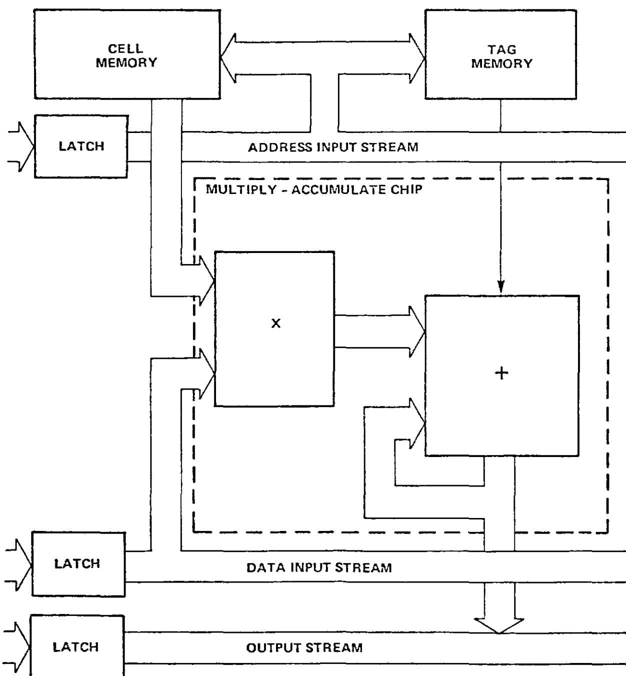


Figure 3—Systolic cell architecture

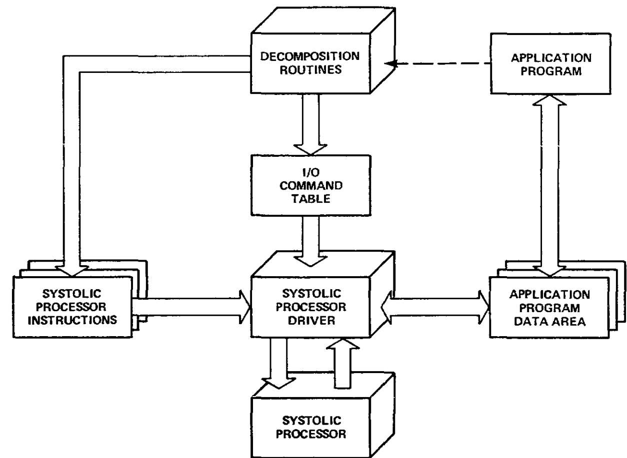


Figure 4—System software

The *decomposition routine* invoked by one of these calls divides the problem into subproblems manageable by the systolic processor and assembles a systolic processor program that processes the subproblems. The operations compiled by the decomposition routines use operands that reside in the host's primary memory and produce results that are then stored in the host's primary memory; the transfer of data and results to or from primary memory is the responsibility of the user's application program. The decomposition routines construct two data structures: a list of *systolic processor instructions* and an *I/O command table.*

The *systolic processor driver* transmits instruction lists and data arrays to the systolic processor, receives results and status information from the systolic processor, and places the results and status in the application program's data area. It interprets the commands in the I/O command table that describe the sequential DMA transfers needed. When all the tasks in the I/O command table are completed, control is returned to the application program.

System Performance

The estimated performance of a 20-cell systolic processor on several types and sizes of problems is shown in Table III. The calculations are based on the use of an interface capable of sustaining a 4-Mbyte/second transfer rate during block transfers. The execution times shown in Table III include the time to load the input data and kernel into the systolic processor and the time to place the results back into the host's memory. The effective computation rate is calculated as the total number of multiplications and additions performed for the task divided by the total execution time. A 20-cell systolic processor is capable of a peak computation rate of 200 MOPS.

Four types of functions are tabulated in Table III: 1-D convolution on a 4096-element signal, 2-D convolution on a 128×128 image, and a radix-8 DFT on a 64×64 image and a 512×512 image.

TABLE III—System performance

Function	Input data size	Kernel size	Host memory to host memory time (milliseconds)	Effective computation rate (MOPS)
		128 (16-bit)	8.6	119
1-D convolution	4096	256 (16-bit)	13.1	150
48-bit results	(16-bit)	512 (16-bit)	22.9	161
		1024 (16-bit)	38.7	163
		5 × 5 (8-bit)	10.6	72
2-D convolution	128 × 128	9 × 9 (8-bit)	20.6	113
8-bit results	(8-bit)	15 × 15 (8-bit)	53.4	110
		19 × 19 (8-bit)	66.5	131
		31 × 31 (8-bit)	155.0	113
Radix-8 DFT	64 × 64	8 × 8 (16-bit	13.0	81
16-bit complex		complex		
results	512 × 512	matrices)	1510.0	67
Radix-8 DFT	64 × 64	8 × 8 (16-bit	15.0	70
32-bit complex		complex		
results	512 × 512	matrices)	1640.0	61

The table illustrates the high effective computation rate achievable by coupling the systolic processor to a minicomputer through a standard interface. The processor is used more than 80% of the time for large 1-D convolutions. The use drops off for problems with small-sized kernels as the nonoverlapped I/O time becomes significant. Performance on 2-D convolutions exceeds 100 MOPS for a large range of kernel sizes (from 9 × 9 to 31 × 31). For the case of a 5 × 5 kernel, 20 cells are used to produce 4 output scan lines at a time. For the case of a 9 × 9 kernel, 18 cells are used to produce 2 scan lines at a time.

CURRENT SIGNAL PROCESSING APPLICATIONS

Although the basic operations of the systolic processor are matrix multiply and convolution, many applications can be treated as variations on these themes. In a standard 1-D convolution, all the cell memories are loaded with copies of the same kernel, appropriately shifted. If different weights are stored in the cell memories, variations on convolution can be performed. Appropriate choices of weights can be used to perform interpolation and decimation. In fact, the same technique is used to do both 2-D convolution and decimation. The only requirement for interpolation is that the results do not overwrite each other in the systolic output data stream. This will not happen if the number of cells employed is less than the length of the kernel. The decomposition routines guarantee that this will not happen.

A Fourier transform can be expressed as a vector-matrix multiplication where the signal samples are multiplied by the powers of the basic frequency. This approach performs many redundant computations but can be performed quickly if the number of cells in the systolic array is as large as the size of the input vector. An alternate approach is a fast Fourier transform (FFT), which minimizes the number of multiplications but cannot be performed efficiently by a systolic processor because of the complicated data flow. A compromise is a hight-radix version of the FFT algorithm, which uses the FFT approach to decompose the problem into a series of matrix multiply problems that are matched to the size of the array.[8] The performance of the systolic processor using this approach to perform a DFT is shown in Table III.

FUTURE SIGNAL PROCESSING APPLICATIONS

As experimentation has proceeded with the systolic processor, several modifications of the system have been suggested that would increase the range of algorithms it could perform. This section discusses some of these modifications and the algorithms that they would permit. The final portion describes a major new effort by ESL to design systolic processors that can do many basic linear algebra operations and discusses how these linear algebra processors can be used in signal processing.

Adaptive Filtering

The real power of the linear systolic array described here depends on two factors: the cell memories that are unique to each cell and the systolic address path that is used to address them. For the three basic operations the address stream is purely a function of time; it does not depend on the input stream. An alternative approach is to allow the address stream to vary with input data. The cell memories then perform a table lookup function. Many nonlinear functions of the input stream can be computed in this fashion, including sum of squares (or higher powers), computation of entropy, sum of magnitudes, or autocorrelation.[9]

This approach can be used to solve geometric warp problems, including 2-D interpolation.[6]

Systolic Arrays for Linear Algebra

Many of the fundamental computational problems of digital signal and image processing can be stated as classical prob-

lems in numerical linear algebra.[10] The optimum choice of weights for the control of an adaptive phased-array sensor system—an important problem in radar, communication, and radio astronomy—is an example of the linear least-squares problem of minimizing

$$\left\| Ax - b \right\|^2 = \sum_{i=1}^{m} \left| b_i - \sum_{j=1}^{n} a_{i,j} \cdot x_j \right|^2$$

Identification of multiple emitters by the MUSIC algorithm[11] requires solution of the generalized eigenvalue problem

$$Ax = aBx$$

for symmetric positive definite A and B. Least-squares estimation is used in solving the singular value decomposition problem used as part of a data compression technique.[12]

These problems have all been largely solved, from the mathematical standpoint. The algorithms for their solution are not real-time on a conventional serial computer: they take $O(N^3)$ operations on $N \times N$ matrices. Often, suboptimal solutions generated by interactive, fast, approximate methods have been used,[13] but these techniques can fail to deliver high-quality results. Hence there is strong interest in special-purpose parallel hardware for the real-time solution of these problems. The basis of this work is the discovery[14] that a family of systolic array architectures, using a simple lattice of processing elements and many identical cells, can efficiently carry out the matrix factorizations required to solve linear systems, least squares, and eigenvalue problems. With an economical VLSI implementation of these cells as building blocks, a panoply of powerful systolic processors can be easily assembled.

REFERENCES

1. Kung, H. T. "Special-Purpose Devices for Signal and Image Processing: an Opportunity in Very Large Scale Integration (VLSI)." *Proceedings of the Society of Photo-Optical Instrumentation Engineers,* 241 (1980), pp. 76–84.
2. Kung, H. T. "Why Systolic Architectures?" *Computer,* January 1982.
3. Kung, H. T., and C. E. Leiserson. "Systolic Arrays (for VLSI)." In I. S. Duff and G. W. Stewart (eds.), *Sparse Matrix Proceedings,* 1978. Philadelphia, Pennsylvania: Society for Industrial and Applied Mathematics, 1979, pp. 256–282. (A slightly different version appears in Mead, C. A., and L. A. Conway. *Introduction to VLSI Systems.* Reading, Massachusetts: Addison-Wesley, 1980, section 8.3.)
4. Kung, H. T. "Notes on VLSI Computation." CREST Parallel Processing Systems Course, Loughborough, England, 1980.
5. Kulkarni, A., and D. Yen. "The ESL Systolic Processor for Signal and Image Processing." *Proceedings of Workshop on Computer Architectures for Pattern Analysis and Image Database Management,* 1981.
6. Kung, H. T., and R. L. Picard. "Hardware Pipelines for Multi-Dimensional Convolution and Resampling." *Proceedings of Workshop on Computer Architectures for Pattern Analysis and Image Database Management,* 1981.
7. Blackmer, J., G. Frank, and P. Kuekes. "A 200 MOPS Systolic Processor." *Proceedings of the Society of Photo-Optical Instrumentation Engineers,* 298 (1981).
8. Brigham, E. *The Fast Fourier Transform.* Englewood Cliffs, New Jersey: Prentice-Hall, 1974.
9. Kulkarni, A. "Everything You Thought A MAC-Based Linear Systolic Array Could Do But Were Skeptical About!" Technical Memo, ESL APTL, 1981.
10. Speiser, J. M., and H. J. Whitehouse. "Architectures for Real-Time Real-Time Matrix Operations." Government Microcircuits Applications Conference, Houston, 1980.
11. Schmidt, R. "Multiple Emitter Loaction and Signal Parameter Estimation." *Proceedings of the RADC Spectrum Estimation Workshop,* Rome Air Development Center, Griffiss Air Force Base, N.Y., October 1979.
12. Andrews, H. C., and C. L. Patterson. "Singular Value Decomposition (SVD) Image Coding." *IEEE Transaction Communications,* 24:4 (1976), pp. 425–432.
13. Widrow, B. "Adaptive Filters." In Kalman, R. E., and N. DeClaris (eds.), *Aspects of Network and Control Theory.* New York: Holt, Rinehart, and Winston, 1970, pp. 563–587.
14. Gentleman, M., and H. T. Kung. "Matrix Triangularization by Systolic Arrays." *Proceedings of the Society of Photo-Optical Instrumentation Engineers,* 298 (1981).

Parallel-processing a large scientific problem

by ROBERT HIROMOTO
Los Alamos National Laboratory
Los Alamos, New Mexico

ABSTRACT

We discuss a parallel-processing experiment that uses a particle-in-cell (PIC) code to study the feasibility of doing large-scale scientific calculations on multiple-processor architectures. A multithread version of this Los Alamos PIC code was successfully implemented and timed on a UNIVAC System 1100/80 computer. Use of a single copy of the instruction stream, and common memory to hold data, eliminated data transmission between processors. The multiple-processing algorithm exploits the PIC code's high degree of large, independent tasks, as well as the configuration of the UNIVAC System 1100/80. Timing results for the multithread version of the PIC code using one, two, three, and four identical processors are given and are shown to have promising speedup times when compared to the overall run times measured for a single-thread version of the PIC code.

INTRODUCTION

Anticipating a need for increased computational speed[1] for laboratory codes (which is unlikely to be attained by single-processor systems), we have initiated studies to test the feasibility of doing parallel processing on multiple-processor architectures.[2] In part, our hope is to learn about multiple-processor architectures, the compatibility of algorithms with particular parallel-processing environments, parallel-processing speedups as a function of the number of processors, and the desirable characteristics of multiple-processor architectures in general.

This paper presents the results of our investigation concerning the feasibility of parallel processing a specified scientific problem on a commercially available multiple-processor system and particularly the computational speedups as a function of the number of processors employed. The problem used in this experiment involves a particle-in-cell (PIC) method for simulating the electrostatic interactions of a collisionless plasma. We first outline the PIC algorithm and graphically describe its parallel-processing structure as implemented in our experiment. A general description of the UNIVAC System 1100/80 is then given, followed by a discussion of the implementation of the PIC code on that system. The results of our experiment are given, showing overall computational speedups as a function of the number of processors and the equivalent number of parallel activities.

PARTICLE-IN-CELL

The problem selected for our parallel-processing experiment models the collisionless, electrostatic interaction between two superimposed plasma beams with a relative drift velocity.[3] The code uses a particle-in-cell method for studying the interaction and resulting motion of the charged particles in this simulation.[4] This code is of general interest to us because it represents a class of algorithms exhibiting limited vector capabilities for implementation on our vector computers. Due to the PIC algorithm (discussed in the following paragraph), the conversion to parallel processing was made with relative ease.

PIC Algorithm

The particle-in-cell method used in this study decomposes a region of space into a collection of cells. These cells are then used for tracking particle movement, and they assist in evaluating relevant physical properties. An initialization stage sets up two ensembles of charged particles (we shall use particles to mean charged particles throughout this paper) constituting the two superimposed, collisionless plasma beams. During this initialization, the particles are distributed uniformly in space and randomly in velocity. The movement of particles is discretized in a time step (dt).

During each computational time step of the simulation (see Figure 1), cell-centered charges (C) are calculated by linearly weighting each particle's charge contribution to the four nearest-neighbor cell centers. Using this charge distribution, Poisson's equation with periodic boundary conditions is solved for the associated electrostatic potential (ϕ) on the grid of cell centers, with the resulting electric (E) field interpolated to individual particle positions. Under this E field, each particle's position and velocity (see Figure 2) are advanced (pushed).

PIC-Parallel-processing Structure

The computational structure of the PIC algorithm, as implemented on the UNIVAC System 1100/80, takes advantage of the large, natural computational divisions of the particle initialization and aspects of the particle-in-cell calculations. Figure 3 graphically displays the multi/single-thread diagram of our PIC code, with accompanying definitions of the respective calculation(s) each thread performs.

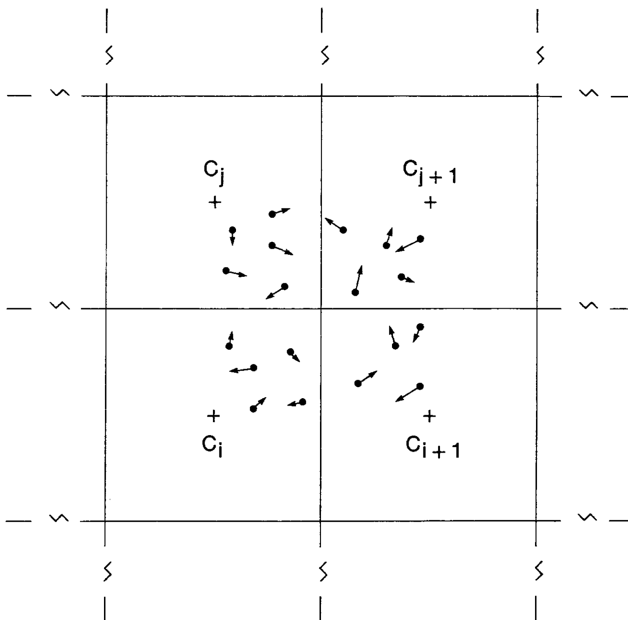


Figure 1—A distribution of particles (dots with attached arrows—denoting position and velocity, respectively) contained within the four nearest-neighbor, cell-centers (+) from which the charges C_i, C_{i+1}, C_j, and C_{j+1} are in part calculated.

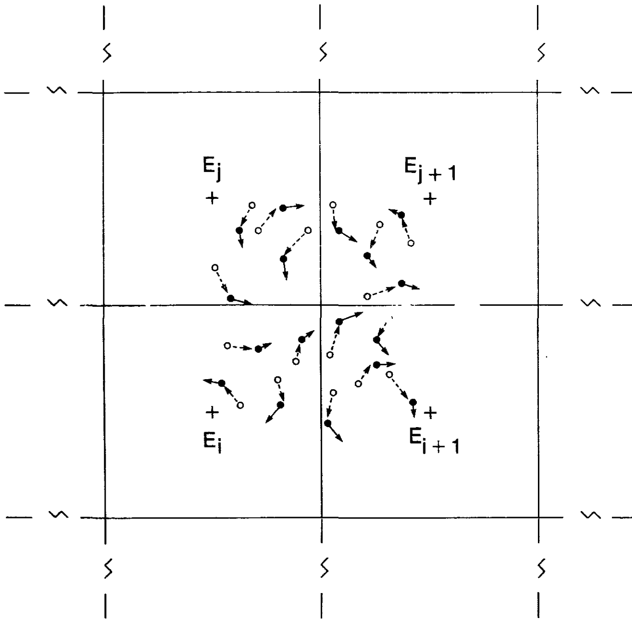


Figure 2—Distribution of particles pushed under the influence of the four nearest-neighbor, cell-centered (+) electric fields E_i, E_{i+1}, E_j, and E_{j+1} (determined by solving Poisson's equation ($-\nabla^2\phi = C$) for ϕ) and the uniform background electric/magnetic fields.

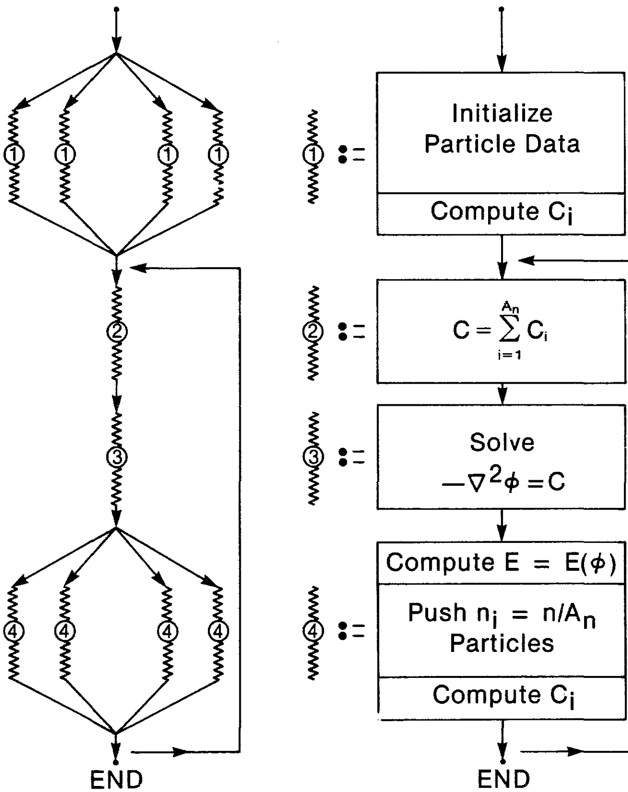


Figure 3—A multithread version of PIC as implemented on a UNIVAC System 1100/80 with two parallel-processing tasks (1 and 4), where A_n = total number of parallel activities (multithread), n = total number of particles, n_i = number of particles for activity i, C = total charge (distribution), and C_i = charge computed for activity i.

IMPLEMENTATION ON A UNIVAC SYSTEM 1100/80

Our parallel-processing version of the PIC code was implemented on a UNIVAC 1100/80 multiple-processor system.* The System 1100/80 may be configured with from one to four processors. UNIVAC's designation for its System 1100/80 with a one-, two-, three-, or four-processor configuration is denoted by 1100/81, 1100/82, 1100/83, or 1100/84, respectively.

A global software manager (EXEC) executes out of all processors and, coupled with hardware devices, drives the multiple-processor architecture of the System 1100/80. The aggregate of processors share a common memory, which allows for multiple-program execution for tasks written in FORTRAN or COBOL. A principal feature of the System 1100/80 is its ability to parallel-process a single instruction stream upon data in common memory. This capability, supported by the COBOL compiler but not by the FORTRAN compiler, was essential for our particular experiment.

Implementation

The PIC code was written entirely in FORTRAN and implemented with a single copy of the instruction stream. The management of data addressing and the mechanics of parallel-processing synchronization were devised and implemented by Dave Hammer of Sandia National Laboratories, Albuquerque, NM.†

Figure 4 represents a simplified diagram of a UNIVAC 1100/84 (four-processor) system, on which our PIC timing

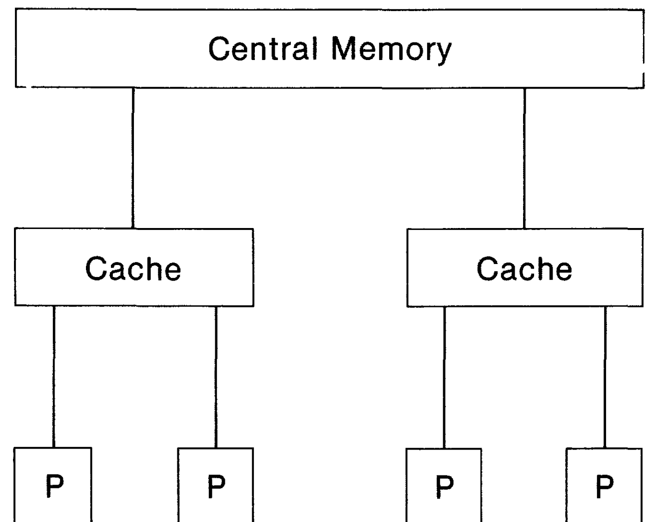


Figure 4—A simplified diagram of the UNIVAC System 1100/80 with four processors (P), designated 1100/84.

*Provided for our use by the Computer Operations Department, Sandia National Laboratories, Albuquerque, New Mexico.

†By devising an address mapping and a synchronization scheme for multithread activities, Hammer essentially converted the System 1100/80 into a FORTRAN parallel-processing machine for our use.

runs were made. Although not indicated in the diagram, the processing of each activity is not necessarily handled by only one physical processor. In fact during the complete computational cycle of such an activity, all processors may timeshare the execution of the activity. A distinction, therefore, is made between activities and processors.

All relevant particle-in-cell data were put into various common blocks and partitioned for use by specific activities. Due to software addressing limitations, the PIC code was restricted to a maximum of 262k (decimal) words of total memory. For each particle, five data quantities (two for position and three for velocity) were required. Three mesh quantities, constituting a 34×34 mesh size, were required and duplicated for a maximum of eight (particle-push) activities. A total of 37k particles were initiated for processing, requiring 213k words of memory (particle plus mesh data). A further 47k of memory was used for the instruction stream, address mapping, and activity synchronization scheme.

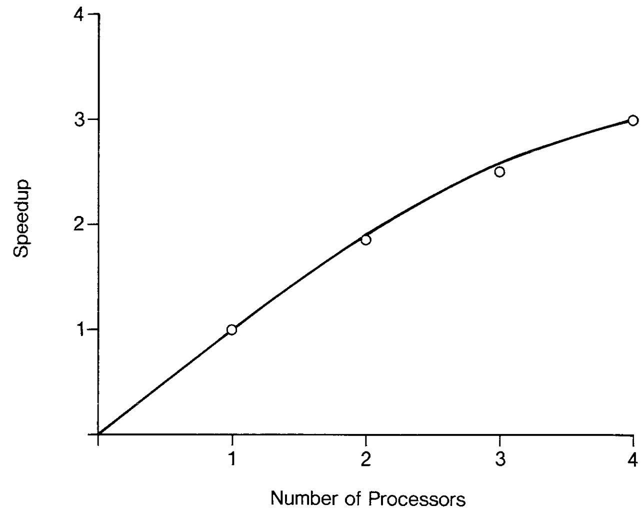


Figure 5—Plot of number of processors versus speedup corresponding to Table I.

PIC PARALLEL-PROCESSING RESULTS

A multithread version of the PIC code was executed on a UNIVAC System 1100/80‡ with one, two, three, and four identical processors. Overall run times were measured, and the results are given in Table I and Figure 5. The speedup values are the ratios of the overall execution time of a single-thread version of PIC (running on one processor) to the overall execution time of a multithread PIC code running on two, three, and four processors. We found that a maximum speedup of three was attained when using four processors with four activities spawned for each task.

Because the multithread PIC was not totally parallel (see Figure 3), the speedup for four processors may not indicate the full potential of the PIC algorithm. The times recorded and used for the parallel-processing speedup calculations were based on wall clock times, with timing runs made in a dedicated mode. Due to resource and time limitations, actual CPU times were not measured; therefore, no estimates could be determined for losses in effective processing time during the synchronization stage of each multithread activity.

TABLE I—Run times and speedups as a function of number of processors and number of activities for each parallel task spawned.

Number of Activities Per Parallel Task	Number of Processors	Average Run Time (millisecond)	Speedup
1	1	102631	1
2	2	57110	1.80
3	3	42214	2.43
4	4	33263	3.09

‡Provided for our use through the courtesy of SPERRY UNIVAC, Roseville, Minnesota.

CONCLUSIONS

Our results strongly suggest the possibility of significant computational speedups for a multiple-processor architecture similar to the UNIVAC System 1100/80. The coupling between algorithm and processing architecture illustrates not only the seemingly high degree of compatibility between our particular code and the computing environment, but also the need to distinguish those algorithms for which specific multiple-processor architectures are most effective.

The straightforward use of FORTRAN in coding the multithread PIC algorithm greatly simplified the overall task of implementing our parallel-processing experiment. Programming in FORTRAN is certainly a characteristic of Laboratory codes, and would be a desirable feature to retain when converting such codes from serial- to parallel-processing systems.

Encouraged by our results, we currently are studying the possibility of a totally parallel version of the PIC algorithm. We also plan to investigate parallel processing on multiple-processor architectures possessing as many as 16 processors.

REFERENCES

1. Buzbee, B. L., W. J. Worlton, G. Michael, and G. Rodrigue. "DOE Research in Utilization of High-Performance Computers." Los Alamos Scientific Laboratory report LA-8609-MS, December 1980.
2. Bucher, I. Y., P. O. Frederickson, and J. W. Moore. "Experience with a Multiprocessor Based on Eight FPS 120B Array Processors." Los Alamos Scientific Laboratory Report LA-UR-81-1082 (unpublished), 1981.
3. Morse, R. L., and C. W. Nielson. "One-, Two-, and Three-Dimensional Numerical Simulation of Two Beam Plasmas." *Physical Review Letters,* 23 (10 November 1969), 19, pp. 1087-1090.
4. Morse, R. L., and C. W. Nielson. "Numerical Simulation of Warm Two Beam Plasma." *The Physics of Fluids,* 12 (November 1969), 11, pp. 2418-2425.

Design of software for distributed/multiprocessor systems

by TERRENCE R. MCKELVEY and DHARMA P. AGRAWAL
Wayne State University
Detroit, Michigan

ABSTRACT

Software design methodologies for distributed/multiprocessor systems are investigated. Parallelism and multitasking are considered as key issues in the design process. Petri-nets and precedence graphs are presented as techniques for the modeling of a problem for implementation on a computer system. Techniques using the Petri-net and precedence graph to decompose the problem model into subsets that may be executed on a distributed/multiprocessor system are presented. These techniques offer a systematic design methodology for the design of distributed/multiprocessor system software.

INTRODUCTION

Since the advent of the digital computer, the need for faster, larger, more reliable, and expandable systems has existed. Distributed/multiprocessor computer systems have resulted from this need. Though hardware has been developed to allow the exploitation of concurrent computation, the application of this hardware to real-world problems, and the development of software to solve these problems is developing at a slower pace. This paper presents some of the current thinking on the subject of software design methodologies for distributed/ multiprocessor systems. The following sections discuss some basic concepts and definitions, parallelism at the various levels of a computer system, multitasking as a design approach for multiprocessor systems, graphical techniques for the representation of an application, and finally some techniques for using these graphical representations to decompose the application into sections that can be run concurrently on a multiprocessor system.

For the purposes of this paper, the concept of a distributed/ multiprocessor system may be defined as follows: "Distributed computing refers to the use of multiple, quasi-independent processing modules, whose actions are coordinated to accomplish a large task or to implement a large system."[1] Though the system need not be large, the need for multiple, fairly independent processing modules tied together into a system is key to the concept of distributed/multiprocessor computation. Another key point is this one: "In a distributed computing system, the fact that multiple processing modules exist is visible to the user of the system, and therefore meant to be exploited in the design of applications."[1] Though much work is being done to develop tools that hide this visibility from the user, optimal use of distributed/multiprocessor systems will result when the designer detects and exploits parallelism existing in the application, as examined in the rest of this paper.

PARALLELISM IN COMPUTER SYSTEMS

Parallelism may be introduced into a system at various levels. Computation that can be done in parallel may be done within separate processor modules, thus obtaining the speed and reliability advantages offered by distributed systems. The design of the software will ultimately determine the success or failure of a computer system to solve a given real-world problem. "Matching a program representation to the underlying hardware or interpretive resources in a computer system is a key problem in computer system design. Failure to accurately and completely represent the computation significantly degrades the performance of the resulting execution."[2] The translation of a problem into a computer implementation may be represented as a hierarchical structure:

1. An algorithm (or solution of the problem) specifies
2. A set of tasks (functions to be performed), composed of
3. Higher-level-language statements, represented by
4. Computer instructions, which cause
5. State transitions in the computer hardware.

This hierarchy forms a pyramid, where each level of the hierarchy is composed of a number of elements of the next level. (An algorithm is composed of tasks, for example.) Parallelism may be detected at any of these levels. Of these levels, the following three are the important levels at which parallelism may be detected:

1. Algorithm level
2. Source language level (i.e., the higher language level)
3. Machine language level (i.e., instruction and state transitions)

Detection and exploitation of parallelism is the key to the effective design of distributed/multiprocessor systems. Studies in the dynamic detection of parallelism at the machine language level have been performed.[3] These studies have led to the conclusion that an overall net parallelism detection of less than a fivefold factor over the strictly sequential execution of machine language instructions is theoretically possible. This would probably lead at the practical level to a twofold improvement over the sequential approach. This level manifests itself in the detection and parallel execution of independent machine language instructions. Pipelining at the instruction level is a similar technique to exploit parallelism at the machine instruction level.

Detection and exploitation of parallelism at the source language level is currently an area of active research.[2,4] Program analyzers have been written to detect inherent parallelism in programs written in higher-level languages.[4] It has been empirically observed via such an analyzer that a speedup, SP, on P processors would be possible:

$$SP = P / 10 \log_{10}P$$

This result is based on the analysis of FORTRAN programs and may be found to be better or worse for other higher-level languages or implementations of FORTRAN on other computers.

The detection of parallelism at the algorithm level is, of course, very dependent upon the problem to be solved. Various approaches to the parallel execution of sorting and searching algorithms appear in the literature. However, very little

has been written concerning a design methodology to detect and exploit parallelism at the algorithm level.

Very few real-world applications are as specific as searching and sorting algorithms. Many real-world applications consist of several algorithms, and one would most probably consider the algorithm level of the previous discussion to be the system design level as practiced by the computing community. This level may produce a system made up of hundreds of individual programs to perform the intended function. The process of breaking up the system into a number of tasks (to be defined shortly) and determining which of these tasks may be executed in parallel offers a mechanism to detect and exploit parallelism at the algorithm level. This method, known as multitasking, has been used on uniprocessor systems for years. The subject of multitasking will be discussed in the following section. As an aside, however, it is worth noting that the methodologies presented offer no panacea for the design of distributed systems. The designer must intelligently, and often iteratively, apply these techniques in order to find the optimal design for his or her application.

MULTITASKING AS A DESIGN APPROACH FOR DISTRIBUTED SYSTEMS

As discussed in the previous section, multitasking offers a design methodology for the detection and exploitation of parallelism for distributed/multiprocessor systems. This section will define tasks and multitasking and present some examples of the multitasking approach to system design.

A task may be defined as a unit of computational activity.[5] When the computer first became available, a programmer would code an application as one large program or algorithm. The computer would load and execute this and any other tasks, one at a time, from start to completion of the algorithm. As time went on, it was found that the central processing unit (CPU) would be idle during certain activities—I/O, for example. During this time, it was proposed, another task could be executed until CPU had to wait. Thus was born the concept of multitasking, the capability of executing more than one task concurrently. This capability, extended to multiple-processor computer systems, is known as multiprocessing. The capability of multitasking is also known as multiprogramming.

The concept of multitasking, as stated above, was initially conceived to take advantage of expensive CPU idle time. However, it is a valid method for the design of software systems. There is no reason why a problem must be programmed and executed one step at a time from start to finish. In fact, for many problems, this approach becomes extremely awkward, especially in many real-time applications where data must be collected, displayed, and analyzed concurrently.

Designers often shy away from the multitasking approach, since people generally tend to think sequentially. However, the multitasking approach offers advantages in terms of efficiency of resource use; improvement in the overall speed of execution; and, most important; a natural design methodology. An example will make the above statement clearer as well as allow a comparison of the sequential and multitasking approaches to design.

Suppose a designer is asked to design a system that will read a record of data from a data collection subsystem, display the data on a CRT, and save the data on a disk. Assume the following additional requirements:

1. Data records must be averaged across ten readings.
2. The display must be updated at least once every five seconds.

A sequential design, presented in a pseudo-high-level language, might be that shown in Figure 1. Though the solution is relatively straightforward, it could easily not fulfill a requirement of the system: it may very well take longer than five seconds to reach the statement to display the results on the CRT. In fact, one may find that records from the data collector might be missed if it takes too long to write the record to disk or to update the CRT display. Admittedly, this example ignores the fact that direct memory access and interrupt processing capabilities could solve some of these problems, but the main point remains that the sequential approach may take too long and could potentially fail to do the required job. The multitasking approach asks the question, "Is there any way to break up this system into a number of cooperating tasks which could run concurrently?" The answer is "yes," and a solution is shown in Figure 2. The above solution assumes the existence of three record buffers for use by the three tasks. These three tasks can operate in a pipelined manner: that is, Task 1 collects a record bufferful and passes it to Task 2, which averages and saves the buffer on disk while Task 1 is collecting the next bufferful. Task 3 gets the results of Task 2; and while Task 3 displays the results, Task 2 processes the second bufferful while Task 1 is collecting the next bufferful. Once the pipeline is running, all three major functions (collection, averaging and disk storage, and display) are proceeding concurrently. On a uniprocessor any one task could run while the others are in wait states. To implement this any-one-of-three

```
DO FOREVER:
        CLEAR RECORD_BUFFER
        COUNT = 1
        DO WHILE COUNT < = 10:
                WAIT FOR RECORD
                RECORD_BUFFER(COUNT) = RECORD
                COUNT = COUNT + 1
        END
        COUNT = 1
        CLEAR RESULT_BUFFER
        DO WHILE COUNT < = 10:
        RESULT_BUFFER = RESULT_BUFFER +
                          RECORD_BUFFER(COUNT)
                COUNT = COUNT + 1
        END
        RESULT_BUFFER = RESULT_BUFFER / 10.
        WRITE RESULT_BUFFER TO DISK
        DISPLAY RESULT_BUFFER ON CRT
END
```

Figure 1—Sequential design presented in pseudo high-level language

```
TASK 1

    DO FOREVER
            WAIT FOR FREE RECORD_BUFFER
            CLEAR RECORD_BUFFER
            COUNT = 1
            DO WHILE COUNT < = 10
                    WAIT FOR RECORD
                    RECORD_BUFFER(COUNT) = RECORD
                    COUNT = COUNT + 1
            END
            SET RECORD_BUFFER = FULL
    END

TASK 2

    DO FOREVER
            WAIT FOR FULL RECORD_BUFFER
            COUNT = 1
            DO WHILE COUNT <  10
                    RECORD_BUFFER(1) =
                      RECORD_BUFFER(COUNT+1)
                    COUNT = COUNT + 1
            END
            SAVE RECORD_BUFFER(1) ON DISK
            SET RECORD_BUFFER = STORED
    END
TASK 3

    DO FOREVER
            WAIT FOR A STORED RECORD_BUFFER
            DISPLAY RECORD_BUFFER ON CRT
            SET RECORD_BUFFER = FREE
    END
```

Figure 2—A multitasking approach

actions approach in a single task would result in a much more complex solution than that originally presented and would definitely be more complex than the three-task approach. Additionally, the three-task approach could be effectively run on a multiprocessor or distributed system. To summarize, the above example has shown how a multitasking approach can yield a simpler design which can be run on a distributed system.

A group of such tasks, which work in concert to perform some application, is known as a task system. Some tasks within a task system must be executed in sequence, but many parts may not require this restriction. This definition of required sequentiality among tasks is known as a precedence relation. This precedence relation, often represented graphically as a precedence graph, will be discussed in a succeeding section.

An important concept relative to task systems is that of determinacy. A task system is determinant if and only if it always produces the same results, given the same inputs. For a task system to be determinant, the tasks making up the task system cannot interfere with each other. Given the premise that a task requires some set of inputs, called its domain, *D*, it produces a set of outputs, called its range, *R*. Also given a task system *C*, made up of tasks $T1, T2, \ldots, Tn$, Tasks *T* and T' of *C* are noninterfering if either of the following conditions is true:

1. *T* is a successor or predecessor of T'. That is, *T* runs to completion before (predecessor) or after (successor) T' runs to completion. In other words, *T* and T' run in a strictly sequential relationship to one another.
2. The intersection of the following sets is the null set:
   a. Ranges of *T* and T'
   b. Range of *T* and domain of T'
   c. Domain of *T* and range of T'

That is, $R_T \cap R_{T'} = R_T \cap D_{T'} = D_T \cap R_{T'} = \phi$

For $C = \{T1, T2, \ldots, Tn\}$, Ti and Tj are mutually noninterfering if Ti and Tj are noninterfering for all i,j where i not $= j$. Task systems made up of mutually noninterfering tasks are determinate.[5]

What the above discussion indicates is that one establishes a precedence relationship to assure the determinacy of a task system, thus assuring consistent results when executing the task system. A method will be presented under the discussion of precedence graphs to find the set of tasks that can be executed in parallel, given a determinant task system.

A key question one might ask is, "How does one go about detecting possible tasks?" The answer lies in the concept of stepwise decomposition of the application into major functions to be performed and the major functions into subfunctions until one reaches a level of detail sufficient for understanding how the application will be implemented. This list of functions and subfunctions defines a potential list of tasks and steps within tasks. Using the previously presented example, the application was a monitoring system. This system was decomposed into three major functions: read input, average and store on disk, and display on a CRT. For this particular system, this level of decomposition defines the application enough to make software implementation possible. Of course, on a larger system, more functions and even subfunctions could be defined. The relationships between tasks are established, and it becomes possible to model the system by means of one of a number of graphical techniques.

Various existing graphical techniques may be used to design the algorithms and tasks required for a specific application. Methods exist for partitioning these graphs into segments that may be executed in parallel. These techniques will now be discussed.

GRAPHICAL TECHNIQUES FOR THE REPRESENTATION OF SYSTEMS

As stated previously, this section will discuss graphical techniques for the representation of application systems. These techniques may be used to partition an application into tasks and tasks into programs. Additionally, techniques exist for the partitioning of these graphs into segments that may be

executed in parallel. Two graphical techniques will be presented:

1. Petri nets
2. Precedence graphs

These are commonly used techniques in the computer engineering and computer science disciplines, respectively.

PETRI NETS

The Petri net has been discussed extensively in current literature;[6, 7, 8] therefore only a brief introduction will be presented here. The major emphasis will be upon partitioning techniques. A Petri net is a graph model for "modelling the flow of information and control in systems, especially those which exhibit asynchronous and concurrent properties."[6] Two types of nodes exist in Petri nets: circles (called places) that represent conditions and bars (called transitions) that represent events. Black dots (called tokens) appear in circles to represent the holding of a condition at that place. The distribution of tokens throughout the graph represents the state of the system. The behavior of the system can be determined by tracing the flow of tokens through the system. Tokens move from one place to another when a transition fires. The following rules define the conditions under which transition firing may occur:

1. A transition is enabled if and only if each of its input places has at least one token.
2. A transition can fire only if it is enabled.
3. When a transition fires:
   a. A token is removed from each of its input places.
   b. A token is deposited into each of its output places.

Figure 3[6] depicts a simple computer system modeled via a Petri net. The concurrency between I/O on tape and the execution of computing on the CPU is apparent: when the tape is ready, the processor is ready, and an input queue entry exists, the transition fires, I/O on tape and computing run concurrently, and the results go to the output queue after the appropriate I/O and computing have completed. One could "walk through" the system in Figure 3 by moving the tokens from place to place.

As shown in Figure 3, concurrent operations fork at the transition at the top of the figure and join at the transition at bottom of the figure. In a Petri net of more considerable size there may be many such forks and joins in the net, which represent sites of concurrent activities. By partitioning the Petri net along these forks (called distribution AND nodes) and joins (called synchronization AND nodes) the net can be broken up into subnets having an initiation point and a termination point with no forks or joins between nodes. In other words, a subnet of strictly sequential places and transitions can be produced by breaking up a Petri net at each of its fork and join nodes. However, these subnets, while representing the maximum level of parallelism, cannot all execute in parallel at the same time. This is due to the fact that the transitions firing resulting in forks happens at different times. Therefore,

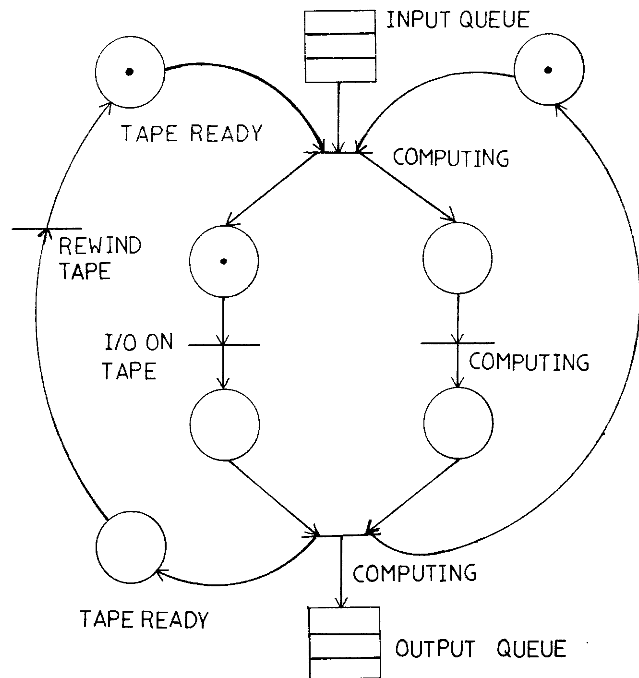


Figure 3—Petri net model of computer

although one has all the subnets that could potentially run concurrently, one does not have subnets of operations that can all run concurrently.

Since one does not have the set of all subnets that can all be executed concurrently, one would not be efficiently using processors if one loaded each subnet into a separate processor and triggered each processor at the appropriate transition point. Some processors would be executing in parallel, it is true; but many would be idle during the course of execution of each subnet. What is desired is not to split the Petri net into subnets at each fork and join point, but rather, after the first fork, where one does do a split, to follow sequential chains through each fork and join. In other words, one breaks the Petri net at the first fork and proceeds down the places and transitions until either a fork or a join is encountered. In the case of a fork, one of the possible paths is chosen and the subnet chain continues along that pathway. The other possible pathway at the fork becomes the beginning of another chain. If a join is encountered, again a subnet chain continues; however, the other arrows entering that join transition become the termination points for the other subnet chains followed up to that point along other pathways. By following various pathways through the Petri net in this way, one produces subnets of sequential places and transitions that are longer than the approach of splitting the Petri net at each fork and join. If the paths are carefully chosen, it is possible to produce subnets that can be executed concurrently, although not necessarily all at the same time. To summarize, what is desired is to produce a set of subnets from an initial Petri net so that the following conditions are met:

1. A minimal number of subnets are produced, all of which are sequential chains of places and transitions.

2. The subnets are chosen to allow maximal concurrent execution of each subnet.

Toulotte and Parsy[7] present an algorithm for this decomposition that would satisfy Condition 1. This algorithm produces a set of subnets based on the idea that the optimal set of such subnets is the set having the least number of subnets, where each subnet is a sequential chain. The algorithm may be summarized as follows:

1. Define the initial place.
2. Trace down the chain of places and transitions until a fork or join transition is encountered (called an AND node).
3. If the AND node is a fork, determine which output path will result in the smallest number of additional subnets.
4. If the AND node is a join, determine which input path, if continued, would result in the smallest number of additional subnets.

Though this algorithm would produce a minimal set of subnets (see Toulotte and Parsy[7] for more details on the algorithm itself), this minimal set may not be the optimal set for maximally concurrent execution. That is, Condition 2 is not covered by this algorithm. With some alteration, the algorithm could probably be modified to find the set of such subnets such that both Conditions 1 and 2 above would be met. This would result in an algorithm that would allow one to determine the maximum number of subnets that could concurrently execute on a set of processors, requiring one processor per subnet. This algorithm could be automated and done on a computer once a Petri net of the application was produced. Some guidelines for producing the initial Petri net will be made after a specific example of the subnet splitting technique is presented.

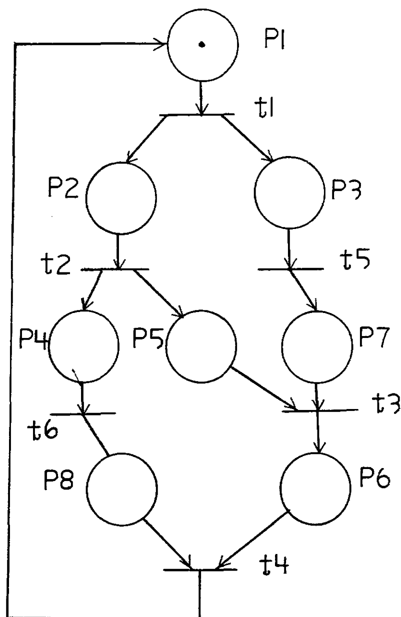


Figure 4—Petri net

Figure 4 presents a Petri net having several AND nodes. The places are labeled $P1$ through $P6$ and the transitions $t1$ through $t6$. If one were to split up this net at each fork and join, one would produce the following subnets:

$S1 = P1, t1$
$S2 = P2, t2$
$S3 = P3, t5, P7, t3$
$S4 = P4, t6, P8, t4$
$S5 = P5, t3$
$S6 = P6, t4$

The above subnets represent the maximally parallel set of subnets. However, these subnets cannot all be run concurrently. Applying the above algorithm, one produces the subnets illustrated in Figure 5. These three subnets happen also to fulfill both the conditions listed above. For this example, the optimal number of processors would be three, where the second processor begins executing at $t1$ and the third processor at $t2$, with all three running until $t3$, at which point the second processor stops and the first and third processors continue until $t4$. While satisfying Condition 2 above was fairly obvious for this example, in a larger Petri net various alternative chains might have to be tried to find the optimal set of subnets. This, like the discussion on Petri net generation that follows, may require an iterative process to obtain the optimal results.

As stated above, several guidelines may be presented on the

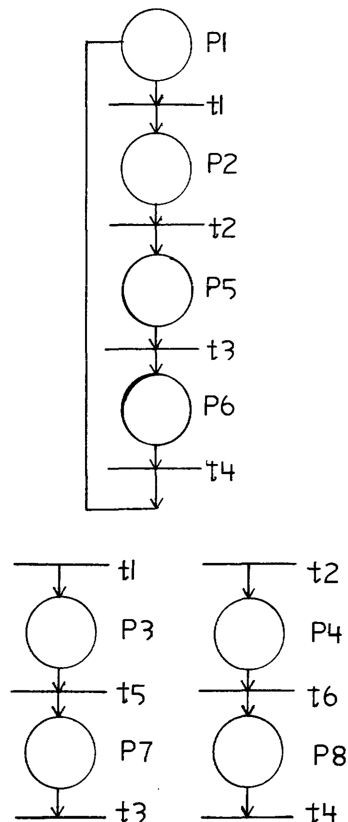


Figure 5—Decomposed Petri net

generation of the Petri net model for an application. The guidelines may be summarized as follows:

1. Break up the application into major tasks to be performed. These become the places in the Petri net.
2. Define the precedence relationships between the major tasks (i.e., which tasks depend on results from other tasks). These precedence relationships define the transitions between the tasks. Tasks that produce results needed by more than one other task are connected to those other tasks via a fork transition and have a predecessor relationship to tasks needing the results of that task. Tasks that need results produced by more than one other task are connected to those other tasks via a join transition and have a successor relationship to those other tasks.
3. Apply the splitting technique, based on the two conditions listed above.
4. Having found the major concurrent task subnets, further decompose each task, represented by a place, into subtasks; and repeat Steps 1 and 2 above to decompose task subnets into subtask subnets that can run concurrently.

In other words, the decomposition technique presented above is used on Petri nets to find the set of subnets that can be executed on a distributed/multiprocessor system.

PRECEDENCE GRAPHS

As previously stated, precedence graphs may be used to show the relationships of tasks within task systems. The method of decomposition of an application into a set of subfunctions, and subfunctions into tasks is used as the first step in the creation of a precedence graph. One then defines the precedence of the tasks based on their required order of execution to assure that a determinate task system results. Predecessor tasks trigger successor tasks, which is indicated by a directed arc in Figure 6. As stated earlier, a task takes inputs, performs some transformation function upon the inputs, and produces outputs. Predecessor tasks produce outputs, which become the inputs to successor tasks. Figure 6 presents a simple precedence graph. Task $T1$ is the initiator task to the entire task system. It is the immediate predecessor of tasks $T2$ and $T3$, which are $T1$'s immediate successors. $T3$ is the immediate predecessor of $T4$ and $T5$, and $T6$ is the terminator task for the task system. Once one has established a determinate task system, it becomes possible to apply a theorem to find the maximally parallel graph of the task system. Given a maximally parallel graph, one can visually ascertain the maximum number of tasks that may execute in parallel at any given time. The theorem given previously states the following:

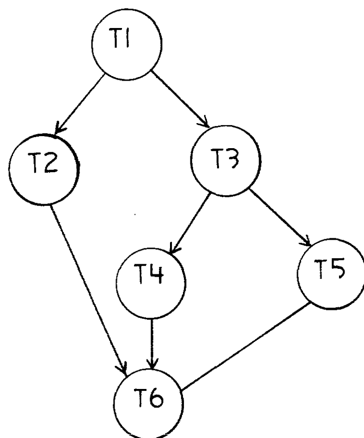


Figure 6—Precedence graph

From a given determinate task system C, construct a new system C' that is the transitive closure of the relation

$$X = (R_T \cap R_{T'}) \cup (R_T \cap D_{T'}) \cup (D_T \cap R_{T'}) = \emptyset$$

then C' is the unique maximally parallel system equivalent to C. In other words, one performs the following steps to find the maximally parallel system equivalent of a task system:

1. Calculate the relation X:
   One finds the union of the intersections of the following sets:
   a. Ranges of T and T'
   b. Range of T and domain of T'
   c. Domain of T and range of T'
2. Take the transitive closure of X by drawing the precedence graph of the relation, X, and eliminating redundant arcs.

The basic idea of this theorem is to take a determinate system and "relax" the determinacy to the point where any further "relaxation" would result in the system's becoming nondeterminate. Therefore if one starts by defining the task system as being entirely sequential—i.e., Task 1 is followed by Task 2, etc., one has defined a nonparallel, determinate task system. One then applies the procedure to find the maximally parallel task system resulting from relaxing the determinacy applied to the system by defining a strictly sequential precedence relationship among the tasks making up the original task system.

This procedure is best understood by example. Assume that a task system is given whose input and output values are represented by the set M, where

$$M = \{M1, M2, M3, M4, M5\}.$$

These five values lie in the various domains and ranges of eight tasks that make up the task system. Table I summarizes which values lie in the domains and ranges of each task.

TABLE I— Values in relation to domains and ranges of tasks

Value	In domain of tasks	In range of tasks
$M1$	1, 2, 7, 8	3
$M2$	1, 7	5
$M3$	3, 4, 8	1
$M4$	3, 4, 5, 7	2, 7
$M5$	6	4, 6, 8

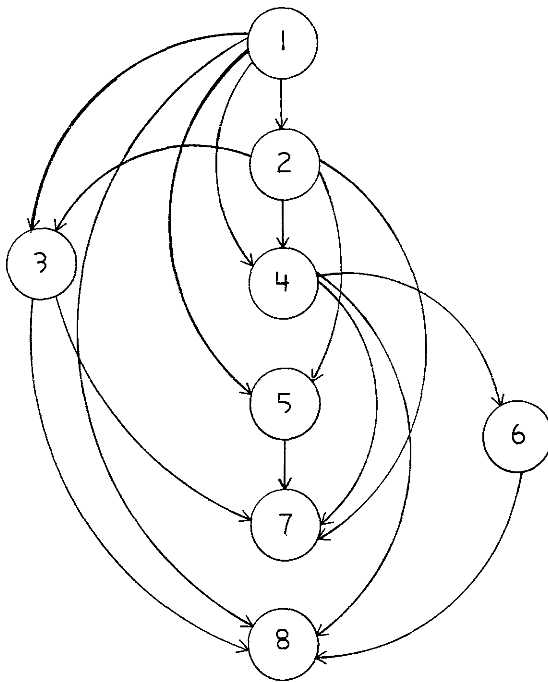


Figure 7—Graph of relation, X

The relation, X, is then calculated. As an example, $M1$ lies in the domain of Task 1 and in the range of Task 3. This defines the ordered pair (1,3). One proceeds to find all the ordered pairs resulting from comparing the domains and ranges of the tasks as defined by Relation X. This results in the set X:

(1, 3), (1, 4), (1, 5), (1, 8)
(2, 3), (2, 4), (2, 5), (2, 7)
(3, 7), (3, 8)
(4, 6), (4, 7), (4, 8)
(5, 7)
(6, 8)

One then draws a precedence graph, G, of the relation, X, as shown in Figure 7. The transitive closure of X can be found by by eliminating all redundant arcs in G. For example, Task 1 has an arc to Task 3 and to Task 8. Task 3 has an arc to Task 8. Therefore, the arc from Task 1 to Task 8 is redundant and can be eliminated. Having done this for all redundant arcs, and redrawing G to produce G', one has the maximally parallel graph of the task system originally defined to be the strictly sequential task system executing from Task 1 through Task 8. This maximally parallel graph is shown in Figure 8. From Figure 8 one can see that Tasks 1 and 2 can run concurrently and that when they are done, Tasks 3, 4, and 5 can then run concurrently. When Task 4 is completed, Task 6 may start; when Tasks 4 and 5 are completed, Task 7 may run. Finally, Task 8 runs when both Tasks 3 and 6 are completed. Therefore, one could use three processors effectively to implement the example task system on a distributed/parallel system.

COMPARISON OF TECHNIQUES

As a final example, to illustrate the previously presented techniques as applied to a real problem, an automotive trip computer/speed control is to be designed. It will monitor mileage, fuel use, and time and maintain speed. The required system can be broken down into 10 tasks that perform the following functions:

Task	Function
1	Read fuel flow, odometer, time, desired speed
2	Calculate delta time: current time minus old time
3	Calculate miles per gallon: $\text{mpg} = \dfrac{\text{(new odometer minus old odometer)}}{\text{(new flow + old flow) / 2}}$
4	Calculate speed: $\text{speed} = \dfrac{\text{(new odometer minus old odometer)}}{\text{delta time}}$
5	Calculate throttle value: If speed $<$ desired speed minus 2, then increment throttle; otherwise, If speed $>$ desired speed plus 2, then decrement throttle
6	Output throttle value to throttle control
7	Calculate fuel left: fuel left = fuel left − (new flow + old flow). delta time/2
8	Read selected function button
9	Update time, flow, odometer: old time = new time old flow = new flow old odometer = new odometer
10	Display selected function: odometer, miles per gallon, time, or fuel left

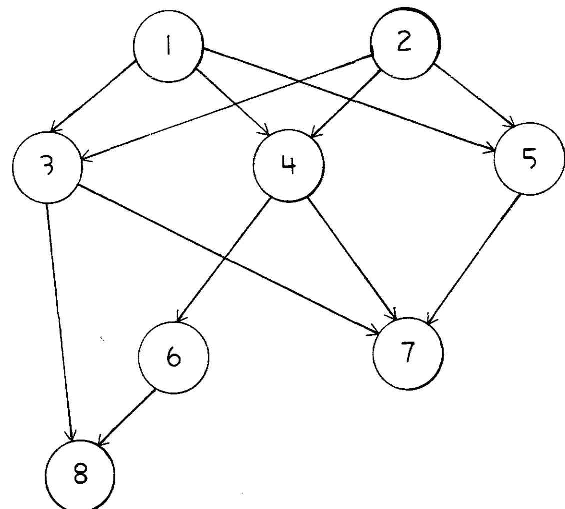


Figure 8—Maximally parallel graph

TABLE II—Task list variables in relation to domains and ranges

Variable	Domain of task	Range of task
New odometer	3, 4, 9, 10	1
New time	2, 10	1
Desired speed	5	1
Old time	2	9
Delta time	4, 7	2
Mpg	10	3
Old odometer	3, 4	9
New flow	3, 7, 9	1
Old flow	3, 7	9
Speed	4, 5	4
Throttle	5, 6	5
Throttle control	—	6
Fuel left	7, 10	7
Function index	10	8
New time	9	1
Display output	—	10
Read input	1	

From this task list a table of variables (Table II) is created that specifies the variables themselves and whether they lie in the domain and range for each of the tasks. From this table it becomes possible to construct either a Petri net or a maximally parallel precedence graph. If the latter approach is taken, determinacy is established by requiring the strictly sequential execution of Tasks 1 through 10 in that order. Figure 9 presents the Petri net constructed from the above table, and the subnets broken out from it. As can be seen, four subnets are possible, the fourth being one task. This implies that the optimal number of concurrently running processors is three. Figure 10 presents the calculation of the X relation and the resulting precedence graph. Figure 11 presents the result of taking the transitive closure of the graph in Figure 10—i. e., elimination of all redundant arcs—to produce the maximally parallel precedence graph for the above task system. Depend-

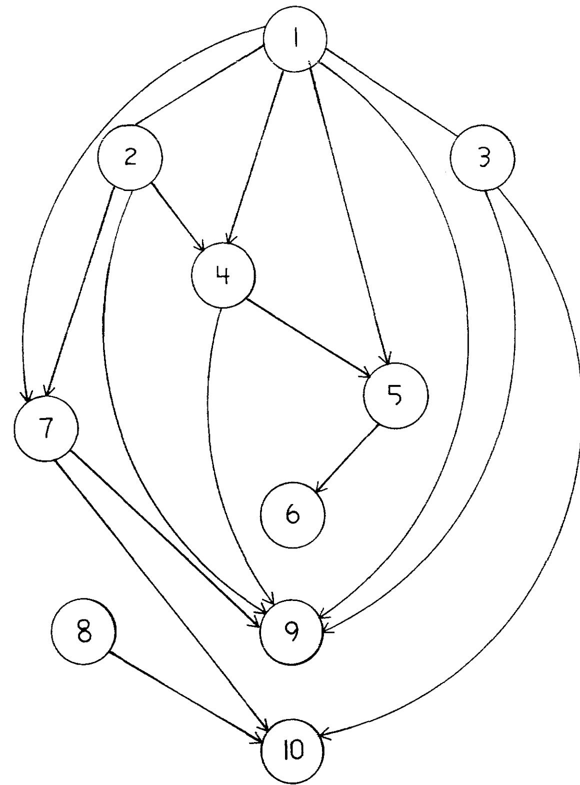


Figure 10—Graph of trip computer

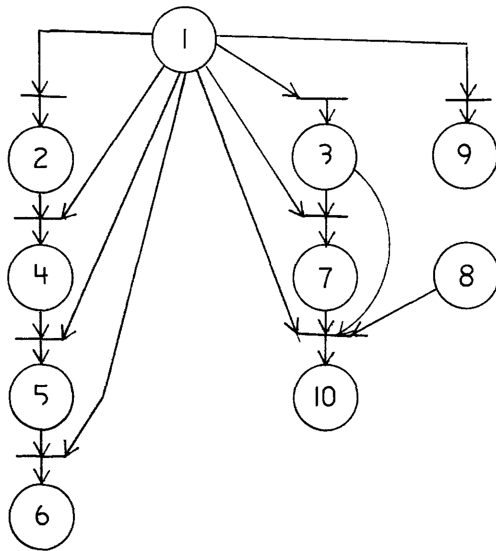


Figure 9—Petri net of trip computer

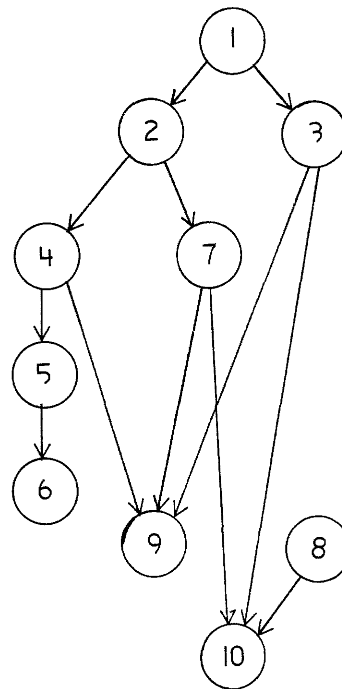


Figure 11—Maximally parallel graph of trip computer

ing on the speed of execution of Tasks 2 and 3, the task system could function on at most three processors concurrently.

One assumption made here is that the dependencies of some of the variables upon being updated by Task 9 before any of the tasks could run is eliminated by an initialization step in Task 1. What this means is that the first execution of the task system starts with Task 1 as noted and that after this the results of Task 9 are used, since the task system is basically completed at Task 10 and then recycles back to Task 1. This recycling can be represented, or left out, without affecting the way the task system would function. In this case it was left out for clarity.

Having performed the above exercise, the authors found the precedence graph approach to be easier to apply, since from Table II the calculation of the X relation is straightforward. From the X relation a set of ordered pairs could be defined which in turn provides the transitive closure and hence, could easily produce a precedence graph. The production of the Petri net proved to be more difficult and time-consuming, since no calculation could be performed to create the ordering for the graph. However, both procedures could be computerized to perform the work after the initial task specification and identification of variables required had been performed.

The Petri net, in the authors' opinion, presents more of the details of the intertask relationships, since the transitions indicate that variables are input and output to cause the transitions to fire. However, as stated above, the approach using the production of a maximally parallel precedence graph was easier to implement. Perhaps a method of combining the two techniques would be possible; however, that lies beyond the scope of this paper.

CONCLUSION

This paper has presented the concept of multitasking as a design methodology for the production of software to execute on a distributed/multiprocessor processing system. Two graphical techniques were presented: the Petri net and the precedence graph. These techniques offer a means of visualizing the flow of control in such a software system. Both Petri nets and precedence graphs have been analyzed and methods found to find concurrent segments of these graphs. However, in the final analysis, all these methods still require the designer to be cognizant of potential concurrency in the decomposition of his or her application into tasks to which the above techniques may be applied.

REFERENCES

1. Flynn, M. J., and J. L. Hennessy. "Parallelism and Representation Problems in Distributed Systems," *1st International Conference on Distributed Computing Systems,* Huntsville, Ala., Oct. 1–5, 1979. Piscataway, N.J.: IEEE Publications, 1979. pp. 124–130.
2. Kieburtz, R. B. "A Hierarchical Multicomputer for Problem-Solving by Decomposition." *1st International Conference on Distributed Computing Systems.* 1979. pp. 63–71.
3. Tjaden, G., and M. J. Flynn. "Detection and Simultaneous Execution of Independent Instructions." *IEEE Transactions on Computers,* C-19, (1970), pp. 889–895.
4. Kuck, D., et al. "Measurement of Parallelism in Ordinary Fortran Programs." *1973 Sagamore Conference on Parallel Processors,* Sagamore Lake, Syracuse, Aug. 22–24, 1973. Piscataway, N.J.: IEEE Publications.
5. Coffman, E. G., and P. J. Denning. *Operating System Theory.* Englewood Cliffs, New Jersey: Prentice-Hall, 1973, pp. 31–43.
6. Ramamoorthy, C. V., and G. S. Ho. "Performance Evaluation of Asynchronous Concurrent Systems Using Petri-Nets." *IEEE Transactions on Software Engineering.* SE-6 (1980), pp. 440–449.
7. Toulotte, J. M., and J. P. Parsy. "A Method for Decomposing Interpreted Petri-Nets and Its Utilization." *Digital Processes,* Vol. 5 (1979), pp. 223–234.
8. Auguin, M., et al. "Systematic Method of Realization of Interpreted Petri-Nets." *Digital Processes,* Vol. 6 (1980), pp. 55–68.

The use of performance models in systematic design

by K. M. CHANDY, J. MISRA, R. BERRY, and D. NEUSE
University of Texas
Austin, Texas

ABSTRACT

The paper describes a top-down methodology for evaluating the performance of computer/communication systems and describes tools that help in implementing the methodology. It also deals with performance analysis during the design of new hardware and/or software systems. The goal of the methodology is to detect and correct performance problems early in the design cycle.

INTRODUCTION

Much has been written about systematic design.[1,2] This paper describes tools that aid in designing to meet performance goals. We briefly review top-down performance design methodology as expounded in Smith and Browne and elsewhere and then show how the methodology is implemented by the use of appropriate tools.

In a data processing system, entities called transactions consume resources. At different stages in the design cycle it is convenient to view transactions and resources at different levels of detail, incorporating more detail as the design proceeds. For instance, at an early stage in a software design, a transaction may be "opening a bank account" or "sending a reminder about accounts past due." At a later stage in the design cycle the "send reminder" transaction may be refined into a sequence of smaller transactions such as:

- If amount owed exceeds AMOUNT-LIMIT, start transaction to cancel credit.
- If account is past due for a period exceeding TIME-LIMIT, start transaction to send warning letter and notify collection agency.
- Send reminder to customer.

Note that the manner in which a transaction is refined may be conditional; i.e., the refinement may depend on certain conditions (such as "amount owed exceeds time limit"). The refinement must also indicate which subtransactions may be executed in parallel and which must be executed sequentially. We will discuss a formal method for specifying the refinement of transactions later.

It is convenient to view resources as logical resources, physical resources, and a mapping from logical to physical ones. For instance, a transaction specification might state that a transaction needs to execute 10,000 lines of instructions; here the resource specification is logical, because the identity of the CPU to be held (if there are several) is not specified, and neither is the duration of time for which the CPU is held. Of course, the duration of time for which the CPU is held will depend on CPU speed. It is convenient to separate the hardware (physical-resource) specification from the software (transactions requesting logical resources). An application program will have the same specification no matter what hardware it runs on. A physical computing system, computers (CPUs, channels, controllers, disks), and message communication links will be specified in the same way, no matter what application programs run on them. Changing the way an application program runs (its priority, the specific site at which it runs, the allocation of databases to a different set of disks) only changes the mapping from logical resources to physical resources.

If both hardware and software are being designed, the specifications of the physical resources are also refined as the design proceeds. If the hardware is in place and only the software is being designed, the specifications of the physical resources are available in detail and no refinement is necessary. At the start of a design, a physical resource may be a computer; as the design progresses, this physical resource may be refined into a memory resource, CPU resources, and an I/O resource. At even later stages in the design the I/O resource may be refined into channels, controllers, and disks.

The refinement of transactions and resources may not proceed in a synchronized fashion. Either the application programs may be already given while a new hardware system is being designed, or the hardware system may be given while a new application is being designed. To allow a decoupling of application program design and hardware design, the logical mapping must be capable of mapping any level of transaction definition to any level of hardware definition. The mapping is discussed later.

In the next section we describe a tool called Performance Analyst's Workbench System (PAWS) that can be used to predict performance at various stages in the design cycle, given various levels of definition of transactions and hardware. PAWS is derived from other languages, notably RESQ.[3] This discussion also applies to RESQ.

PAWS

PAWS is a tool developed by Information Research Associates.[4] We will not describe PAWS in depth, but we will describe it in enough detail that the reader can understand how PAWS can be used to predict performance within the framework of a systematic design methodology.

PAWS is a very high-level simulation language designed specifically to model computer/communication/office systems. The specificity of the problems PAWS was designed to handle is both its strength and weakness: computer/communication systems can be modeled easily, whereas GPSS, SIMULA, or SIMSCRIPT are probably preferable for general-purpose simulation. Resources modeled by PAWS include memory, buffers, CPUs, and I/O devices. Scheduling disciplines to handle memory in PAWS include first-fit and best-fit. New disciplines and new resource features are being added. Thus it is easy to study computer design problems such as the memory fragmentation problem in PAWS. A variety of scheduling disciplines are available for resources such as CPUs.

Entities that use resources in PAWS are called transactions (see Figure 1). There are several facilities to control the manner in which transactions use resources; facilities include branching on condition (to simulate if-then-else, while loops, go tos), probabilistic branching (to simulate nondeterminism), interrupts (to simulate one transaction's being inter-

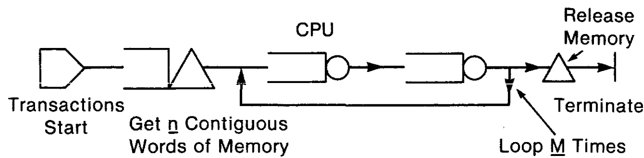


Figure 1—A diagram showing how a transaction uses resources

rupted by some external action) and forks/joins (to simulate parallelism). Transactions have local variables, and the system has global variables that may be accessed by all transactions. Transactions acquire and release resources at resource-manager nodes. For instance, a transaction may go to a memory management node and request a block of 100 words of contiguous memory. The transaction will have to wait at the memory management node until its request is satisfied or it is interrupted. Thus the sequence in which transactions request resources can be represented by a graph with resource management nodes, decision nodes such as forks/joins (to control the flow of transactions), and edges showing how transactions use resources.

Though the specific sets of resources and transaction control facilities vary from one computer modeling language to another, the key ideas are common.

In the next section we show how computer modeling languages can be used in systematic design. To help focus our discussion, we shall consider one specific language, PAWS, though the discussion applies equally well to languages such as RESQ.

Design Methodology

Performance modeling in the design of computer/communication systems occurs in three categories: modeling of (1) application programs (transactions), (2) hardware (resources), and (3) the map between application programs and hardware. We now consider each of the three categories in turn.

Application Programs

The first step is to identify all transactions at a coarse level of detail—the business level. An example of a transaction at this level of detail is "adding to a checking account." The logical resources used by the transaction are identified next, also at a coarse-grained, or business, level of detail. Examples of logical resources for the adding-to-checking-account transaction are drive-in-windows, bank tellers, communication lines, and computing facilities. At this point in the design it may not be appropriate to describe the computer facilities resource at a finer level of detail; thus all resources of bank computing—terminals, CPUs, disks, databases—are lumped together into a single entity. If a bank is considering setting up small suburban drive-in branch offices, resources such as real estate and personnel may be more important than issues such as the number of disk accesses. At early stages in the design, the designer may be concerned with questions such as these: How many square feet, drive-in windows, and bank tellers will I need? What is the *overall* additional load on computing facilities? If answers to these general questions suggest that the new application is cost-effective, the design should proceed further.

We shall now study the problem of specifying demand for a composite resource such as a computing facility. An adequate level of detail for specifying the load placed by a new application on computing resources can be derived from accounting data. The amount charged for running a transaction on a system is a function of the amounts of resources consumed. In many cases, computer centers use linear functions such as *service units,* with weights attached to consumption of the different resources—CPUs, I/Os, memory. Our goal at the first stages in design is to estimate the service units (or some other composite accounting entity) required by each transaction in the new application. Using service units or an accounting function is a very imprecise way of estimating load, because a computing system is not a homogeneous entity, but consists of very different parts. However, this level of detail may be sufficient at the first stage in design. The level of detail used in modeling real estate (drive-up windows), personnel, and computing depends on their relative incremental costs in the new application.

The amount of service units required by a transaction in a new application is estimated by comparing the new application with an existing one. No attempt is made to analyze the application in detail. It is quite common to hear an analyst say, "In Company X they handle about 20,000 transactions of an application very similar to our new application, and their application takes about 30 percent of their machine." It is from such imprecise statements that the first models should be built, because such statements give one a ballpark estimate of load. As a first guess, we might estimate that each transaction in the new application takes 0.3/20,000 hour of a system equivalent to Company X's.

An initial model for a new drive-in-banking facility may have the form shown in Figure 2.

Figure 2 has also set the level of detail for the resources. For instance, we assume M bank-tellers, all of whom are equivalent, though in reality the bank manager or supervisor may be the only one who can handle special transactions. We have also assumed that there is enough bankwidth in the communication lines linking the tellers to the computers that communication delays may be ignored.

The mapping of logical resources to physical resources is straightforward if the computing system is centralized: all logical computing system requests for all transactions map into the same centralized computer. When the computing system is distributed, the map from logical to physical states where each transaction will be processed. This map may be dynamic; however, as a first pass, it is often sufficient to assume that the

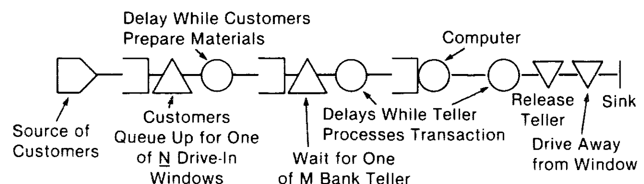


Figure 2—Describing a business transaction

map is fixed by the designer. The map becomes a critical design variable. The amount of logical resources requested is stated in some standard units, usually with respect to a standard system. An important part of the logical-physical map is to specify the power of the computing systems at the various sites. The scheduling discipline we assume for the computing system may not correspond to any real discipline, because a multiplicity of disciplines may be used for the different component resources. Since several transactions may be processed in parallel by the computing system, it is natural to assume either a priority discipline or a processor-sharing discipline (if all transactions have equal priority).

After constructing transaction diagrams such as those in Figure 2 for all the new "business" transactions, and after modeling the existing load on the system in an approximate manner, we are now ready to conduct modeling experiments to help guide the design. The next step is to identify the design parameters that need to be studied at this stage in the design. In our running example the design parameters are N, the number of service windows; M, the number of tellers; the priorities of each transaction type; and the logical-to-physical map. Since we wish to model several branch offices, it is convenient to construct a model with an array 1 . . J of branch offices, and to refer to parameters $N(i)$ and $M(i)$ of the ith branch office, $i = 1, . . , J$. Assume that we have K distinct physical computing systems that may be distributed. The logical-physical map is an array L, where $L(i)$ is the index of the physical computing system on which transactions from the ith branch office are run. All the parameters $L(i)$, $N(i)$, $M(i)$, the priorities, and the relative powers of the computing systems at the different sites are easily set in PAWS; running a variety of experiments and changing the logical-physical map can be carried out simply.

The experiments will show the designer how the various resources (drive-up windows, tellers, and computing facilities) interact and contribute to customer service levels. For instance, if the logical-physical map is a poor choice, contention for computing facilities will cause the tellers to take longer for each transaction, thus causing cars to remain at drive-up windows for longer periods of time, which in turn may cause automobile traffic jams around the drive-in area! Since computers play such a ubiquitous role in business, it is important to model data processing as an integral part of business rather than in isolation.

Cost, investment, and tax-related factors and personnel policy often play a more important role in data processing system design than the intricacies of resource-scheduling algorithms. Simple models based on PAWS, RESQ, or other modeling systems will help designers and accountants make tradeoffs early in the design. Performance analysts tend to ignore the huge, vital tradeoffs to be made at the business level and focus on the relatively unimportant tradeoffs at the data processing subsystem level. Modeling systems can be used to look at tradeoffs at all levels, starting with tradeoffs at the business level and proceeding hierarchically to increasingly fine tradeoffs.

Even for simple models the space of design parameters is vast. Use of models early in the design stage will help narrow the design space that needs to be considered in later stages of design.

Refinement

As the design proceeds into greater detail, the application/hardware/mapping models become refined. The assumptions made in the earlier stages of design should be checked against the results of the refined models. If these latter results show that the assumptions made in the earlier models are grossly in error, the decisions made in the earlier stages of the design cycle must be reexamined.

Refining a transaction model consists of (1) describing a transaction in terms of more detailed and smaller transactions and (2) specifying the logical resources at a greater level of detail. For instance, a transaction to process accounts past due that is specified as using x seconds of a standard computing system may be partitioned into several transactions, as shown in Figure 3.

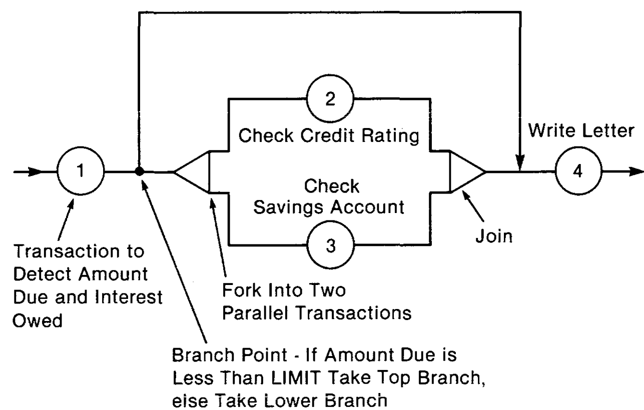


Figure 3—Refining a business transaction

In this example a single transaction at a coarse level of detail is partitioned into four smaller transactions. If the amount due is less than LIMIT, the single business transaction manifests itself as two transactions, (1) figure amount owed and (4) write letter. If the amount due is greater than LIMIT, the single business transaction manifests itself as four transactions: after completing 1, 2 and 3 are executed in parallel; after both 2 and 3 are complete, 4 is executed. Resource demands for the smaller transactions are computed in the usual way. Branch points such as those shown in Figure 3 are normally modeled as probabilistic branches; some fraction of all transactions takes one branch, and some fraction takes the other. The modeling languages have explicit facilities for models of fork/join and routing, so transaction refinement is straightforward.

In addition to specifying a business transaction in terms of more detailed transactions, the refinement step may also specify logical resources in greater detail. For instance, instead of specifying a transaction's demand for data processing resources in terms of x seconds of system time, we may specify the demand in terms of $x1$ units of memory, $x2$ units of CPU time, and $x3$ I/O accesses to data sets. The specification of which data sets are accessed and the relative frequency of access to each of the data sets is not given here; but it will be given, with further refinement, in later stages of design.

As the design proceeds, the refinement of the specification of the physical resources will continue as well. Thus a computing system may be refined into memory, CPU, and I/O devices. If the refinement of transactions and physical resource models proceed hand in hand, the logical-physical mapping is straightforward. Let us consider the case where the hardware design lags behind the application program design. Suppose the hardware model is still that of a composite computing system. In this case the logical-physical map must map a relatively refined application program onto a relatively gross hardware model. In the initial design stages, the load placed on data processing resources by transactions is in terms of a single metric (processing time or service units) on a "standard" or benchmark computing system. The refined transaction model will result in better estimates of resource demand, though once again the estimates will be made with respect to the standard system. As long as the hardware model remains at the composite computer system stage, resource demand must be estimated as a single metric on a standard system and the proposed hardware must be defined in terms of its speed relative to that of the standard system. Now consider the case where the hardware model has also been refined into distinct physical resources—memory, CPU, and I/Os. The logical-physical map does not change, because once a transaction is assigned to one computing system, all logical-resource requests (memory, CPU, and I/O) will refer to physical resources within that computing system. Thus the map could continue to be in the form of an array L. The use of the physical resources by transactions in this more detailed model is also easily represented (Figure 1).

In the detailed model of the data-processing system, the other aspects of the business should not be modeled at all or should be modeled in an extremely approximate fashion. For instance, we considered a problem in which, at the first stage in the design, tradeoffs were made between real estate (drive-in windows), personnel (bank tellers), and data processing, using approximate models of the data processing system. At the next stage we construct more detailed models of each of the subsystems, including data processing. However, at this second stage we will ignore the other components, such as drive-in windows and bank tellers, and focus primarily on the data processing system. The load offered to the data processing system in terms of transactions per hour is obtained from previous models, but no other aspects of the previous model are brought into the current model. The objective is to keep each model at a manageable size. Only if the results of the detailed model show that the assumptions made in the coarse model are grossly erroneous do we go back to the coarse model.

The use of modeling languages in the design process in a systematic, top-down refinement procedure is extremely helpful in catching performance problems early. Moreover, the methodology is manageable, and the tools to implement the methodology exist.

SUMMARY

We have shown how modeling languages such as PAWS and RESQ can be used with performance design methodologies such as those in Smith and Browne[1] and Le Mer.[2] Our approach consists of building separate models for application programs, hardware, and the map from programs to hardware. At each step in the design the models for application programs and hardware are refined. We have shown that these models can be represented naturally in modeling languages.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the U.S. Air Force, AFOSR-81-0205.

REFERENCES

1. Smith, C. U., and J. C. Browne. "Performance Engineering of Software Systems: A Case Study." *AFIPS, Proceedings of the National Computer Conference, 1982* (Vol. 51).
2. Le Mer, Eric. "MEDOC—A Methodology for Designing and Evaluating Large-Scale Real-Time Systems." *AFIPS, Proceedings of the National Computer Conference, 1982* (Vol. 51).
3. Sauer, C. H., E. A. MacNair, and J. F. Kurose. "The Research Queueing Package: Past, Present, and Future." *AFIPS, Proceedings of the National Computer Conference, 1982* (Vol. 51).
4. Information Research Associates. "PAWS: The Performance Analyst's Workbench System User's Manual." IRS, Austin, Texas, 1981.

Performance modeling in the design process

by WILLIAM ALEXANDER and RICHARD BRICE
Los Alamos National Laboratory
Los Alamos, New Mexico

ABSTRACT

Performance modeling and analysis of computer systems are often ignored during the project design phase in favor of other techniques collectively known as structured design or software engineering. We describe benefits that can result from including performance analysis as an integral part of the design process. Several different goals, time frames, and roles played by performance analysis during system design are illustrated by three case studies of current projects at Los Alamos.

INTRODUCTION

In the past decade, since the ideas collectively referred to as *structured programming* and *structured design* have gained acceptance, the conventional wisdom has become that one should ignore performance considerations in the design phase of a computer system project in favor of modifiability, maintainability, and correctness. This doctrine assumes that one can always "tune" the system to meet performance criteria after it is built.[1] If one is astute enough in design choices so that the first running version of the system comes within, say, an order of magnitude of the performance goals, these goals often can be met by relatively simple modifications; in this case the attention paid during design to understandable structure and modifiability will be rewarded. But performance is a result of interactions among many elements of software, hardware, and environment, and sometimes these interactions are counterintuitive. It is not uncommon for systems to fall so far short of performance goals that only fundamental, and therefore very expensive, changes will serve.

It is our experience that the chances of such catastrophes can be reduced by applying modeling techniques during the system design process in such a way that none of the advantages of structured design need be sacrificed.

The models we refer to in this paper may be analytic or simulation; in some cases the simpler methods of operational analysis are adequate.[2] The point is that sufficient data should be collected, and sufficient analysis done on them, to give some assurance that the performance goals of the system being designed will be met. The kinds of data that must be gathered include hardware and software characteristics of the components of the system; measures of the behavior of the environment in which the system will run, including workload and competing systems; and even the ways in which the system being designed will alter the existing environment. Because most design projects have deadlines, it is much more likely that sufficient performance analysis will be included if the installation already has models of the proposed system's environment, or at least has data collection facilities installed.

Differences in performance goals as well as in other design objectives imply that modeling will assume different roles, occupy different time frames, and require different information from one design project to the next. Three design projects with which we have been involved at Los Alamos illustrate some of these points.

COMPUTING AT LOS ALAMOS

At Los Alamos, the integrated computing network (ICN) allows all validated computer users at the laboratory access to almost any of the machines or services of the Central Computing Facility (CCF). Figure 1 is a schematic diagram of the ICN. (Dotted portions indicate future plans.) At the "front end" of the network (the right side of the diagram) over 1,350 terminals and remote entry stations are concentrated in stages to front-end switches (the SYNCs) so that traffic can be routed between any terminal and any worker computer. The worker computers include four Cray-1s, four CDC 7600s, two CDC Cyber-73s, and a CDC 6600. Each of the worker computers is connected to the file transport (FT) switches and so to the "back end" of the network (left side of the diagram). The FTs are the means by which workers can send files to each other and to the special-service nodes in the network. The special services provided at present by the network include an output station (PAGES) to which are attached a wide variety of printing and graphics devices, a mass storage and archive facility (CFS),[3] a gateway that handles file traffic between workers and computers outside the ICN, and an integrated performance monitoring and batch job control station (FOCUS).

Although all types of computing are done at Los Alamos, most of the CPU hours on the large workers are spent executing large, long-running scientific programs. Many of these produce graphics output. Some users have a need to run programs larger and longer than even our present worker computers can handle.

THREE CASE STUDIES IN DESIGN

The distributed interactive graphics project

The goal of the distributed interactive graphics project is to improve user productivity by improving the performance of an existing interactive graphics system that runs on a large scientific computer (a Cray-1 or a CDC 7600).[4] It is hoped that system responsiveness can be improved by adding an intelligent terminal or a larger minicomputer as a front end and by distributing the software between the two computers. The front end is intended to handle graphics-terminal or device interactions and drive the graphics screen. Design issues include the choice of minicomputer, the hardware and software constituting the link between the two computers, the distribution of the graphics software, and whether the distribution should be static or dynamic.

A simple model of this system might include CPU and memory on the two computers and simple links between the two computers and between the front end and the terminal. Input to this simple model would include the speed and size of the CPUs and memories and the bandwidth of each link. We would also need the CPU burst sizes and distributions and the

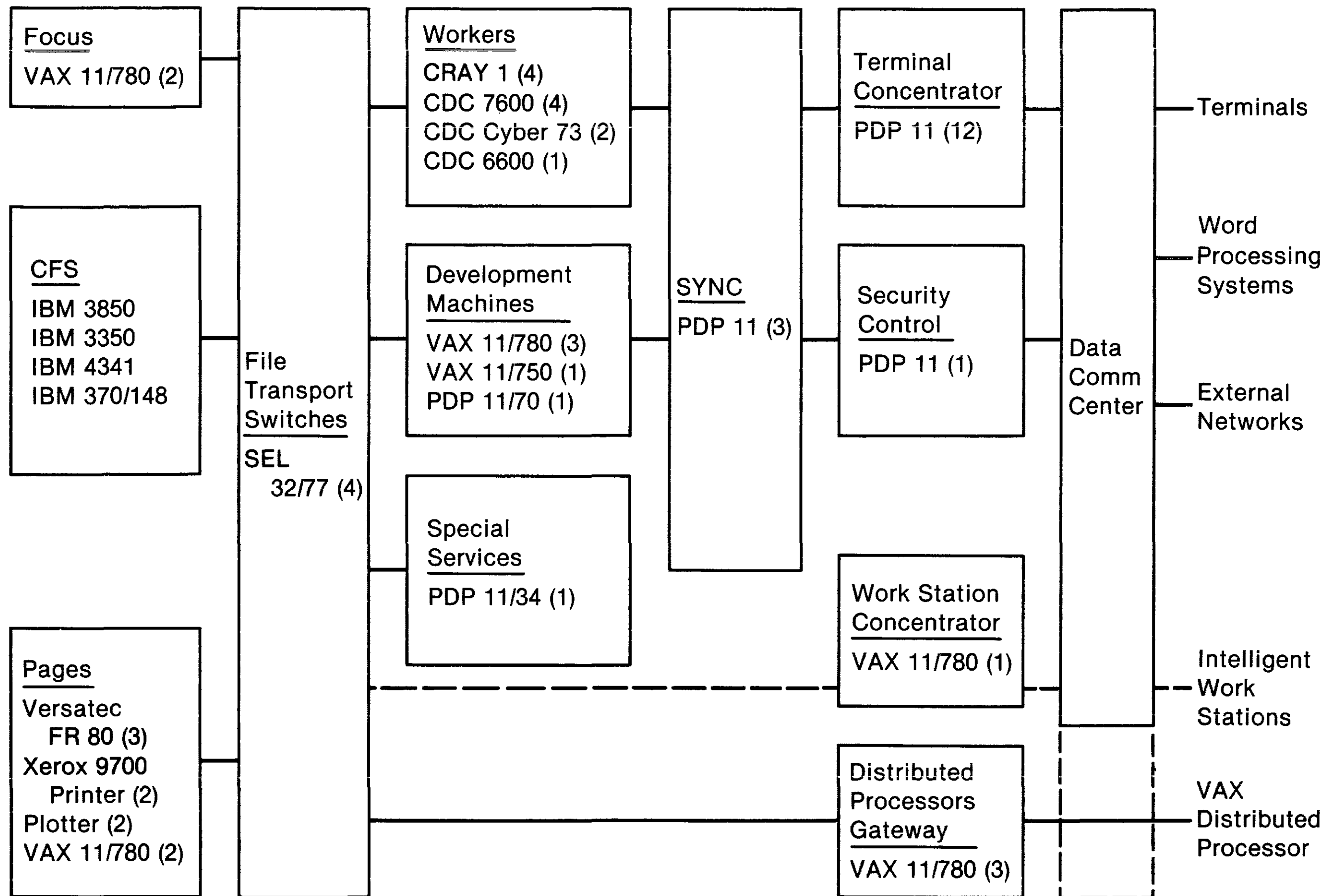


Figure 1—Los Alamos integrated computing network

frequency and size of communications over each link for a given distribution of the software. All this information will probably be known or can be obtained by the system designers. This model can only give an order-of-magnitude estimate of performance and tell the designers whether a component of the system is an obvious bottleneck.

A more realistic model of this system would incorporate other information not so readily available to the application designers. The link between the two computers envisioned by the designers is actually the back end of the network depicted in Figure 1. Contention for network services, overhead in communication protocols, error rates, and buffer space within the network are all likely to reduce the effective bandwidth of the communications links. Contention for CPU and memory resources on the two computers will also alter the communication bandwidth and the rate at which the distributed application can execute. As the distribution of software between the two computers changes, the competition this system introduces into each of the two computer systems will change also. Quantitative knowledge about the effects of these factors will be available only if there has been an ongoing data collection effort on the mainframes and the network. There probably would not have been time or staff available to obtain the information for the design effort if it were not already being collected.

One way to get around the problem of missing information is to distribute a prototype application and measure the effective communication bandwidth and rates of CPU service. The designers did this, using a DEC VAX 11/780 as the front end. The initial performance results were discouraging. The designers concluded that the chosen division of software between the two computers was wrong. They also observed that the number of ways of splitting the application was too large for exhaustive trial and error. Yet the frequency and size of communications between each pair of modules was not known, and the model needed this information to predict performance for a given split of the software. The designers therefore invested the time to build distributing tools that automated the code conversion process, allowing them to move modules from one machine to another more easily. This effort turned their prototype into a flexible data collection tool for performance analysis, as well as providing the designers with useful insights.

The prototype multiprocessor project

The prototype multiprocessor project has as a short-term goal the production of a tool for evaluating various approaches to parallelizing certain classes of numerical com-

putations.[5] A longer-term goal is to determine the properties of applications that are candidates for multiprocessing and to determine the properties of a multiprocessor that will efficiently execute the applications at Los Alamos. The prototype is explicitly intended to serve as a data collection device in designing or specifying an eventual production multiprocessor to meet production performance goals. The prototype is required to be as flexible as possible to allow hardware simulation of a variety of architectures, such as banyon, perfect shuffle, star, ring, and hypercube networks. Its flexibility and self-measurement abilities are more important than its absolute performance.

Clearly, however, the prototype must perform well enough to allow timely and reliable execution of all test cases of interest. Thus performance analysis and modeling are appropriate during its design to evaluate choices of CPUs, memories, and hardware and software communication mechanisms. The data required to support this first-phase performance analysis include the characteristics of the hardware components under consideration and the overhead costs of the communication protocols, all of which should be easily obtainable. We also need computational and communication characteristics of the application programs resulting from a given parallelization; here previous execution measurements and analysis of the individual application programs would be extremely helpful.

This prototype will not be installed in our ICN, although some provision will probably be made for downloading programs to it. Thus the hardware will run in isolation, and we need not take environment into account when modeling its performance. In another sense, the existing large numerical programs that will be run on the prototype constitute its environment, and information about their current performance characteristics will aid performance analysis on this prototype just as information about the network and operating systems did for the distributed graphics project.

Los Alamos has a long history of using supercomputers for scientific computation. The results of this experience have always been made available to vendors; by explaining the computing needs of scientists and by specifying the performance goals which each new generation of supercomputers must meet, the laboratory has contributed to their design. In the second phase of the multiprocessor project, performance modeling can be helpful in scaling up performance projections from the prototype to various production alternatives. Not all components will be speeded up by the same amount in going from prototype to production version, and some communication functions that were being simulated in software may be implemented in hardware. The data for this model will come, we hope, from the prototype.

The network switch project

The goal of the network switch project is to design a network switch to replace the SEL 32/77 minicomputers that act as file transports in the back end of the ICN.[6] The new switch will have more ports so it can be connected to more network nodes than each SEL can, and it will have hardware support for error detection so that the network can provide more reliability without software overhead. The performance of the current FTs is quite satisfactory, so improving performance is not a primary motivation for this project. Building a prototype of the proposed new switch is unnecessary, because the present FTs serve very well in that role.

It is desirable that the new FTs meet performance needs imposed by ever-increasing message traffic rates in the back end of the network as far into the future as possible. A relatively simple model of the proposed switch and its environment was constructed to investigate its performance under loads in excess of those observed at present. The data needed for this model included characteristics of the CPU, buffer memory, channels chosen, and current back-end network message rates. Once again, the fact that these message rates were already being collected made the modeling effort more practicable.

Merely increasing the present message rates with the same distribution of large versus small messages and with the same set of nodes currently comprising the network constituted a reasonable extrapolation of future workload for the FTs for two or three years. But predicting workload growth in any computing system more than a few years in the future is practically impossible because there are certain to be qualitative changes in the structure of the system as well as in the way people use it. We have discussed this problem in Alexander and Brice.[7] Our approach was to vary all the workload parameters over a wide range of values. In this way we were able to predict what loads the proposed system would handle and the characteristics of loads that may cause its performance to deteriorate.

MODELS VS. PROTOTYPES AS PERFORMANCE ANALYSIS TOOLS

In these three examples, we have seen both models and prototypes play differing roles in the design process. Obviously, prototypes serve many useful purposes in design, but we are interested here only in their uses for performance prediction and their relationship to models. Models and prototypes have different strengths and weaknesses as performance prediction tools.

Prototypes can provide order-of-magnitude performance information, especially if they can be installed in the actual environment in which the production system being designed is to run. But as performance predictors they have three major drawbacks.

1. It is difficult to extrapolate measured service rates of a prototype to the production system, because it is not possible to predict accurately how the different behavior of the production system will interact with its environment. The production system will presumably have memory size, computational needs, communication behavior, and interaction rates different from the prototype, and these will affect the environment as well as being affected by it. Models can incorporate the environment.
2. Although prototypes usually can be used to predict the effect on performance of simple changes in the system, they cannot do the same for changes in the environment.

For example, we know that the distributed graphics system will run faster if we are allowed to make one simple change in the scheduling algorithm of the operating system used on the large scientific computers, but this fact could be learned only from a model that incorporated the operating system.

3. Prototypes usually cannot be made as flexible as models; hence, prototypes cannot be used to predict the effect of fundamental design changes. This point requires further discussion.

Flexibility can sometimes be achieved in prototypes, but usually at higher cost than in models. Prototypes are not always inflexible, but it is unusual to spend so much time and effort in the computer system design process on a prototype. The multiprocessor prototype was consciously designed as a hardware simulator of a variety of multiprocessor architectures, and this capability makes it very like a model. One can imagine a continuum characterized by increasing cost and complexity as one moves from operational analysis through models and prototypes to the actual production system. The design process is characterized by making choices among alternatives; typically one can try different alternatives more quickly and much more cheaply with a model than by actually implementing them. There is a subjective cost/benefit function that applies to the choice between trusting the results of a model and implementing a prototype. The cost of implementing a prototype is usually more easily justified in extreme situations, such as designing with new technologies (including new software technologies) or in completely unfamiliar situations.

Modeling as part of the design process produces benefits besides performance prediction. Because modeling is a relatively quick method of "implementing" a design, issues that normally come up only during implementation sometimes arise much earlier. Ambiguities in the specifications may be noticed early, and if these would have necessitated redesign, time and money can be saved. Beyond merely predicting performance, modeling can specify the performance levels that individual components will have to achieve for the whole system to meet its performance goals. Modeling can also help explore the behavior of the system under a variety of workloads or other external conditions.

Although modeling is an art requiring some expertise, it is not nearly so difficult as it used to be. A number of commercial packages and languages are available to support analytic and simulation modeling of computer systems.[8,9] It is not unreasonable to develop models and in-house modeling capabilities with the aid of these tools, and the investment can pay repeated dividends.

There are, of course, difficulties in integrating modeling into the design process. It is particularly difficult to analyze the performance of software that has not been written. Although the work of Smith and Browne[10–14] is promising, much remains to be done. We do not imply that performance modeling as part of design always works or can answer all questions, only that it has proved useful in our experience.

CONCLUSIONS

Here, in capsule form, are some lessons we have learned in trying to integrate performance modeling into the design process:

- Performance modeling should play a central role in system design; ignore it at your peril.
- The role of performance modeling is not the same in all design projects. Clearly specify your performance goals and what factors will affect performance; then try to model those factors.
- Obtaining the data for the models can be a major problem; ongoing measurement projects are always worthwhile.
- Prototypes can be valuable data-gathering tools if they are instrumented for this purpose.
- Anticipate the effect of environment on the system you are designing and the effects of the system on the environment.
- Include the performance analyst on the design team from the beginning; if he/she is perceived as an "outsider," he/she is more likely to be ignored, especially if decisions have already been made.

REFERENCES

1. Yourdan, E., and L. Constantine. *Structured Design.* Englewood Cliffs, New Jersey: Prentice-Hall, 1979, pp. 290–291.
2. Denning, P. J., and J. P. Buzen. "The Operational Analysis of Queueing Network Models." *Computing Surveys,* 10 (1978), pp. 225–262.
3. Blood, M., R. Christman, and B. Collins. "Experience with the LASL Common File System." *Digest of Papers from Fourth IEEE Symposium on Mass Storage Systems, April 1980,* pp. 51–54.
4. Hamlin, G., and J. E. George. "Experiences with Distributing Graphics Software Between Processors," submitted to SIGGRAPH 82, Boston, MA, July 1982.
5. Trujillo, V. A. "Multiprocessor System Description." Los Alamos National Laboratory memorandum to Bill Buzbee, June 26, 1981.
6. Tolmie, D., R. Christman, T. Klingner, C. Stallings, and R. Jurgens. "Network Switch Functional Design Requirements." Los Alamos National Laboratory memorandum to Robert Ewald, March 31, 1981.
7. Alexander, W., and R. Brice. "Long-Range Prediction of Network Traffic." Proc. CPEUG81, November 1981.
8. *ASPOL Reference Manual.* Minneapolis: Control Data Corporation, Pub. No. 17314200-B, 1975.
9. *The Performance Analyst's Workbench System Users Manual.* Austin, Texas: Information Research Associates, 1981.
10. Browne, J. C. "Designing Systems for Performance." *Proceedings of the ACM/SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* September 1981, p. 1.
11. Smith, C. U., and J. C. Browne. "Performance Specifications and Analysis of Software Designs." *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems,* August 1979.
12. Smith, C. U., and J. C. Browne. "Modeling Software Systems for Performance Predictions." Proc. CMG X, 1979.
13. Smith, C. U., and J. C. Browne. "Aspects of Software Design Analysis: Concurrency and Blocking." Proc. Performance 80, May 1980.
14. Smith, C. U. "The Prediction and Evaluation of the Performance of Software From Extended Design Specification." PhD Dissertation, Department of Computer Science, University of Texas, August 1980.

MEDOC: A methodology for designing and evaluating large-scale real-time systems

by ERIC LE MER
ECA Automation
Saint-Cloud, France

ABSTRACT

We introduce a global design methodology for large-scale real-time systems; it is based on such concepts as data-flow analysis, sequential processes, communication links, abstract architecture. These concepts give us a guide for designing real-time systems; two tools are also introduced, having a specific action in the design process: OGIVE, a Petri net analyzer, makes the verification and the validation of the abstract structure; OSCAR, analytical queuing network, oriented, evaluates quantitatively some implementation choices of the real system. These tools are integrated very early in the design process to make us sure we shall not have dramatic regressions to do for, eventually, redesigning the system.

INTRODUCTION

The main problem in developing large-scale real-time systems is (1) to be sure the system will do what it has to do; (2) to be sure it will do it in a proper way; (3) to be sure the system is a real-time one. This is why we need some techniques and tools defining a methodological environment, as an aid for the designer, which is as automated as possible. Such an environment is introduced in this paper; the focus will be on validation and evaluation aspects. The paper attempts to show how the continuity of the concepts is preserved during the design process according to a top-down methodology by stepwise refinements.

We first introduce two tools: (1) OSCAR (Outil de Simulation pour la Conception d'une Architecture Répartie) and (2) OGIVE (Outil Graphique Interactif de VErification)

OSCAR[1]

OSCAR is an analytical tool based on queuing theory concepts.[2, 3, 4] We can the use this tool in the following diagram:

Build the model

In this respect we have constructed a modelization language. The main features are (1) nets and (2) queues and chains. That means that in the nets we describe the global structure of the queuing network; we may also note the hierarchical ability of the designer to describe the internal structure of the network in more and more detail. For instance, in the first phase, it is possible to model something as a queue—which is, in fact, a macro-queue—and in a second step to refine this queue into a net description. We shall see further how to handle these subnets, which can be considered macro-queues.

In the queues we describe the main parameters of the network, which are as follows:

1. Local classes, by which we establish a relation between the server and the job. (Note: *The job* signifies the entity circulating through the network. It can be a batch job, a transaction, a DBMS request, a message, or a similar item. This means the job is characterized according to its specific behavior with the server it requires. One can see the local class as the semantics of the job.)
2. Service discipline, i.e., FCFS, LCFSPR, PS, IS.
3. Service rate, which can be independent of or dependent on the length of the queue of the server.
4. Type of server, active or passive. In fact, because of analytical restrictions, we only model the memory; and we suppose that, for instance, each job needs the same number of partitions.
5. Service time according to the class of job the server is proceeding with.

The chains are topological descriptions of the job circulation; therefore we can say that the chain is the syntax of the job. OSCAR can deal with open or closed chains.

The global model becomes a tree whose sub-root nodes are always nets and whose leaf nodes are queues and chains. For execution, we consider only the leaves.

Create a new library

At each step of the modelization, the designer may build nets or refine some queues issued from the previous step. These nets are put in libraries, which constitute the global model.

By the command language of OSCAR, one can:

1. Select a library
2. Merge several libraries
3. Overlap several libraries

in order to build more or less complex models by means of elementary descriptions. After each command the user is given a description of the net that has been built and validates it by creating a new library.

Run the model

Once the designer has built a satisfactory model, he can run this same basic model after initialization and eventual loop declarations on data. We have two analytical algorithms: NCA multichain and MVA multichain (NCA stands for *normalizing convolution algorithm* and MVA for *mean-value analysis*). The appendix introduces our version of the NCA multichain, which is nearly a straightforward extension of the NCA monochain. As far as I know, I have never seen the convolution algorithm with multichain developed in the literature. (Note: In the NCA the normalizing constant is not computed.)

According to the structure of the network and to the desired results, OSCAR chooses the best algorithm with some internal criteria. However, as networks get very large, it is no

longer possible to get exact results, and we need some approximation techniques. We can share these techniques in two sets:

1. Algorithm approximations[5] that allow us to reduce significantly (1) the run time and (2) the memory requirements.
2. Structure approximations that allow us to reduce the complexity of a model.

The basic concept is the aggregation technique.[6] According to the nature of the complexity, we can aggregate (1) jobs, (2) chains, or (3) queues. (Note: It is under this technique [aggregation of chains] that we handle open and mixed networks.) The selection of any kind of approximation can be manual (the will of the user) or automatic (entry point of OSCAR). One can find the outputs for a 3-chain academic example in the appendix, in French.

OGIVE[7, 8, 9]

OGIVE is a graphic and interactive tool dealing with Petri nets. The main features of OGIVE are the same as those of OSCAR; however, instead of modeling in terms of queues, we model by means of simple Petri nets. The general structure of OGIVE is the following one (Figure 1): The user is guided by

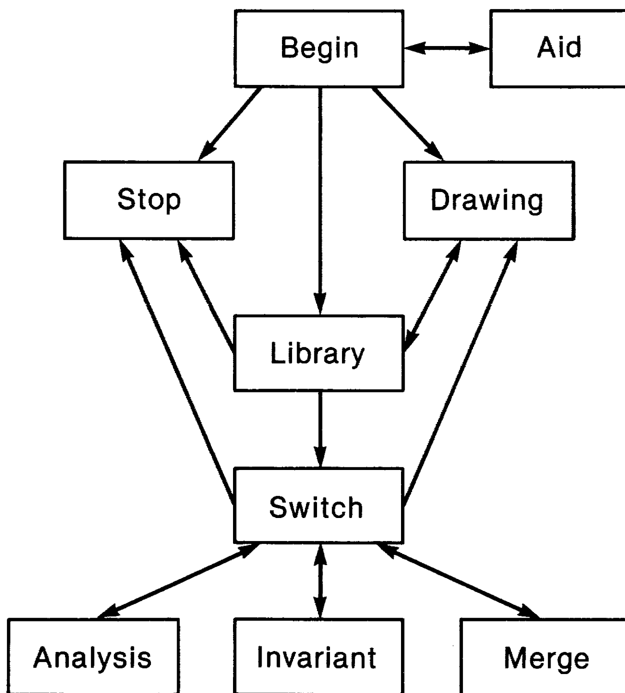


Figure 1—Menu structure

a menu, as shown on Figure 1, at each step of the modelization. There are two means to draw a Petri net:

1. A graphic method, menu-directed: places, transitions, and weights on the graph are drawn by showing the word on the menu and by pointing the pen on the screen. One can move, suppress, or mark on the drawing any kind of elementary entities.
2. An interactive method, menu-directed, by which we declare the nodes of the net, the next of each node, the weights and so forth.

Once the net is drawn, it is possible to keep it in a library; the state of the library can be obtained at the beginning of the session by BEGIN.

By the module MERGE one is able to merge several nets by two means—merging transitions and merging places—as soon as the nets are homogenous. The MERGE operation is manual; that is, the user must declare what places/transitions he/she wants to merge. Then it is possible to proceed to the analysis of the Petri net. The usual structural properties of Petri nets can be obtained by the ANALYSIS module (safeness, liveness, boundedness, marking graph, and similar properties); if the net is too large, it is possible to reduce the Petri net in order to eliminate some places or transitions. Of course, this operation must preserve the properties of the original net. This process must be interpreted as an abstraction process, because we reveal the skeleton of the model and thus abstract the implementation aspects.

Another way to analyze Petri nets is the use of the INVARIANT module. By this means we can get the place/transition invariants; elementary, minimal, or total invariants; or other invariants.

METHODOLOGICAL PROCESS

This section shows how these tools are integrated into the entire design phase and what they are supposed to do or to prove. Usually the design phase is shared in some two or three steps with, eventually, the possibility of regressions.[10,11,12] MEDOC suggests two steps: in the first one, called *functional design,* we define and specify what the system is going to do; in the second one, called *organic* design, we describe how the functions will work.

Functional design

At that level we isolate the main functions of the system—for instance, initialization, display, and computation—and we specify the interfaces of these with the environment, especially the flow of the data. This is why we have a description that is data-flow-oriented, with some basic entities, which are (1) processes, (2) dynamic data, (3) static data, and (4) sources and sinks. We establish the links between processes, sources, and sinks through dynamic data. The representation is graphic, with circles, squares, queues, slashes, and arrows. It is possible to translate this specification into a high-level specification language, by which we control the internal completeness of the description. Throughout the process we must respect two essential rules:

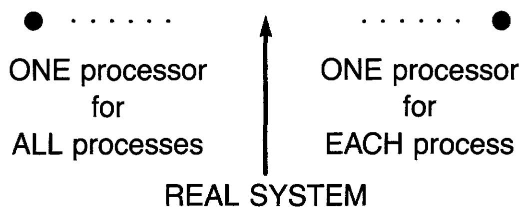
1. The refinement of a process P from level i to level $i + 1$ must be done on one sheet of paper.
2. From level i to level $i + 1$ all the interfaces must be preserved.

For each level we define (1) the (new) static and dynamic data to be *described* and (2) the processes to be *described* and *refined*. A process must be refined if its specification is not detailed enough to be described or if it still contains some external entities belonging to the environment. In this case we must discard this specification element from the strict specification of the process: for instance, in the design of a display function we usually find at the top levels the display screen and the operator, which do not belong to the implemented display function.

At the end of this phase, when *all* the processes are to be described, we can start the organic phase. We have hidden an important aspect of this design phase: at each level, in parallel with the functional specification, we build a queuing model that we may call an abstract model, since we do not actually take care of the implementation on a computer and in a data processing environment. We are more interested in the evaluation of a process, or a set of processes, if we do not have any hardware constraints.

Since the specification of the whole system is a tree, with OSCAR and the hierarchical models and merge operations, we are able to evaluate the performance of each process, or specific part of a process, or a set of processes, by grouping and merging the corresponding models. If we have some hardware constraints, we can obtain very early an evaluation and a prediction of the performance of the system and a first idea of the best implementation of the software. If we do not have such constraints, we can evaluate at least some configurations, centralized or distributed. For a study case we can optimize the process communications, the storage of the data, or similar items. In such a case we estimate our own constraints for the future hardware configurations in terms of CPU speed, mean access time for a disk, maximum number of terminals, speed of the communication lines, and so forth. In doing so, we express only a tendency of the future behavior of the system; we do not attempt to reveal the exact truth.

Here we have a dummy architecture in which each process is run on a specific processor and the communications between processors and processes are implemented separately. Even if this architecture is not realistic today, the actual architecture is nothing other than a simulation of the internal behavior of the designed system. We now have to choose between the minimum and the maximum one.



We have to consider two questions before making the choice: (1) Will the designed system work properly? (2) How will it work regarding the real-time constraints?

Before detailing the process, we show it globally in Figure 2, keeping in mind that the structure of the system is a connected graph, regardless of the implementation details (see

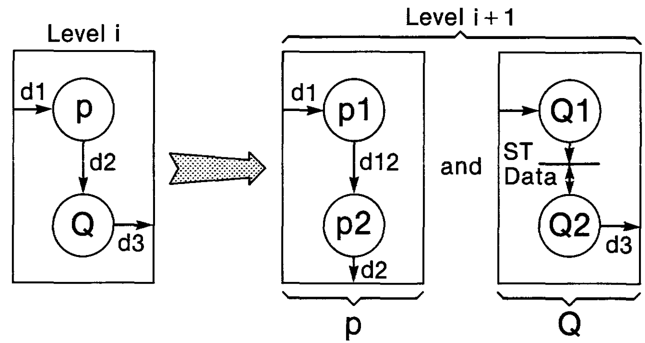


Figure 2—Refinement of level i

Figure 3). One can see that the process described in Figure 4 is a down-top process: i.e., we start from the basic descriptions and models of sequential algorithms (SA) and communications links (CL) and then by successive merge get the complete model. This is a direct consequence of the top-down design process.

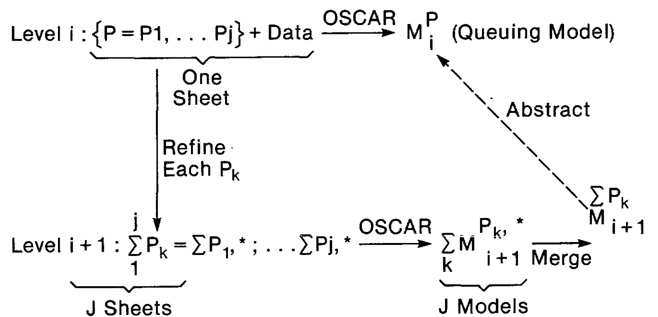


Figure 3—Design process

The first question is answered with the aid of OGIVE. By this means we can validate the qualitative structure of the communication links, especially parallelism, synchronization, deadlocks, starvation, and so forth. Once this work is done, we input time in the model(s) when necessary. To do so, we

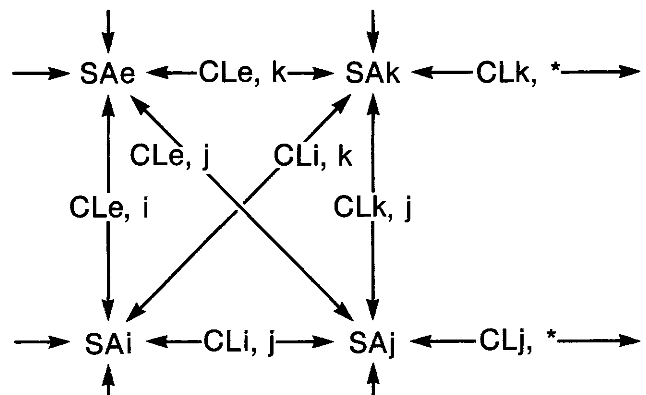


Figure 4—Graph structure of the system

have some macro-libraries at our disposal to describe some specific mechanisms, as, for instance, mutual exclusion, fork-join, send-receive, and remote call. These indications give guidelines for the coming organic phase and its future implementation in an operational environment. We can summarize the functional phase as shown in Figure 5.

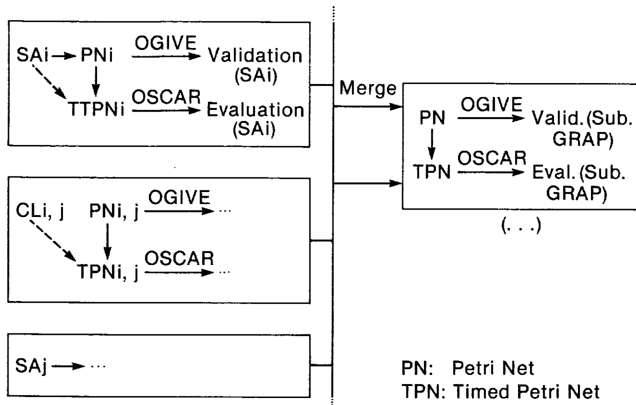


Figure 5—Validation and evaluation process

Organic phase

At the end of the functional phase the leaves of the functional tree have to be *described.* The description includes two separate aspects: (1) sequential algorithms (SA) and (2) communication links (CL). At this level we must explain how the system works. The first step is to transform our functional description into an implementation description by means of the following entities: (1) processes (sequential), (2) channels, (3) clocks, and (4) nets.

Each process communicates with the other processes by exchanging messages via channels. One process *cannot* know which other process is going to consume the message it is sending. We suppose, first, that each process is implemented on a specific immaterial processor (we know, approximately, the required performances of this processor, issued from the functional model of the process it is supposed to run). So we get a theoretical distributed architecture in which we specify the communications, CL, by means of a meta-language PAMELA;[13] we obtain the global structure of the architecture, the skeleton, in terms of SA and CL. The dynamic data are transformed into messages going through the CL, and the static ones are distributed with the processes to which they are related. The internal structure of the data is also a tree; each tree is a *type,* which can be seen, more or less, as an OSCAR local class.

Then the sequential processes (SA) are described by means of PAMELA. Mainly, we specify the global structure of the process: i.e., the control structures (IF. . . THEN. . . ELSIF ; LOOP ; EXIT ; . . .), the procedure/action calls, and the declaration of static data. These parametrized macros are connected automatically by OGIVE. The designer specifies time on each transition to get a timed Petri net model; there are two ways for including time.

One can see that the second way of firing a transition complicates very much the study of the net. This work is going on in collaboration with LAAS/CNRS at Toulouse, France.

The goal is the following: once we have obtained a timed Petri net model of a local mechanism, we run it under an operational workload; we compute some "cycle time"[14, 15, 16] and some information queuing model oriented: service discipline, service time, service rate. Then we introduce the Petri net model in a global OSCAR model (hierarchical net) and transform it in a local queue with the parameters computed before it is integrated into the global queuing network. We only model some specific complex mechanisms that we want to validate qualitatively and quantitatively.

This process allows us to choose the best implementation of the entire system. It should be noted that often this approach is oriented by the fact that, at the beginning of the project, the hardware architecture and basic software are imposed by the results of the first investigations. In such cases the main work consists of evaluating the performances of these elements, especially those of the basic software, e.g., path length, interactions with the application, and exceptions.

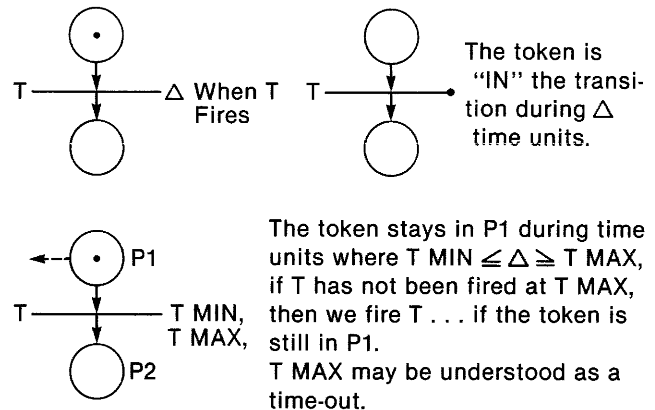


Figure 6—Firing timed petri-nets

CONCLUSION

We have introduced a global design methodology based on concepts such as data flow analysis, sequential processes, communication links, and abstract architecture. These concepts give us a guide for designing real-time systems. Two tools have been also introduced that perform a specific action in the design process: OGIVE, a Petri net analyzer, performs the verification and validation of the abstract structure; and OSCAR, which is analytical-queuing-network-oriented, evaluates quantitatively some implementation choices of the real system. These tools are integrated very early in the design process to insure that we will not have dramatic regressions to make in redesigning the system.

REFERENCES

1. Le Mer, E. "Outil de Simulation pour la Conception d'une Architecture Répartie." *Actes de la Convention Informatique,* Vol. A (1981), pp. 227–230.

2. Baskett F., K. M. Chandi, R. R. Muntz, and F. G. Palacios. "Open, Closed and Mixed Networks of Queues with Different Classes of Customers." *Journal of the ACM,* 22 (1975), pp. 248–260.
3. Reiser, M. "Mean-value Analysis and Convolution Method for Queuing Dependent Servers in Closed Queuing Networks." *Performance Evaluation,* 1, (1981), pp. 7–18.
4. Reiser, M., and S. S. Lavenberg. "Mean-value Analysis of Closed Multichain Queuing Networks" *Journal of the ACM,* 27 (1980), pp. 313–322.
5. Schweitzer, P. "Approximate Analysis of Multiclass Closed Networks of Queues." *International Conference on Stochastic Control and Optimization,* Amsterdam, 1979.
6. Zahorjan, J. "The Approximate Solution of Large Queuing Network Models." Technical Report CSRG-122, University of Toronto, August 1980.
7. Chezalviel-Pradin, B. "Un Outil Graphique Interactif pour la Vérification des Systèmes à Évolution Parallèle Décrits par Réseaux de Pétri." Thèse Docteur-Ingénieur No. 671, University of Toulouse, 1979.
8. Pradin, B., B. Berthomieu, P. Azema, M. Diaz, and S. Bachmann. "OGIVE: Un Outil Graphique Interactif de Vérification de Réseaux de Pétri." *Revue MIKADO,* 35 (1980), pp. 1–10.
9. Peterson, J. L. "Petri Nets." *Computing Surveys,* 9 (1977), pp. 223–252.
10. Woodgate, H. S. "Management of Large Scale Computer Program Production." *AFIPS, Proceedings of the National Computer Conference* (Vol. 46), 1977, pp. 277–283.
11. Putnam, L. H. "Software Costing and Life Cycle Control." *Proceedings of the Workshop on Quantitative Software Models,* 1979, pp. 20–31.
12. Lehman, M. M. "Programs, Life Cycles, and Laws of Software Evolution." *Proceedings of the IEEE,* Col. 68, No. 9 (1980), pp. 1060–1076.
13. Lalanne, R. "PAMELA: Meta-language de Spécification et Processeur d'Analyse." Technical Report ECA AUTOMATION, 1980.
14. Ramamoorthy, C. V., and G. S. Ho. "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets." *IEEE Transactions on Software Engineering,* Volume SE6 (1980), pp. 440–449.
15. Ramchandani, C. "Analysis of Asynchronous Concurrent Systems by Petri Nets." Project MAC, Technical Report 120, Massachusetts Institute of Technology, 1974.
16. Zuberek, W. M. "Analysis of the Effectiveness of Central Processing Units." *Groupe de réflexion sur le temps réel THOMSON-CSF,* Thomson-CSF, Orsay, France, May 1980.
17. Laurent, M. "Les Algorithmes d'OSCAR: NCA et MVA Multichaînes" Technical Report, ECA Automation, 1981.
18. Memmi, G. "Méthodologie d'Analyse et de Programmation des Systèmes; Outils d'Evaluation. Rapport D.A.I.I. No. 79.35.059, February 1981.
19. Boussinot, F. "Résaux de Processus Avec Mélange Equitable: Une Approche du Temps Réel." Thèse d-état Université Paris VII, 1981.

APPENDIX—NCA multichain and an academic OSCAR example[17]

We first introduce the following notations:

$\vec{K} = (K_1, \ldots, K_s)$ if s is the number of chains; state vector of Q (K) network of queues

$Q^{[i]}$ The network issuing of Q without queue i

$g(\vec{K})$ Normalizing constant

$P_i(\vec{j}, \vec{k})$ Steady-state probability of the event: j customers at queue i when there are k customers in the network

$\theta_{i,r}$ Visit rate at queue i for class r

$t_i(\vec{k})$ Mean-waiting time at queue i

$S_{i,r}$ Service time of queue i for class r

$n_i(\vec{k})$ Queue length at queue i for a k population in the network

$\Lambda_{i,r}(\vec{k})$ Throughput of queue i for class r

$\mu_{i(j)}$ Service rate of queue i when j jobs are waiting in the queue.

As an extension of NCA monochain we compute

$$P_N(\vec{j}, \vec{k}) = \Pi_N(\vec{j})\, g^{[N]}(\vec{k} - \vec{j}) / g(\vec{k}) \tag{1}$$

where

$g^{[N]}(\vec{k})$ is the normalizing constant for network $Q^{[N]}$

and $\Pi_N(\vec{j}) \dfrac{({}^{W}N, 1)^{j1} \times \ldots \times ({}^{W}N,r)^{jr}}{{}^{\mu}N(1) \times \ldots \times {}^{\mu}N(|\vec{j}|)} \dfrac{(|\vec{j}|)!}{j_1! j_2! \ldots j_r!}$

with

$$|\vec{j}| = j_1 + \ldots + j_r \text{ if } \vec{j} = (j_1, \ldots j_r)$$

and

$$g(\vec{k}) = \sum_{F(\vec{k})^i} \Pi_i(\vec{k}_1)$$

with

$$F(\vec{k}) = \left\{ (\vec{k}_1, \ldots, \vec{k}_N) \,/\, |\vec{k}_i| \geq 0 \text{ and } \sum_{1=i}^{N} \vec{k}_i = \vec{k} \right\}$$

then, from equation (1) we get

$$P_N(\vec{O}, \vec{k}) = g^{[N]}(\vec{k}) / g(\vec{k})$$

by induction we get the following recurrent equation:

$$P_N(j, \vec{k}) = \begin{cases} \sum_{s=1}^{r} p_N(j-1, \vec{k} - \vec{e}_s)\, W_{N,s}\, \Lambda_s(\vec{k}) / \mu_N(j) \\ \text{if } j \geq 1 \\ \text{if in direction } \vec{e}_s \text{ the coordinate of } \vec{k} \\ \text{is nul} \\ \text{then } p_N(j-1, \vec{k} - \vec{e}_s) = 0 \end{cases}$$

using the relation on the throughput

$$\Lambda_s(\vec{k}) = g(\vec{k} - \vec{e}_s) / g(\vec{k}) \tag{3}$$

We get

$$p_N(j,\vec{k}) = \sum_{s=1}^{r} p_N(j-1, \vec{k} - \vec{e}_s)\, g(\vec{k} - \vec{e}_s)\, W_{N,s} / g(\vec{k})\, \mu_N(j)$$

If we set

$$p'_N(j,\vec{k}) = p_N(j,\vec{k})\, g(\vec{k})$$

we have

$$p'_N(j,\vec{k}) = \sum_{s=1}^{r} p_N(j-1, \vec{k} - \vec{e}_s)\, g(\vec{k} - \vec{e}_s)\, W_{N,s} / \mu_N(j) \tag{4}$$

$$\sum_{j=0}^{|\vec{k}|} p'_N(j,\vec{k}) = g(\vec{k})$$

So we have the algorithm NCA multichain:

For a queue N, knowing $g^{[N]}(\vec{t})$; $p(0,\vec{t}) \ldots p(|\vec{t}|,\vec{t})$; $g(\vec{t})$; ... but $g(\vec{t}) = g^{[N+1]}(\vec{t})$ and by the same way it is possible to compute p', g, p for queue $N+1$.

We solve the network by iterating on the queues after the initialization as follows:

$$g^{[2]}(t) = \sum_{F(\vec{t})} \Pi_1(t)$$

$$= \left[({}^{W}1,1)^{t_1} \ldots ({}^{W}1,r)^{t_r} / {}^{\mu}1^{(1)} \cdots {}^{\mu}1^{(|\vec{t}|)} \right] (|\vec{t}|)! / t_1! \ldots t_r!$$

The example is the following (See Figure 7):
There are three chains containing respectively 2, 4, 1 jobs.

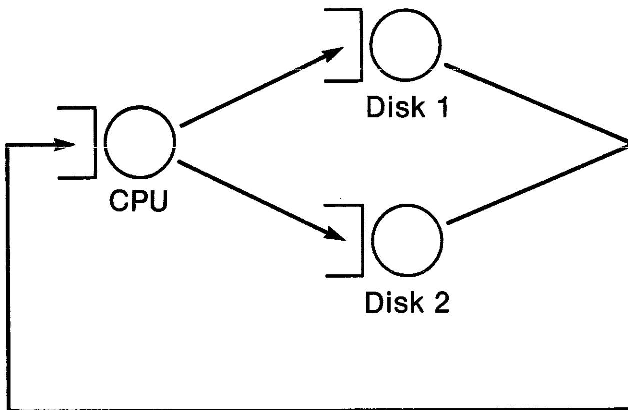


Figure 7—Network CPUIO

The description could be as follows (the underlined words are keywords):

Reseau cpuio (0, 3, 3)
File disk 1, *fifo, active*
nbclass = 2
temps serv = t_1, 1 ; t_2,2
Taux serv = 1
File disk 2
copy disk 1
File cpu, *ps, active*
(. . .)
Chaîne Ch1/ fist chain /
type = *fermée*
charge = 2
cpu, disk 1, 1; disk 1, cpu, 1 ;
chaîne Ch2
(. . .)
Fin reseau

The outputs from MVA analysis are shown in Figure 8.

Specification of bounded channel in L. R. with test on full channel and its translation in Petri net[18, 19]

L. R. is the high-level kernel of PAMELA for specifying the real-time concepts, especially the communication links and fairness; consider the net in Figure 9, which expresses communication by means of a bounded channel with test on full channel (no determinism).
A producer of X sends a message ≠ ≠ 1 when he wants to put an information in X. If it is allowed to (≠ ≠) it sends the message on X_e.
When the producer wants to know the state of X it sends a message ≠ ≠ 2 on X_e and waits for the answer on REP.
The consumer sends message ≠ ≠ 3 on DEM when it wants an information and waits for the information on X_S.
The specification of this net R_X and of the corresponding actions is the following:

```
DEF(&XS,&rep PORTE mes)rx(&xe, 1dem PORTE mes)
RESEAU
  STRUC
  (g&) merge (&xe,&dem)
  (&xs,&rep) g (merge)
  FSTRUC
DEF (&intm PORTE mes) merge (&ei,&e2 PORTE mes)
    CANAL TAILLE NON BORNEE
FDEF
DEF (&s1,&s2 PORTE mes) g (&intg) PORTE mes)
PROCESS
  DEF f VAR TYPE file
    depos,prise CAR BOOLEEN
    z VAR TYPE mes
FDEF

DEF     requete ACTION
       SI p(f) ALORS depos := VRAI SINON
       METTRE(&s2, ≠ ≠ ok) FSI
         test ACTION
       SI p(f) ALORS METTRE(&s2,VRAI) SINON
       METTRE(&s2,FAUX) FSI
         prise ACTION
       SI v(f) ALORS prise : = VRAI SINON (z) get (f);
                METTRE(&sl,z) ;
                SI depos ALORS METTRE
                                          (&s2, ≠ ≠ ok);
                    depos : = FAUX
                PSI
       FSI
         depos ACTION
       SI prise ALORS METTRE(&s2,z); prise : = FAUX
            SINON (f) put (z)
       FSI
FDEF
EXEC
  prise : = FAUX ; depos : = FAUX ;
  BOUCLE
       PRENDRE(&intg,z);
       CAS z VAUT ≠ ≠ 1 FAIRE requete
           VAUT ≠ ≠ 2 FAIRE test
           VAUT ≠ ≠ 3 FAIRE prise
           AUTREMENT depos
       FCAS
    FBOUCLE
  FEXEC
FDEF %g%
FDEF% rx%
```

Then we translate it in a Petri net (see Figure 10) to evaluate the length of the channel "rep" (communication between g and the producer), the channel "XS" (communication between g and the consumer), and the channel "int" (between merge and g)
One can see that this net, modeling R_X, works well (no deadlocks, no starvation) if

length ("rep") = 1
length ("XS") = 1
length ("int") = 3

These results are obtained by OGIVE, using, for instance, the INVARIANT module.

3 STATIONS 3 CHAINES TAUX CONSTANTS

NOMBRE DE FILES 3

NOMBRE DE CHAINES ET REPARTITION DES JOBS
3 2 4 1

DEMANDES DE SERVICE
FILE 1
CHAINE 1 .1000E − 01
CHAINE 2 .1500E − 01
CHAINE 3 .2000E − 01
FILE 2
CHAINE 1 .3000E − 01
CHAINE 2 .0000E + 00
CHAINE 3 .3000E − 01
FILE 3
CHAINE 1 .0000E + 00
CHAINE 2 .3500E − 01
CHAINE 3 .3500E − 01

MVA MULTICHAINES

NO RESSOURCE	TAUX UTILISATION	LONGUEUR FILE	TEMPS DE REPONSE	DEBIT
RESSOURCE 1	.7389E + 00	.2018E + 01	.3638E − 01	.5548E + 02
(CHAINE 1)	.2499E + 00	.6468E + 00	.2588E − 01	.2499E + 02
(CHAINE 2)	.3625E + 00	.1039E + 01	.4297E − 01	.2417E + 02
(CHAINE 3)	.1265E + 00	.3330E + 00	.5266E − 01	.6323E + 01
RESSOURCE 2	.8445E + 00	.1579E + 01	.5607E − 01	.2815E + 02
(CHAINE 1)	.7497E + 00	.1353E + 01	.5415E − 01	.2499E + 02
(CHAINE 2)	.0000E + 00	.0000E + 00	.0000E + 00	.0000E + 00
(CHAINE 3)	.9485E − 01	.2253E + 00	.7126E − 01	.3162E + 01
RESSOURCE 3	.9566E + 00	.3403E + 01	.1245E + 00	.2733E + 02
(CHAINE 1)	.0000E + 00	.0000E + 00	.0000E + 00	.0000E + 00
(CHAINE 2)	.8459E + 00	.2961E + 01	.1225E + 00	.2417E + 02
(CHAINE 3)	.1107E + 00	.4417E + 00	.1397E + 00	.3162E + 01

Figure 8—Outputs from MVA analysis

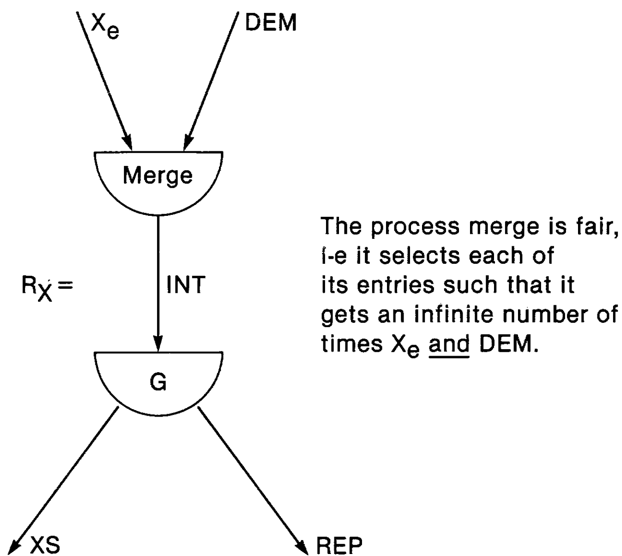


Figure 9—Bounded channel

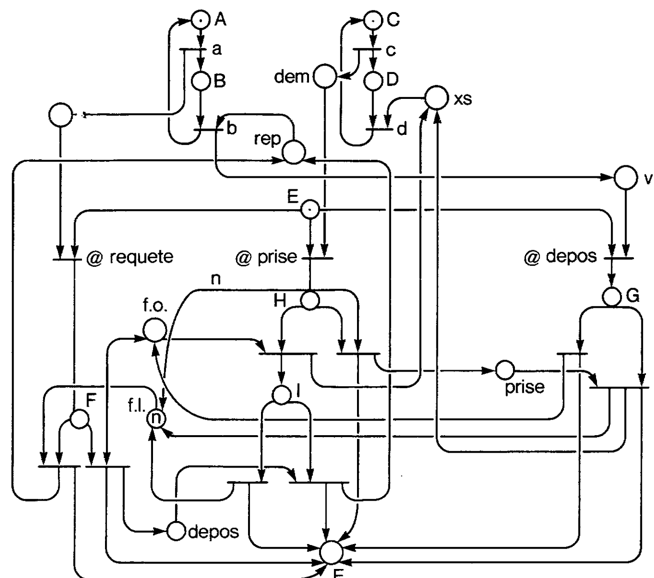


Figure 10—Bounded channel model

The research queueing package: past, present, and future

by CHARLES H. SAUER, EDWARD A. MACNAIR, and JAMES F. KUROSE
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT

Queueing networks are important as performance models of systems where performance is principally affected by contention for resources. Such systems include computer systems, communication networks, office systems and manufacturing lines. In order to effectively use queueing networks as performance models, appropriate software is necessary for definition of the networks to be solved, for solution of the networks (by numerical, approximate and/or simulation methods) and for examination of the performance measures obtained. One of the most widely known and influential pieces of queueing network software is the Research Queueing Package (RESQ). This paper discusses the evolution of RESQ and plans for further RESQ development.

INTRODUCTION

Many physical systems, including computing systems, communication networks, office systems, and manufacturing lines, are heavily dependent on sharing of resources. Sharing of resources necessarily leads to contention, i.e., queueing, for resources. Contention and queueing for resources are typically very difficult to quantify when estimating system performance.

Queueing models have been used for decades in studying the performance of manufacturing lines, communication networks, and similar systems. In the last two decades, queueing models have become important as performance models of computing systems. Since office systems have become heavily dependent on computing and communication, queueing models are appropriate in office system performance evaluation. These models are often networks of queues because of the interactions of system resources. For a general discussion of queueing network models, see Sauer and Chandy[1] and recent special issues of *Computing Surveys* (September 1978) and *Computer* (April 1980).

For queueing network models to be used effectively, appropriate software is necessary for constructing models and obtaining solutions for models. One of the most widely known and influential pieces of queueing network software is the Research Queueing Package (RESQ).[2–5] Other pieces of software influenced by RESQ include the Queueing Network Analysis Package (QNAP)[6] and the Performance Analyst's Workbench System (PAWS).[7]

RESQ is important and influential because of (1) the "extended" queueing networks associated with RESQ, (2) the diagram language used to informally represent queueing networks to be handled by RESQ, (3) the user language and machine interfaces used to formally represent queueing networks and their solutions, and (4) the multiple-solution methods of RESQ, including the research effort that has gone into their design and implementation. This paper discusses these points from a historical viewpoint and discusses the expected future evolution of RESQ.

QUEUEING NETWORK MODELS

The following discussion will primarily use computing system terminology and assume that the reader can provide the analogous terminology for other systems. A typical queueing network model consists of a set of queues (corresponding to resources in a computer system) and a set of jobs (which correspond to processes in a computer system, users at terminals, messages sent from computer to computer, etc., depending on the system). The individual queues are usually described in terms of types of resources, numbers of units of resources, queueing (scheduling) disciplines, and probability distributions for the service times of jobs at the queues. The jobs are described by their individual characteristics, by their routing from queue to queue (corresponding to the sequence of resource requirements in the system), and by their arrival processes (and departure procedures).

Much of the research on queueing network models has focused on methods for obtaining solutions, i.e., performance estimates, for the models. Efficient numerical algorithms have been developed for networks with a *product form* solution.[1,8–12] However, there are many system characteristics that preclude a product form solution—e.g., priority scheduling or simultaneous resource possession. For models with these characteristics and more than a few queues and/or jobs, the only solution methods available are approximate numerical methods[1,13,14] and simulation. Specialized simulation techniques have been developed that apply to simulation of queueing networks.[1,15]

RESQ incorporates both numerical and simulation solution methods. Though RESQ includes simulation components, we do not consider RESQ to be a *simulation* language, but a *modeling* language. We make the distinction primarily because of the higher level of abstraction of RESQ elements, as compared to popular simulation languages, and also because of the numerical (nonsimulation) solution methods provided in RESQ.

EXTENDED QUEUEING NETWORKS

To facilitate more accurate representation of systems, the queueing networks of RESQ have been designed to include and naturally build upon the category of networks with product form solution. Some of the elements are obvious generalizations of product form elements; for example, queues with general (e.g., priority) scheduling disciplines. Other generalizations of product form networks include (1) capabilities for marking jobs with information (such as message length for a job representing a message in a communication network) and (2) routing rules dependent on the current network state (e.g., queue lengths) as well as the usual probabilistic routing rules.

In addition to allowing the characteristics described above, which usually violate product form solution conditions, we provide in RESQ new network elements and refer to the resulting category of networks as *extended* queueing networks.[16] We restrict attention to the most important of these elements, the *passive queue* (Figure 1). We refer to traditional queues as *active queues*. One of the limitations of a network consisting only of active queues is that a job can hold only one resource at a time. This can be a severe restriction in studying systems in which a job requires several resources simulta-

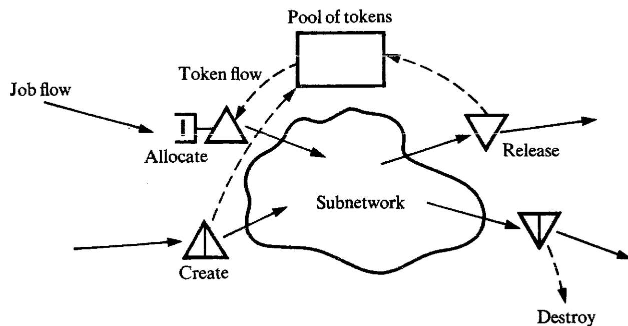


Figure 1—A passive queue

neously. For example, a program requires memory as well as a CPU before it can be run, but most traditional queueing models will ignore either memory contention or CPU contention. In extended queueing networks a job can hold resources at several passive queues and one active queue simultaneously.

A passive queue consists of a set of *allocate nodes,* a set of *release nodes,* a set of *create nodes,* a set of *destroy nodes,* and a pool of identical *tokens* of a resource. A job joins a passive queue when it arrives at an allocate node. Upon arrival the job requests one or more tokens. If sufficient tokens are available, the requested number of tokens is allocated to the job, which then moves on to another queue of the network without delay. However, the job belongs to the queue from which it received the tokens as long as it holds the tokens. If insufficient tokens are available, the job waits until enough become available and then immediately moves on through the network after receiving them. When several jobs wait for tokens of a passive queue, they are allocated tokens according to a specified scheduling discipline. A job gives up tokens, and thus leaves the corresponding passive queue, when it is routed through a release node of the queue. The job passes through the release node instantaneously. Create nodes have no effect on the job; jobs passing through a create node simply add new tokens to the pool. Destroy nodes are similar to release nodes but do not return the tokens to the pool. Jobs pass instantaneously through create and destroy nodes. (See Figure 1.)

The terms *active queue* and *passive queue* are intended to indicate the nature of the queue's effect on a job's use of a server or token, respectively, and of the relative dominance of the modeled resources. With an active queue the length of time a job holds a server is entirely determined by the characteristics of that queue and the jobs at that queue. With a passive queue the length of time a job holds a token is determined entirely by events at other queues.

Figure 2 shows a simplistic representation of a widely used model of interactive computer systems. The resources represented by active queues are the terminals, CPU and I/O device(s). A passive queue is used to represent memory contention. After a think time at the terminal, a user keys in a command. A job representing the process executing the command requests memory. After receiving memory, the job alternates between CPU and I/O activities until the command is finished. The job then releases its memory and returns to the terminals queue for another thinking and keying time. For other examples of passive queues and extended queueing networks, see Sauer and MacNair[16] and Sauer.[17]

RESQ HISTORY

The original solution components of RESQ, QNET4 (numerical solution), and APLOMB (simulation) were separately developed in 1974 by M. Reiser at the IBM Thomas J. Watson Research Center and C. H. Sauer at the University of Texas, respectively.

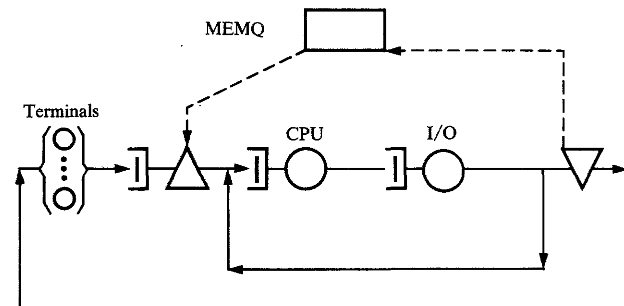


Figure 2—Computer system model

QNET4 was initially implemented in APL and subsequently reimplemented in PL/I. Though QNET4 is essentially unchanged, it represents the state of the art of the "convolution" algorithm it uses. (The Research Queueing Package Version 2 (RESQ2) provides an alternate computational algorithm, Mean Value Analysis,[10] for the same class of networks handled by QNET4.)

APLOMB was initially implemented in Fortran. Two special features of APLOMB are (1) the use of extended queueing networks (including passive queues) used to represent models and (2) the provision of statistical output analysis techniques (including confidence interval estimation and stopping rules). APLOMB has been (and is being) continually revised and improved over the years. In late 1976 APLOMB was translated from Fortran to PL/I.

In the spring of 1976, QNET4 and APLOMB were provided with a common user interface implemented by E. A. MacNair. The three programs became collectively known as RESQ. This prototype version of RESQ used the APL QNET4 with the interface implemented in APL.[2,3] In the spring of 1977, a new version of RESQ, "RESQ1," was developed. RESQ1 was implemented entirely in PL/I, though some components were duplicated in APL for users who preferred that environment to CMS or TSO.

In the summer of 1978, an entirely new user interface was designed to overcome a number of fundamental limitations of the original interface. This new version is known as "RESQ2." It became operational in July 1980.

TECHNICAL CONTRIBUTIONS OF RESQ1

Two of the primary technical contributions of RESQ1 are the extended queueing networks and the diagram language for

describing extended queueing networks. The extensions, especially passive queues, make queueing networks a powerful framework for abstracting the essential characteristics of systems' performance. The diagram language provides a concise means of describing systems, even when actually constructing a model with RESQ is not contemplated. The extended queueing networks and diagram language have had a strong influence outside of IBM, e.g., have influenced software packages such as QNAP[6] and PAWS.[7] Though RESQ2 is a much more powerful tool than RESQ1, there have been only relatively minor additions needed in the extended queueing networks of RESQ1 in the development of RESQ2.

Besides the extended queueing networks and diagram language, the focus of RESQ1 development was the simulation portion, APLOMB. The numerical solution portion, QNET4, remained essentially unchanged from its state before RESQ1 (and remains essentially unchanged today). However, since APLOMB provides the simulation capability needed to solve extended queueing networks, APLOMB continued to evolve as the queueing network extensions were developed. APLOMB and the extended queueing networks were also important in that they provided an impressive demonstration that the regenerative method for confidence intervals,[15] new in the literature at that time, had practical application far beyond the "toy" applications in the literature. APLOMB also included a sequential stopping rule for determining simulation run length.[18]

The greatest failing of RESQ1 was the rigid language used to formally define and describe the queueing models. Little attention was given to this language, though much effort went into the interactive implementation of the language. An implicit assumption in the language design was that models constructed with RESQ1 would be small in terms of numbers of elements and the language could thus be designed for implementation convenience and efficiency, rather than user convenience and efficiency. Because of this rigidity, the interactive interface (and its subsequent "dialogue file" mode) were inconvenient, at best, for the large system models made attractive by the extended queueing networks and APLOMB.

RESQ2 DESIGN AND DEVELOPMENT

RESQ2 Language Design

The objective in the language design was to provide a language similar in appearance to the RESQ1 dialogues[16] but providing the features and flexibility of a modern programming language. (Block structured programming languages, in particular Pascal, were especially influential, though this influence is not obvious in the actual syntax.)

Some of the changes from the RESQ1 language to the RESQ2 language are simple, yet important, improvements which corrected some of the deficiencies of the RESQ1 language. For example, the RESQ1 dialogues require that queues (and other network elements) be numbered sequentially and referenced by these numbers. The RESQ2 language allows symbolic naming of elements. The RESQ1 dialogues generally allow only numeric constants where numeric values are required. The RESQ2 language allows arbitrary numerical expressions in such places. These expressions may include symbols previously defined to have constant values, symbols representing parameters to be defined by the user before solution begins, and symbols representing values which may vary during a simulation. The RESQ1 dialogues require that the number of queues (and numbers of other elements) be specified at the beginning, forcing the user to make a count and stick to it. The RESQ2 language avoids all such requirements.

In addition to these changes, the RESQ2 language provides two kinds of "templates" (macro-like constructs) which greatly enhance its power. The use of templates makes it possible to describe networks in a much more "structured" manner (in the sense of structured programming) and to sharply reduce the effort required to construct models. One kind of template, the "queue type," provides the ability to create parameterized definitions of queues. Once a queue type has been defined, it can be repeatedly used (invoked) to define specific instances of queues. Queues defined using queue types have default characteristics specified in the queue type definition; other queue characteristics are specified by parameter values given with the queue type invocation.

The other kind of template, the "submodel," allows definition of a parameterized template of an entire subnetwork, which may be used repeatedly in defining a network. Previous work on programming languages provided little guidance on how such subnetworks should be specified and interfaced with the remainder of a network. Following is a submodel definition for part of the network of Figure 2:

```
SUBMODEL:cssm /*Computer System SubModel*/
  NUMERIC PARAMETERS:pageframes
  DISTRIBUTION PARAMETERS:disktime cputime
  CHAIN PARAMETERS:chn
  NUMERIC IDENTIFIERS:cpiocycles
    CPIOCYCLES:8
  QUEUE:diskq
    TYPE:fcfs
    CLASS LIST:disk
      SERVICE TIMES:disktime
  QUEUE:cpuq
    TYPE:ps /*Processor Sharing*/
    CLASS LIST:cpu
      SERVICE TIMES:cputime
  QUEUE:memory
    TYPE:passive
    TOKENS:pageframes
    DSPL:fcfs
    ALLOCATE NODE LIST:getmemory
      NUMBERS OF TOKENS TO ALLOCATE:
        discrete(16,.25;32,.5;48,.25)
    RELEASE NODE LIST:freememory
  CHAIN:chn
    TYPE:external
    INPUT:getmemory
    OUTPUT:freememory
    :getmemory->cpu->disk
    :disk->freememory cpu; 1/cpiocycles 1-1/cpiocycles

END OF SUBMODEL CSSM
```

and a complete model definition which assumes the submodel has been stored in a library:

```
MODEL:csm /*Computer System Model*/
  METHOD:aplomb
  NUMERIC PARAMETERS:thinktime users pageframes
  NUMERIC IDENTIFIERS:cpiocycles
    CPIOCYCLES:8
  DISTRIBUTION IDENTIFIERS:disktime cputime
    DISKTIME:.019 /*mean of exponential*/
    CPUTIME:standard(.05,5) /*mean and coefficient of
      variation*/
  QUEUE:terminalsq
    TYPE:is
    CLASS LIST:terminals
      SERVICE TIMES:thinktime
  INCLUDE:cssm /*submodel definition from library*/
  INVOCATION:host
    TYPE:cssm
    PAGEFRAMES:pageframes
    DISKTIME:disktime
    CPUTIME:cputime
    CHN:interactiv
  CHAIN:interactiv
    NODE LIST:terminals
    REGEN POP:users
    INIT POP:users
  CONFIDENCE LEVEL:90
  SEQUENTIAL STOPPING RULE:yes
    QUEUES TO BE CHECKED:host memory
      MEASURES:qt
      ALLOWED WIDTHS:10
  SAMPLING PERIOD GUIDELINES-
    QUEUES FOR DEPARTURE COUNTS:host.memory
      DEPARTURES:1000
  LIMIT - CP SECONDS:100
  TRACE:no

END
```

The above are examples of dialogue files, i.e., files similar to the interactive dialogue. Upper case corresponds to prompts in the interactive version, and lower case corresponds to replies in the interactive version. In the true interactive mode, there would be additional prompts of the same form as shown above. These additional prompts would receive no reply from the user, thus indicating the end of a subsection of dialogue. For example, in interactive mode, the actual dialogue corresponding to the above file might be

```
MODEL:csm /*Computer System Model*/
  METHOD:aplomb
  NUMERIC PARAMETERS:thinktime users pageframes
  NUMERIC PARAMETERS: /*Null response*/
  NUMERIC IDENTIFIERS:cpiocycles

...

    TYPE:closed
    POPULATION:users
    :terminals->host.input
    :host.output->terminals
  QUEUES FOR QUEUEING TIME DIST:host.memory
    VALUES:1 2 3 4 5 6 7 8
  CONFIDENCE INTERVAL METHOD:regenerative
  REGENERATION STATE DEFINITION-
  CHAIN:interactiv
```

RESQ2 Translator

Because the RESQ1 "dialogue files" had become the more important mode of usage of RESQ1, and because the severe limitations of the language and translator for the dialogue files were the major motivation for RESQ2, the focus of the language and translator design was the dialogue file. It was clear that a compiler-like program was necessary to support the new language features.

A key design decision was that the compiler-like translator use recursive descent parsing. Recursive descent has two important advantages for our translator over more recent techniques based on parser generators. (1) Recursive descent is much more flexible in terms of error recovery. (2) More importantly, due to the flexibility of recursive descent, it has been possible for the same translator to operate effectively as an interactive prompter. Having the same translator capable of both "batch" and interactive modes has been remarkably useful in model construction because (1) in interactive mode, it is possible to immediately make revisions or corrections to prior dialogue by escaping to an editor to revise a transcript of the dialogue so far (a dialogue file) and to then continue in prompting mode after the (incomplete, edited) dialogue file has been reparsed and (2) revision of an existing model is possible in mixed mode by deleting portions of the existing dialogue file and using interactive mode for specification of revisions or additions. This mixed mode capability provides the "user friendliness" of interactive mode without losing the flexibility and efficiency of "batch" mode for development of significant models.

RESQ2 Expansion Processor

The output of the translator is a highly symbolic form, far removed from the interface expected by the solution components. This is necessarily the case because of the provision of parameters which are left undefined until the model is to be solved. These run-time parameters allow a model to be solved parametrically without retranslation. A hierarchical network definition, with invocations of submodels, cannot be translated into a monolithic network definition until these parameters are specified. Thus a major portion of the RESQ2 implementation has been the "expansion processor," which produces a network definition at the solution component interface from the symbolic translator output. (The term "expansion" is consistent with the analogy between submodels and macros.)

RESQ2 Solution Methods

An implementation of Mean Value Analysis is becoming the dominant numerical solution component of RESQ2. (Mean Value Analysis and the Convolution algorithm of QNET4 both handle the full class of product form networks.[12] Each has advantages over the other.) A major aspect of the evolution of APLOMB has been a gradual redesign of the data structures and rewriting of the code to get away from APLOMB's Fortran heritage. These efforts were critically necessary to obtain the efficiency (both storage and run time) and flexibility needed for RESQ2. Additional extensions to APLOMB were needed to support RESQ2 language features. These extensions include simulation time expression/symbol evaluation and submodel support. Though submodels are nominally hidden from the solution components, submodels must be considered in simulation error messages, trace output and evaluation of expressions which involve submodels.

Other extensions to APLOMB are relatively independent of the RESQ2 language. A major area of improvement in APLOMB is in its output analysis capabilities. Confidence intervals obtained by the classical method of independent replications[1] have been added as an alternative to the regenerative method for models where the regenerative method is not practical or appropriate. The regenerative method implementation has been made more rigorous in its determination of regeneration states. The sequential stopping rule has been refined and made more flexible.

A major new feature of APLOMB is an interactive simulation capability. It is now convenient to continue a simulation run after examining results, either because one wants to see results at intermediate points in the run or because one is not satisified with results at the planned run length or stopping condition.

RESQ2 PLANS

A number of RESQ2 features remain to be implemented. Some of these are parts of the original design, while others have been added to our plans more recently. The most important of these features is the "substitution" (hierarchical/hybrid solution) form of invocation of submodels. Substitutions have the potential of greatly reducing the computational expense of model solution, especially where simulation is involved. A hierarchical solution facility such as this is the best hope for making practical the simulation of very large systems. Since there has been very little work in this area outside of a few feasibility studies,[19,20,21] the substitution design will continue to evolve after we gain experience with it.

Another new feature will be the addition of the spectral method for confidence intervals.[22] The spectral method provides another practical alternative to the regenerative method for situations where the classical method of independent replications is inappropriate.

Finally, some of our most ambitious plans are in terms of graphics capabilities for RESQ. For some time we have had the ability to produce high quality diagrams of extended queueing networks on graphics devices. We have recently added the ability to produce diagrams using the output of the RESQ2 translator as input to the graphics programs. This is of great benefit in documenting and debugging models. We have also begun work on constructing models directly by drawing a diagram with a light pen and graphics display. Eventually, this may be sufficient to dramatically reduce the need for typed input. In order to achieve maximum usability, all of the graphics facilities are intended to be usable on low resolution devices, even though a higher resolution device is needed to obtain aesthetically pleasing results.

ACKNOWLEDGMENT

We are grateful to E. Jaffe, P. Rosenfeld, M. Reiser, S. Salza, and S. Tucci for their contributions to RESQ.

REFERENCES

1. Sauer, C. H., and K. M. Chandy. *Computer Systems Performance Modeling.* Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
2. Sauer, C. H., M. Reiser, and E. A. MacNair. "RESQ—A Package for Solution of Generalized Queueing Networks." *AFIPS, Proceedings of the National Computer Conference* (Vol. 46), 1977.
3. Reiser, M., and C. H. Sauer. "Queueing Network Models: Methods of Solution and their Program Implementation." In K. M. Chandy and R. T. Yeh (editors), *Current Trends in Programming Methodology, Volume III: Software Modeling and Its Impact on Performance.* Englewood Cliffs, New Jersey: Prentice-Hall, 1978, pp. 115–167.
4. Sauer, C. H., and E. A. MacNair. "Queueing Network Software for Systems Modeling." *Software-Practice and Experience,* 9 (1979), pp. 369–380.
5. Sauer, C. H., E. A. MacNair, and S. Salza. "A Language for Extended Queueing Networks." *IBM Journal of Research and Development 24* (1980), pp.
6. Merle, D., D. Potier, and M. Veran. "A Tool for Computer System Performance Analysis." In D. Ferrari (editor), *Performance of Computer Installations.* Amsterdam: North-Holland, 1978.
7. Chandy, K. M., J. Misra, R. Berry, and D. Neuse. "Simulation Tools in Performance Evaluation." *Computer Performance Evaluation Users Group Conference 1981,* San Antonio, Texas, November 1981.
8. Jackson, J. R. "Jobshop-like Queueing Systems." *Management Science* 10 (1963), 131–142.
9. Buzen, J. P. *Queueing Network Models of Multiprogramming.* Ph.D. Thesis, Harvard University, Cambridge, Massachusetts, 1971. New York: Garland Publishing, 1980.
10. Reiser, M., and S. S. Lavenberg. "Mean Value Analysis of Closed Multichain Queueing Networks." IBM Research Report RC-7023, IBM, Yorktown Heights, New York, March 1978. *Journal of the ACM,* 27 (1980), pp. 313–322.
11. Chandy, K. M., and C. H. Sauer. "Computational Algorithms for Product Form Queueing Networks." IBM Research Report RC-7950, IBM, Yorktown Heights, New York, November 1979. *Communications of the ACM,* 23 (1980), pp.
12. Sauer, C. H. "Computational Algorithms for State-Dependent Queueing Networks." IBM Research Report RC-8698, IBM, Yorktown Heights, New York, February 1981.
13. Chandy, K. M., and C. H. Sauer. "Approximate Methods for Analysis of Queueing Network Models of Computer Systems." *Computing Surveys,* 10 (1978), pp. 263–280.
14. Sauer, C. H., and K. M. Chandy. "Approximate Solutions of Queueing Models of Computer Systems." IBM Research Report RC-7785, IBM, Yorktown Heights, New York, July 1979. *Computer,* 13 (1980), pp. 25–32.
15. Iglehart, D. L. "The Regenerative Method for Simulation Analysis." In K. M. Chandy and R. T. Yeh (editors), *Current Trends in Programming Methodology, Volume III: Software Modeling and Its Impact on Performance.* Englewood Cliffs, New Jersey: Prentice-Hall (1978).
16. Sauer, C. H., and E. A. MacNair. "Computer/Communication System Modeling with Extended Queueing Networks." IBM Research Report RC-6654, IBM, Yorktown Heights, New York, July 1977.
17. Sauer, C. H. "Passive Queue Models of Computer Networks." *Computer Networking Symposium,* Gaithersburg, Maryland, December 1978.

18. Lavenberg, S. S., and C. H. Sauer. "Sequential Stopping Rules for the Regenerative Method of Simulation." *IBM Journal of Research and Development,* 21 (1977), pp. 545–558.
19. Sauer, C. H., L. S. Woo, and W. Chang. "Hybrid Analysis/Simulation: Distributed Networks." IBM Research Report RC-6341, IBM, Yorktown Heights, New York, June 1976.
20. Schwetman, H. D. "Hybrid Simulation Models of Computer Systems." *Communications of the ACM,* 21 (1978), pp. 718–723.
21. Chiu, W. W., and W-M Chow. "A Performance Model of MVS." *IBM Systems Journal,* 17 (1978), pp. 444–462.
22. Heidelberger, P., and P. D. Welch. "A Spectral Method for Confidence Interval Generation and Run Length Control in Simulation." IBM Research Report RC-8264, IBM, Yorktown Heights, New York, 1980. *Communications of the ACM,* 24 (1981).

Audience identification for end user documentation

by JANIS G. RAYMOND
Michigan Wisconsin Pipe Line Company
Houston, Texas

ABSTRACT

Audience identification in writing is comparable to the analysis phase of systems development. Understanding who the users will be, what their business function is and how it relates to the computer system, and what the users need to know is the only way to write effective user documentation. Identifying the users determines the type and amount of detail to include as well as the format, tone, and level of the documentation. Audience identification is the planning phase of the documentation and as such controls the entire writing process. This paper presents guidelines for identifying end users and directing the documentation to their needs.

INTRODUCTION

Companies are just starting to recognize the importance of effective end user documentation. This recognition is based on two tangible benefits:

1. Good user documentation protects the investment in development costs by providing an understanding of what a system can do and how the system can be used.
2. Good user documentation makes the system more efficient by providing the information required for using the system.

With companies scrutinizing system development for a return on investment, it becomes imperative that each step in the development process be evaluated.

End user documentation is the final product, the last deliverable, in the system development process. Without documentation, a system is useless. That is a harsh statement, but true. Systems are developed to help users in their jobs. Without users there would be no system requirements and therefore no systems. Thus, the only way to make a system a profitable investment is to make the system usable. Effective end user documentation is the answer. And the first step in producing effective end user documentation is identifying the audience. Establishing who the users are and what their needs are controls the entire documentation process.

WHY IDENTIFYING THE USER IS IMPORTANT

Identifying the user before you begin writing ensures that the documentation meets the users' needs. Understanding who the users are, what business function the system performs, and what the users know and need to know allows you to write documentation that will help the users.

Typically, users are less knowledgeable about computers than the person writing the documentation. Since the purpose of user documentation is to describe the system functions in terms of the users' jobs, the writer must step away from computerese and present the information in terms that are relevant to the users. This may mean more work up front in the writing process, but ultimately it will save both the writer and the user time and frustration.

Users often see computers simply as a means to an end. They don't want to be burdened with endless computer jargon in trying to decide how to request a report. Nor do they want to wade through pages of paper looking for a simple instruction. Thus the language used and the organization of the documentation should be two primary considerations.

When daily use of a system seems more a problem than an aid, users lose interest in the system and become advocates of the "I can do it easier myself" school of thought. Documentation written from the users' perspective, in terms users understand, is the only way to make the documentation effective and the system worth the development time and costs.

WHO IS THE USER?

For such a simple question, the answer is seldom straightforward. It would be nice if the writer could quickly identify the user as Person X. Furthermore, if you knew the position Person X held in which department, and if you knew that Person X was responsible for Business Function A, you would have a relatively good understanding of the user.

Unfortunately, identifying the user is rarely that easy. Typically, the user is a department or a group of departments in the organization. To have a good understanding of the business functions involved, you must first find out how this department fits into the overall organization. Is there a central user department, but are there also decentralized departments in other areas of the company? If so, it is important to determine the needs of each of these departments to decide whether the same documentation will serve each group.

Once you understand what department or departments you are dealing with, you must become familiar with the internal structure of each group. You must determine who in the department will be using the system. Will only the clerical staff be preparing input data? Or will engineers and geologists as well as the management staff be involved?

It is also important to establish whether there is sensitive information that should not be included in the documentation. Additionally, find out whether authority is required before certain functions of the system can be performed. If either of these situations exist, you may need to have limited distribution of certain sections of the final documentation. Often the final user manual will be sets of documentation that when combined document the entire system.

Understanding these aspects is your first step in identifying the user. This knowledge gives you the broadest concept of who the user is and ensures that the documentation both includes and excludes the appropriate information. This knowledge also gives you the first clue to the organization of the overall user manual.

WHAT DOES THE USER KNOW?

For the documentation to address the specific needs of each user group, the second step is to determine what the users know. This knowledge will direct how much detail is included.

It is important during this step to keep in mind the purpose

of user documentation. Technically, or from a business point of view, you must assume that users know their jobs. A system's user documentation is not intended to be an overall on-the-job training manual. User documentation should explain how to use a system to aid the users in their jobs. The key word here is "aid." If accountants are trying to use a system to produce payroll checks, you have to assume they know the details involved, like gross pay, taxes, and insurance deductions. This does not mean that the documentation should be so brief that users can't decide how the system relates to their business functions. Therefore, a certain amount of overlap will be required to explain what business function is involved and how to use the computer to perform that function.

To do this, besides understanding the work the users do and how the system fits into their work flow, you must also know the educational levels of the users and the language or terminology they use. Certain words, like *field, element, table,* or *key,* that are common to computer personnel mean something entirely different in the user world. Therefore, to write the documentation in terms users understand, you must exercise caution in your choice of words. This is particularly true if the users are novices at the computer game. Too much computer jargon will not only confuse, but also alienate, the users.

Finding out what the users know requires interviewing and working with them. Writing the user manual from a computer person's perspective will not accomplish the purpose intended.

WHAT DOES THE USER NEED TO KNOW?

To use a system, users need to know answers to these questions:

1. What is the system designed to do?
2. How do they get data into the system?
3. What can they expect out of the system?

These, too, may sound like easy questions; but writing a user manual that answers these questions is not so simple.

Defining the System's Purpose

Telling the users what the system does may be the easiest part of the documentation to write. Assuming that the system does what it is supposed to do, you may be able to use the requirements document to help you write about the system's purpose.

If the system is multipurpose (like a database designed for use by several departments), your documentation will be set up in modules. In these cases an overview stating the purpose of the entire system is needed, as well as an overview for each module defining the system's purpose for each department.

A system overview does not have to be a lengthy explanation of every function performed, but it should provide enough information to allow users to determine whether the system is designed to perform their application. A purpose such as, "System X is a database system designed to provide the user with standardized reports" tells the user nothing. That statement could be the purpose of any system.

So, in the overview, give the users sufficient information to determine what business functions the system performs and to decide whether they can use the system to accomplish what they need to do. Write the overview in users' terms and from their perspective.

Getting Data into the System

Don't shortchange users when describing how to enter data. If the system is executed in batch mode, explain in detail how the data are entered. Do the users fill out forms which are submitted to data entry for keypunching or keying? If so, what forms are required (including batch control forms) and what is entered on each form?

If the system is executed in interactive mode, do the users know how to access screens, page forward, and correct errors as they occur?

Are data elements required, optional, dependent on the presence of other data? Are there minimum and maximum limits on data elements? For certain types of data, you need to give instructions on the units expected: Are volumes handled by the programs as MCF per day or MMCF per month? Are rates expected as cents or dollars? The users should be informed if data are converted from one unit into another within the system, since their output may be affected.

If the system is a database, the users will also need to know how to correct stored data or remove data. Are transaction reports stating acceptance or rejection of the data formatted so they are easily understood by the users? If not, the documentation should include a description of how to read the transaction report.

In general, while considering the system's input data, try to presuppose all questions a user might have and plan to answer those questions in the user manual. Additionally, plan a good format for describing the input. Write the input instructions in active voice and make the instructions easy to read.

Getting Data Out of the System

Users also need to know what reports to expect as output. Will the reports be generated automatically or must they be requested? If the reports are requested, what is the procedure for requesting them? Another form?

Think about what data are shown on the output reports. Some of the data will be a regurgitation of the data entered by the user; but what about the calculated items? Some users may want the calculations used within the system to be included in their documentation. A code listing will not suffice for this unless your users know how to read COBOL, FORTRAN, or whatever language has been used in the programming. Just as the calculations were translated into a programming language for the system, calculations presented in the user documentation should be retranslated into normal mathematical equations.

What about the error messages? "GETD FAILED

WHERE NO PARENT RG EXISTS" is not self-explanatory to users who are unfamiliar with databases. If it has not been done before, you should review error messages while you are writing the documentation. Even if the error message is straightforward, is it clear what the users should do to correct the problem? A section on error-handling procedures may be needed in the documentation.

Detailed information on a system's reports is essential if a user is to understand what the system does. Knowing what functions the system performs and how to enter data will not help the users if they can't understand what their reports indicate.

SUMMARY

Identifying the users is really just a series of questions—questions you ask yourself and questions you ask the users. Your goal is to make the documentation thorough, yet simple and easy to understand. Just like developing the system, writing the documentation requires planning, analysis, testing, and review. Months of work go into developing a system to make the users' jobs easier and more efficient. Don't throw those months away with poorly written user documentation. A system that is not understood cannot be used. Make your systems usable with well-written end user documentation.

Computer-aided documentation

by SAUL ROSENBERG
Riverside Research Institute
New York, New York

ABSTRACT

Current standards for high-quality documentation of complex computer systems include many criteria, based on the application and user levels. Important points common to many systems are: targeting to specific user groups; being complete, concise, and structured; containing both tutorials and reference material; being field-tested; and being timely in appearence relative to the software delivery. To achieve these goals, uniform quality standards should be more vigorously applied, the documentation development cycle should be shortened, more documentation/software help should be available on line, and more user interaction should be solicited.

For future computer systems, the proposal is made that the documentation be "machine comprehensible." This should be phased in, with the immediate goal being to facilitate user querying for information, and with an ultimate goal of providing a database for "programmer apprentice" artificial-intelligence programs that assist software development. This new functionality will be the result of several trends, including the drastically reduced cost of read-only online random-access storage via video optical disks, the ongoing successes of artificial-intelligence programs when applied to limited application areas, and the ever increasing cost of software programmers.

INTRODUCTION

This paper first discusses desirable standards for documentation that are realizable using current techniques and then discusses goals for documentation to be produced over the next five years using anticipated technological improvements. A quantum jump in documentation capabilities can be expected as the computer becomes able to aid the user in extracting relevant and timely information.

Good documentation is absolutely essential. It is not an accident that the hottest mini- and microcomputer products that have emerged over the past few years are distinguished from their competitors by higher-quality and easier-to-use documentation.

The paper is written from the perspective of a computer user and of a computer-center manager/systems programmer engaged daily in helping scientific/engineering users in a moderately sized computer center.

CURRENT DOCUMENTATION TECHNOLOGY

Good documentation requires careful planning and analysis on a level comparable to the software development and should be written in tandem with the code. Professional writers should be engaged and should have clearly defined responsibilities to ensure that the final product meets uniform standards.

Documentation must be *readable*. It should follow good English writing practices and flow smoothly. Detailed technical descriptions should be confined to the appropriate technical sections.

Documentation must be *targeted* at specific user groups such as students, casual users, experienced analysts, and so on. The authors should at all times be conscious of the user's level. Terminology and technical content should be adjusted as required.

Documentation must be *complete,* containing all the information the user will require. It must be *concise,* omitting information or background that is not required for a particular application. Extraneous detail clutters manuals, making them harder to use.

A set of manuals should have a common *glossary* and a *combined index* to the important keywords over all the manuals. These are especially important when faced with a transition to a new vendor and/or new system.

Documentation must be *structured.* Each manual must be well organized and have an overall plan that is apparent to the casual reader. The location of material in the document should be predictable to a new user familiar with the general subject matter. Sets of manuals should fit together in a cohesive, relatively non-overlapping fashion. Documentation put together without a guiding plan is often of uneven quality and information content.

Documentation should have an *appropriate structure.* Different uses of documentation can require radically different structures. In particular, tutorial introductions for beginners should be structured differently than reference material for systems programmers. Briefly:

- *Tutorial* material should gradually move from introductory concepts toward detailed descriptions. It should have a moderate number of examples in line and refer to a much larger set of examples in an appendix or auxiliary material, such as computer files distributed with the software. Conceptual diagrams should be strategically placed.
- *Reference* material should be written as an outline, permitting rapid answers to specific questions by concentrating information about particular topics. The reference section can contain many of the examples referred to in the tutorial material. It should also present a compact synopsis of the information for the frequent user.
- *Internal-logic* documentation should follow the conceptual structure of the program. This will vary based on the application area. For most programs, a top-down approach is most valuable for showing logic, but some programs in which data transactions are of primary importance may require a structure that traces data records and ignores the logical program order. High-level charts are important. The overview they provide is usually more important than the particular form (flow, HIPO, Warnier-Orr, etc.). Detailed flow charts should not be used because they are often too obsolete to be reliable. References should be made both to specific code portions and to the formal reference document.
- *Internal-code* documentation must be readable and refer back to the internal-logic documentation. It must be concise and not obstruct the reading of the actual code. Major code sections should be clearly delineated. Unusual or particularly important code techniques should be emphasized. Many of the computer-science structured programming practices that have been developed over the past decade are also highly applicable to documentation.

Documentation must be *timely* in its appearance. Frequently, a good set of documentation covering a product will not be fully formed until several years after the product's initial introduction into the field. This imposes extra problems in using state-of-the-art software technology. It is recommended the documentation proceed more or less simultaneously with the software development. This helps assure its

timely appearance and helps the developers avoid inconsistencies that are not noticed until the rules are codified onto paper.

Documentation must be *updated* periodically, both to integrate new material and to correct mistakes. Areas that have been found to be confusing to many users (based on a user feedback mechanism) should be clarified or rewritten.

A *new method for updates* to documentation is required. Many manuals are used that are not the latest versions. The causes for this problem relate to the high distribution cost of paper, which makes it impractical to circulate entire new sets of manuals with new software releases. The update notification process is often haphazard, having to filter through several layers of people, from the vendor's site through the user's. Finally, many users do not bother with regular insertion of the updates into manuals owing to the menial and time-consuming nature of the job.

The new update method should have a faster cycle. Examples and reference material in the documentation are sometimes incorrect. For most reputable vendors this is not a frequent situation, but when it does occur it can send users in circles until the problem is resolved. Long document-update cycles cause the same errors to be repeated by additional users even after a problem has been identified.

Documentation must be *field-tested!* This is standard practice for new software products and should be applied to new software documentation as well. Usually, new products are placed for Beta testing at experienced sites that can detect bugs in the code and produce their own workarounds. The problem is that these sites typically have person-to-person contact with the vendor and thus do not rely primarily on the written documentation. The suggestion is that software and documentation must also be placed at "inexperienced" sites, and records be kept of the user documentation queries.

Feedback from users must be encouraged. It is vital for vendor employees to be in periodic contact with real end users to learn what is required in the field. Feedback also helps focus attention on obscure sections and is a source of ideas for new manuals. If resources permit, all customer letters should be answered.

REALIZABLE NEW DOCUMENTATION TECHNIQUES

The preceding discussion mentioned many standards that good documentation must meet. Accomplishing this is a highly labor-intensive process, absorbing quality staff resources. Fortunately, continuing and new trends in computer hardware and software technology can make feasible in the near future (the next five years) techniques that will enhance the quality and speed the distribution of documentation.

Problems with Paper-Publishing Cycles

Current documentation techniques are built along standard paper-publishing technology. This leads to a relatively long documentation production and printing cycle before the material appears in the field and makes very expensive the updating and enhancement of the information. It also makes it impossible to tailor the documentation to a specific user's questions.

These problems are typically alleviated via the user's resident systems programmer. His/her job involves reading most of the manuals, knowing where information is located, reading bulletins about updates and new features, and having a reservoir of experience and knowledge to draw on in applying software facilities. While systems programmers will be essential in the foreseeable future, many of the lower-level, straightforward tasks can and should be automated. The result will be friendlier computers that are more tolerant of user mistakes and are able to tailor information to the specific needs of the user.

Software and Hardware Advances Applicable to Documentation

Advances have been made in the following areas:

- Cheap bulk read-only online random-access storage
- Codification and successful implementation of artificial-intelligence techniques in circumscribed applications
- Development of programmer apprentice artificial-intelligence programs

Commercial companies are beginning to interface videodisk systems to computers. These are being used in educational-training centers and inventory-parts applications among others. Capacities of 15,000 to 50,000 pages of text seem achievable using current hardware. These pages can be randomly accessed and are "printed" on a cheap-to-reproduce vinyl disk. These should be used for distribution of system manuals.

Immediate Benefits—Up-to-Date Manuals

The immediate benefit is a consistent set of up-to-date manuals for all the products with each new release. Since vinyl-disk manuals are relatively cheap (say $20–$50, compared to $500–$1000 per full set for many computer systems), they can be distributed automatically to all users.

Interim Benefits—Query Programs

The important interim benefit lies in the machine readability of the documentation. Many current systems are considered "friendly" because, among other things, they have consistently implemented online "Help" documentation throughout their command syntax. The information on line today is usually only a fraction of what could be available, considering that most new manuals are prepared using word processors. The cheap bulk storage approach can make most of this information available and make it amenable to analysis by programs.

Semi-intelligent query programs are already in use today in online database systems. An example is a medical pharmacological database that advises doctors on drug selection, tolerances, and side effects. The rapid growth that has occurred

in these services testifies to the viability and friendliness of query programs. It is time these techniques were applied to computer software. These query programs would allow a user to skim documentation and receive tailored extracts relevant to his/her immediate needs.

Note that general-purpose comprehension of all possible topics written in English is not required. Many artificial-intelligence programs perform very successfully in narrow fields of knowledge. Surely the computer field is eligible, considering the enormous efforts expended to make computer systems rigorous in their definitions, and predictable in their execution.

The query programs' capability to bypass irrelevant information and examples should not be undervalued. They can concentrate information based on what a particular user is requesting and based on his/her recent history of queries. Such programs also permit writing one "document" with all the information and having the query program present different levels of detail based on the applicability to the request on hand and the user's background. Currently, documentation writers must restrain the numbers and types of examples used to avoid cluttering the document and to avoid providing, for example, FORTRAN code to COBOL programmers. Query programs can select the appropriate ones.

Aspects of these query programs would be similar to Computer-Aided Instruction (CAI) programs, such as Control Data Corp's Plato. Indeed, it would seem reasonable that both types of programs could draw from the same database.

The online availability of examples makes it possible to produce programs that extract and retest the examples, to confirm both the continued functioning of the software and the correctness of the examples. This will help to lessen the historical problem of differences between the published reference-manual standard and the actual implementation standard.

System-error messages can refer to specific portions of reference material on code in error and, if requested, quote chapter and verse on possible causes.

Long-Term Benefits—Programmer Apprentice Programs

In the long term (five to ten years), the availability of documentation structured for access by computer programs will form a database usable by high-level artificial-intelligence programs. These are being developed to aid an analyst in formulating programs and to perform much of the mechanics of coding. They will respond to high-level commands, such as "use a quick-sort on the credit records, and then merge them with the master file." These programs will undoubtedly contain high-level abstract descriptions of basic computer-science algorithms. It is unlikely they will be able to contain code fragments to implement all possible variations of them. When faced with applying a specific algorithm, these programs will have to formulate many of the same questions that a person would in selecting language features to implement an idea.

Automatic recording of the high-level commands during the program-generation process would also be a start toward the internal program documentation discussed earlier.

Separation of the algorithm descriptions from the language-implementation details would allow use of new language features as they become available, allow people to give advice to the program on which features are more reliable, and permit comparison of new computer languages against previously developed realistic programs.

The online availability of documentation can also lead to programs that can respond to specific questions, such as "What will this language construct do when this parameter has this value?" or its inverse, "Given this behavior, what were the most likely inputs to this language construct?"

Implementation Considerations

These new features do not depend on immediate integration of video disks in particular. Another plausible candidate is the large-capacity "write-once" laser optical recording disk. More conventionally, rotating magnetic-disk storage costs have been dropping so consistently each year that dedicating say 5% of the total available (in typical moderate to large systems) would be sufficient to begin implementing many of these ideas. With time, this would apply to smaller systems also. The disk tradeoff would be amply repaid by the lowered software costs.

The full implementation of these methods will require considerable document writer/programmer/educator time. Furthermore, when operational, the system will use nontrivial CPU power. However, over a ten-year minimum life (for the more durable languages, such as FORTRAN, PASCAL, BASIC, C, and various well-known operating systems), such an investment would be repaid.

SUMMARY

The trend is clear toward low-cost bulk storage becoming commonly available. This fact, together with the increasing programmer shortage, should lead to the exploration and implementation of as many programmer aids as possible.

Good documentation is vital to the success of software products. By using well-known, current principles, vendors can improve their manuals. Design and writing of documentation should be an integral, simultaneous part of software development. Updates to manuals should be distributed on a regular basis, as frequently as needed.

New techniques that apply the computer's potential for artificial intelligence in specialized areas should make it feasible to query/abstract from documentation databases. This will instrumentally increase the friendliness and usability of systems and ultimately be a fundamental part of the knowledge base for programmer apprentice programs that assist in program development.

REFERENCES

1. Dwyer, Barry. "A User Friendly Algorithm." *Communications of the ACM,* 24 (1981), pp. 556–561.
2. Gass, Saul I., Karla L. Hoffman, Richard H. F. Jackson, Lambert S. Joel, Patsy B. Saunders, "Documentation for a Model: A Hierarchical Approach," *Communications of the ACM,* 24 (1981), pp. 728–733.
3. Waters, Richard C., "The Programmer Apprentice: Knowledge Based Program Editing", *IEEE Transactions on Software Engineering,* Vol. SE-8 (1982), pp. 1–12.

The development of software engineers: a view from a user

by WALTER P. WARNER
Naval Surface Weapons Center
Dahlgren, Virginia
and
RICHARD E. NANCE
Virginia Polytechnic Institute & State University
Blacksburg, Virginia

ABSTRACT

Like many organizations, the Naval Surface Weapons Center (NSWC) has recognized a tremendous growth in the use of computing resources. How NSWC focused attention on the crucial role of software development technology and devised a plan for dealing with the scarcity of competent software development personnel is the subject of this paper.

The necessary knowledge areas for software engineering are identified in discussing the academic requirements, and some conclusions arrived at during the deliberations are mentioned.

INTRODUCTION

Like many organizations, the Naval Surface Weapons Center (NSWC) has recognized a tremendous growth in the use of computing resources. This growth is reflected in several statistics; a simple indicator is the increase in users of the central mainframe from 1,350 in 1978 to approximately 1,800 in 1981. The development of computer software delivered to the Navy and Marine Corps, estimated to consume 610 staff-years in 1979, was projected at 640 staff-years in 1980. Such rapid growth has brought several difficulties to the surface, but the overriding problem is the scarcity of software development talent. The way that NSWC focused attention on the crucial role of software development technology and devised a plan for dealing with the scarcity of competent personnel is the subject of this paper.

Aware of the Bureau of Labor Statistics[1] estimate that the need for software development personnel is to double between 1978 and 1981, NSWC conducted an informal survey of internal requirements. The results were startling: an addition of 117 to the workforce of 2,084 professionals in FY 1981 and an addition of 373 over five years. The National Science Foundation prediction of a 3.5:1 ratio of positions to graduates with master's and bachelor's degrees in computer science clearly acknowledges the global nature of the problem observed by NSWC.[1]

The NSWC approach to the problem of scarcity of software development talent should not be viewed as a preconceived insightful long-range strategy. Admittedly, the plan has evolved, and this description of the evolution takes the following order:

1. A background sketch of the NSWC showing its early and long-standing role in Naval computing efforts and software development
2. A picture of the current weapons system context, in which the software development task is conducted
3. The NSWC initiatives that collectively define the plan
4. The conclusions arising from the various initiatives that have exerted significant influences

NSWC BACKGROUND IN SOFTWARE DEVELOPMENT

The history of software development at the Naval Surface Weapons Center began with the advent of large digital computers for laboratory usage in the late 1940s. Because of its long-standing mission responsibility for the numerical data required to aim, target, or control Navy weapons, the Naval Proving Ground (combined with the Naval Ordnance Laboratory to form NSWC in 1975) was the first Naval activity to have a large-scale computer. Correspondingly, the Center was the first Navy activity to develop and support software for Navy-deployed operational digital systems, beginning about 1960. With the continually evolving Navy requirements for digital computer applications, the Center has sustained its leading role in digital computer applications and software development expertise.

The Center is responsible for the complete weapons control software package development, testing, and operational support for the Fleet Ballistic Missile Systems POLARIS, POSEIDON, and TRIDENT. It has provided the wrap-around simulations and facilities for testing the digital fire control programs for the TARTAR, TERRIER, and TALOS surface missile systems. Similar support has been provided for the Mk 86 and Mk 92 gunfire control systems. In connection with the Navy's Gunnery Improvement Program, the Center has developed the software for the Mk 68 digital fire control system for 5-inch guns.

The Center has pioneered the application of minicomputers for Fleet electronic warfare (EW) systems and ELINT processing systems. These digital systems have enabled orders-of-magnitude advancements in processing quality and quantity, in data response time, and in overall Fleet EW effectiveness. Two current major examples are the development of the Airborne ESM Data Analysis Systems and the support of the AN/SLQ-32 EW Countermeasures Suite. The Intelligence Analysis Center for the Marine Air/Ground Intelligence System (MAGIS) and the shipboard Intelligence Center for LHA and CVV installation are additional examples of systems developed in-house by the Center. These are the first major deployed intelligence database-oriented systems incorporating Navy standard computers. Among the large software systems developed on general-purpose computers are the TRIDENT Advanced Weapon System Simulation and CELEST (a satellite orbit determination program).

COMPUTING SUPPORT FOR NAVAL WEAPONS SYSTEMS

Operational software might also be described as software for "embedded computer systems." The principles and techniques underlying the software development task for large, complex systems apply regardless of the applications context. However, the enclosing system and the application often impose constraints and limitations of time (time-critical requirements) and storage that are shared only by the most challenging general-purpose programs (e.g., operating systems, run-time control systems, etc.).

The typical Naval weapon system is becoming extremely complex. The AEGIS weapon system, for which the Center has life cycle support of the software, is an illustrative example

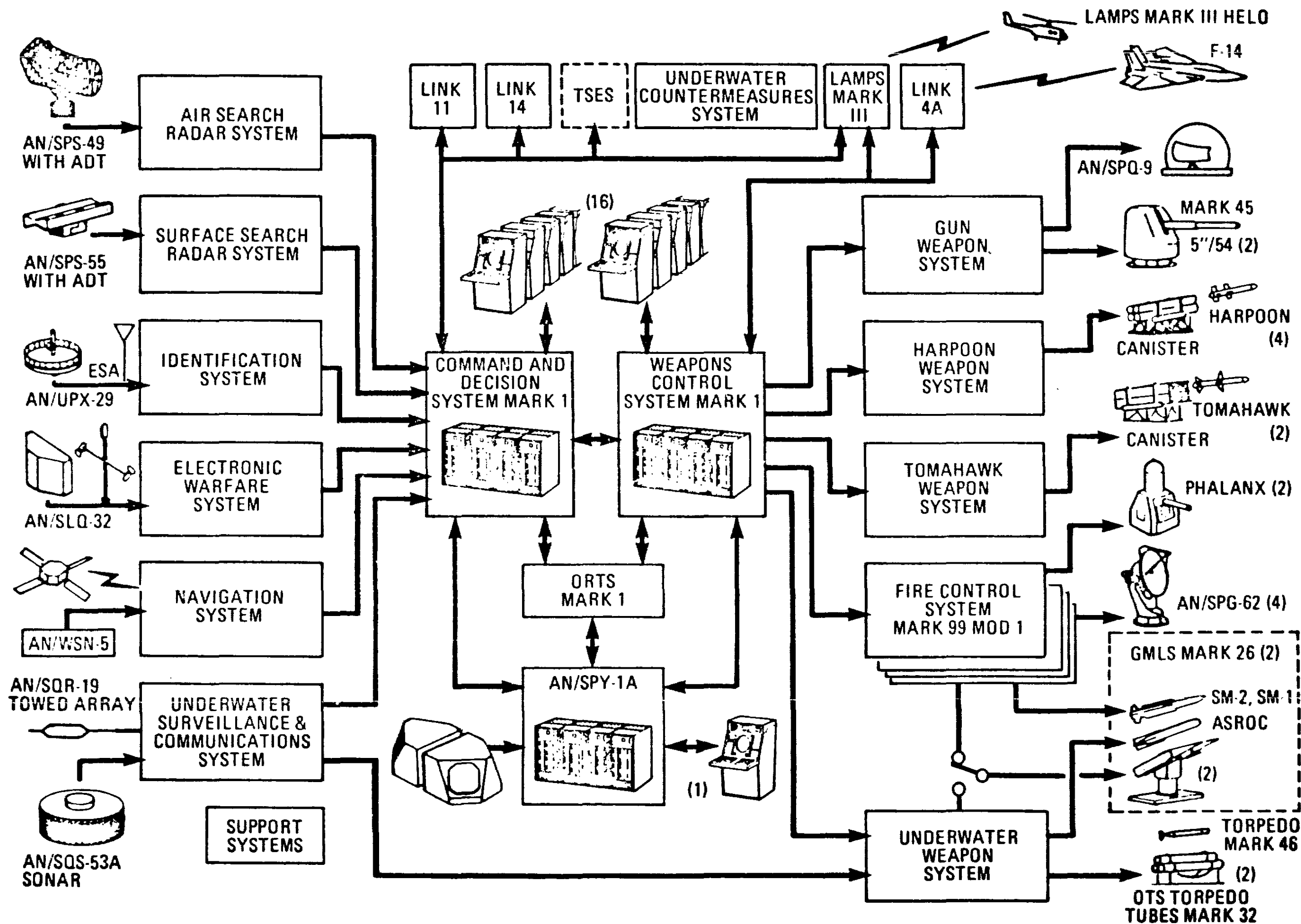


Figure 1—AEGIS ship combat system—baseline

(see Figure 1). Specific configurations for different ship classes may have different subsystems, but most of the subsystems, shown pictorially around the periphery, contain embedded computers. The total software development task includes:

1. The *operational software* for the embedded computing support of the weapons subsystem
2. The *integration software* enabling the necessary communications among subsystems
3. The *control software* by which the weapons subsystems function as a total system
4. The *life cycle support software* necessary for testing, verification and validation, and maintenance and enhancements

The magnitude of this task is underscored by the simple statistic that 45 computers are required in some AEGIS configurations.

NSWC INITIATIVES

The evolutionary plan has proceeded primarily through the work of the following committees:

1. A committee that developed a computing technology seminar for top-level management. This committee consisted of an NSWC middle-level manager with software development experience, an industry representative, and a university faculty member.
2. A second committee that examined the required background knowledge for software development at NSWC. This committee identified the necessary knowledge areas, evaluated their relative importance and assessed the utility of on-the-job training, in-house or academic courses in each area. This committee included experienced project managers and a computer science faculty member.
3. A third committee, comprised of representatives from

technical departments responsible for software development, that explored the alternatives for developing the needed expertise. This committee addressed formal academic programs, internal training, and employee development possibilities.

COMPUTING TECHNOLOGY: A SEMINAR FOR SENIOR EXECUTIVES

A seminar consisting of talks by top experts in the country was conducted by NSWC.[2] The purpose of this seminar was to understand computing technology in general and software development in particular better at all levels in the organization. The historical development and the perceived future technology, integrating the hardware and software evolution, were presented. Other topics were the software development process, distributed computing and computer networks, the regulatory environment, and information support systems for management. This seminar, presented in a three to four day concentrated format, has been given to all managers from the commanding officer and technical director to first line supervisors and to two groups of Naval Flag rank officers (admirals), who are some of the NSWC sponsors.

IDENTIFYING THE ACADEMIC PREPARATION

An NSWC study was made to determine the best academic preparation for the kind of software engineering performed at NSWC.[3]

Needs Specification

The participants in this study could not come to an agreement on a definition of a software engineer; therefore, we began by identifying the knowledge areas important to the development of the Navy systems for which NSWC has, or could have, responsibility. These knowledge areas, considered to be necessary for those individuals developing successful systems, are defined in Table 1. A definition notwithstanding, in the remainder of this paper we use the term "software engineer" to identify the required software personnel.

Areas of Academic Preparation

The study committee encountered a problem experienced by many groups in examining the role of the software engineer; i.e., the inability to differentiate between the function of the systems engineer, who has total systems responsibilities, and the software engineer, who is responsible for the software subsystem and its effect on that total system.

After reviewing several sources,[4–9] we recognized that a consensus exists concerning the definition of software engineering, and consequently, some direction was provided in defining a "software engineer."

- The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines (Bauer, p. 530).[10]

We were pleased to discover that this definition did not differ in principle from that given in the Department of Defense Directive 5000.29 of 26 April 1976:

- (The) science of design, development, implementation, test, evaluation, and maintenance of computer software over its life cycle.

In order to circumvent the problem of differentiating between the responsibilities of the systems engineer and the software engineer, we sought to compare or contrast the two (software and systems) using the 13 knowledge areas. Applying the following scale: 4—in-depth knowledge, 3—good working knowledge, 2—some knowledge, and 1—little or no knowledge; we produced a consensus estimate of the importance of each area in fulfilling the duties of the software and the systems engineer (see Table I).

Academic or On-the-job Knowledge

In some knowledge areas, academic preparation is believed to be less effective than job experience. Table I reflects the consensus regarding the better source of knowledge, and in some cases, a closer description of the contributing experience is given.

Integration and Summary

Obviously no academic program is likely to offer preparation in all the areas marked in the academic column of Table I (Areas 1, 5, 7, 9, 10, 11, 12, 13). The identification of the most essential areas of academic preparation should follow from the inspection of Table I. A reasonable first cut is to begin with the academic areas (those where academic preparation was judged better) designated as requiring in-depth knowledge (level 4). This discrimination identifies only the areas of programming systems techniques (Area 9) and information structures (Area 11). Relaxing the criterion slightly admits interpersonal communication skills (Area 4), which are not clearly designated as better with academic preparation.

At this point we examine the remaining areas, judging the importance of the area considering the better source of knowledge. The consensus is that Areas 5 and 6 should be added as *primary* knowledge, and the remaining designated as either *secondary* or *useful*. This decision produces the resulting classification:

1. Primary
   a. Interpersonal communications skills
   b. Functional capabilities of digital hardware
   c. Software design technology
   d. Programming systems techniques
   e. Information structures
2. Secondary
   a. Process exposure/dynamic interrelationships
   b. Design principles
   c. Systems integration
   d. Human factors engineering
   e. Systems simulation

TABLE I—Software engineering knowledge areas

KNOWLEDGE AREAS:	KNOWLEDGE BEST PROVIDED BY ACADEMIC	ON-THE-JOB	KNOWLEDGE AREA IMPORTANCE SOFTWARE ENGINEER	SYSTEMS ENGINEER
1. Controls - controls, information feedback systems, basic systems distinctions (open loop, closed loop, hierarchical, etc.).	x		2.5	4
2. Process exposure - dynamic interrelationships-time-dependent behavior, system interactions, the "process" concept, cross effects, binding time, process communication, cooperation, and competition.		x(a)	4	4
3. Design principles - the principles of engineering for the scientific method), elements of the design activity (specification, analysis, decomposition, syntheses, testing), maintenance and reliability.		x(b)	4	4
4. Communication skills - written and verbal communication, team participation, and team leadership.	*	*	3.5	4
5. Digital hardware - logical structure and composition. (This area was recognized to be potentially divisible into computer hardware and digital non-computer hardware.)	x		3	2
6. Software design - system life cycle, specification techniques (e.g., PSL/PSA, workbook, Jackson, etc.), development techniques (e.g., chief programmer, structured walk-through, design review, builds, code reading), documentation, modification, maintenance and configuration management.	**	x(c)	4	2
7. Evaluation - systems analysis techniques, models and modeling, identification or creation of alternatives, characterization of trade-offs.	x	**	3	4
8. Systems integration - component and subsystem testing, systems reliability, progressive testing, diagnostic capability, degraded mode options, recovery.		x(d)	4	4
9. Programming - programming languages, systems programs, structured programming, modularity, stubs, program documentation, program testing.	x		4	4
10. Human factors - human/machine interface, dialogue design, prompting, "trainability" and "learnability," adaptability and design for change. (This area was recognized as potentially divisible into software design for human use and hardware design for human use.)	x		3	4
11. Information structures - logical and physical organization of data, data definition, abstract data types, database technology.	x		4	2
12. Communications technology - digital communications, devices, data transmission, coding techniques, protocols, security.	x		2	2
13. Systems simulation - experimentation and system testing using simulation, discrete event and continuous simulation models, wrap-around simulators, emulation.	*	*	3	3

NOTES: * neither or both; ** better job needed; (a) development and support of real-time systems; (b) design of systems emphasizing hardware, software, and firmware interfaces from user needs specifications; (c) development of large software systems; (d) design and testing of systems with software and hardware components.

3. Useful
   a. Controls knowledge
   b. Evaluation
   c. Communications technology

We believe that an academic preparation exposing the student to the primary areas is essential. As many of the secondary areas as possible should be included.

University Contacts

Based on our study of the needs for and the necessary training of software engineers, we have communicated with the presidents of several universities in the local area and plan to follow this up with visits and presentations to their administrations. Our purpose is to try to encourage them to include the academic training needed for software engineering, since it is from these universities that we obtain many of our new hires. Their initial response has been favorable, but it is well known how long it takes a university to shift its gears. It is also well understood that they have a real problem in obtaining adequate faculty for their present programs.

SOFTWARE ENGINEERING DEVELOPMENT PROGRAM

In order to try to meet NSWC needs for software engineers, another committee developed a program to retrain some of our own in-house people and to better educate those who are already doing software engineering tasks.[11]

We reviewed and accepted the 13 knowledge areas identified as being critical to software engineering in the previous study regarding the academic preparation of software engineers.

Three core courses were selected as providing the necessary foundation for learning software engineering: (1) FORTRAN (Structured) for Scientists and Engineers, (2) Computers Systems Organization, and (3) Software Development.

FORTRAN for Scientists and Engineers provides an introduction to computer programming using FORTRAN IV and the CDC 6700 computer. Practice problems, dealing with topics from mathematics, will not be written in an arbitrary fashion, and heavy emphasis will be placed on some of the contemporary structured programming techniques designed to produce readable, coherent, and structured programs. To meet this goal, students will be expected to follow coding conventions provided by the instructor. The instructor will stress: (1) efficiency of algorithm and code, (2) program structure and style, (3) documentation (in-line commentary), and (4) correctness of answers and form.

The Computer System Organization course is an introduction to the generic organization, or structure, of digital computers. It also includes instruction in machine and assembly language programming which requires a more extensive knowledge of computer hardware than does the higher-level language programming covered in Course I. The material is taught from a hardware user's standpoint. The course will not prepare a person for work in designing computers. It is oriented toward preparing the software engineer to specify or select computers for various needs and to understand better the implications of hardware design on the software.

Software Development is a course on the development of software for large single- and multi-computer applications, using both assembly and higher-level languages. Development extends from definition of requirements through introduction into use. Life support functions also are included from the standpoint that for many applications, development continues throughout much of the life of a system. This course, of necessity, is taught within the framework of systems engineering methodology, but avoids addressing the full scope of the technology involved in the broader process. Three examples are carried through the course to illustrate the material and provide the basis for work assignments.

The three core courses at the Dahlgren Laboratory are being sponsored by Mary Washington College. Because they are just getting started in their own computer science curriculum, we are teaching them with our own personnel. The courses are being taught at the White Oak Laboratory under the auspices of the University of Maryland.

Having completed the three core courses, the trainees should have an understanding of the basics of software. It is planned to follow up with the following series of short (three to four days) courses covering the entire spectrum of software engineering:

1. *Comparative Software Engineering:* A summary seminar that acts as an introduction to the technical subject matter covered by the curriculum. The seminar provides a broad overview of the alternatives, experiences, and issues encompassed within the field.
2. *Comparative Design and Analysis Techniques:* A seminar which discusses in detail how various modern requirements and design techniques can be used to reduce rework costs and improve quality through better specifications.
3. *Modern Programming Techniques:* A seminar which describes how structured programming concepts and principles (i.e., structured design, top-down development, modular coding, etc.) can be practically implemented in weapons systems within the Navy.
4. *Software Testing:* A seminar which describes different test approaches, techniques, and tools that can be used to realize more quantitative goals set for software and system testing.
5. *Embedded Computer System Architectural Engineering:* A summary course that provides the attendee with the knowledge needed to make informed hardware/firmware/software engineering trade-off decisions.
6. *Software Project Management:* A summary seminar which addresses the subjects of project planning, organizing, staffing, directing, and controlling real-time weapons projects.
7. *Software Economics:* This seminar surveys the dynamic field of software cost estimation, compares methods, and discusses experiences both pro and con of various software cost estimation models. Productivity measures and evaluation methods are described.
8. *Software Configuration Management:* This seminar discusses the subject of configuration identification, change control, status accounting, and verification in the context of Navy systems and documentation requirements (e. g., MIL-STD-1679, SECNAVINST 3560.1).
9. *Software Quality Assurance:* This seminar discusses how a software quality assurance program responsive to MIL-S-52779 can be implemented to provide for an objective set of checks and balances. It also treats the subject of software testing.
10. *Software Acquisition Management:* A seminar that discusses the issues and experiences associated with planning and conducting software development in an acquisition environment.

CONCLUSIONS AND SUMMARY

1. The shortage of software engineers and the needs of software development activities has caused NSWC to take a hard look at how to solve the problem. Colleges and Universities cannot supply them in sufficient quantities in time to meet the needs.
2. There is a general lack of appreciation for the role of software at all levels of management. There is also a lack of agreement on what software engineering is and, therefore, what the duties of a software engineer are.
3. The NSWC seminar for senior executives has given management a better appreciation for the problems of developing software and has done much to foster the acceptance of new software development techniques. It has also focused the attention of top-level management on the

need to do something to solve the problem of the supply of software engineers.

4. The knowledge areas designated as primary in this paper are essential in any software engineering program. As many of the secondary areas as possible should be included. This list should be helpful in developing software engineering programs and in evaluating existing or proposed programs in an academic, commercial, or internal setting.
5. No amount of formal training, either academic or in-house, can produce expert software engineers, and the best that can be done with an in-house program is to give people enough knowledge to get them started in the field. It is believed that those who successfully complete the in-house training program, with additional on-the-job experience and formal training, can progressively assume the responsibilities of software engineers.
6. The decision that the three core courses should be taught under the auspices of a college reflected the conviction that the participants would perceive the necessity of their commitment to the training, because it begins in an academic environment with attendant homework and grades. It is hoped also that receiving college credit for the courses will encourage some of the trainees to pursue a second degree in a computer-related field.
7. Trainees who wish to change career fields to software engineering will stay in their present jobs while taking the core courses. Thus those who do not succeed in the program will still be in appropriate positions. It is felt also that any amount of knowledge gained from the courses will be beneficial in any job in an R&D organization such as NSWC.
8. The training plan is initially being restricted to holders of scientific degrees in order to have a somewhat homogeneous background and scientific maturity in the classes. It is recognized that there will be others without degrees who may have the capability of succeeding in the program, and they will be considered on an individual basis.
9. The response of the program is encouraging. One hundred and fifteen employees have signed up for the Software Engineering Program. All are non-programmers with degrees in engineering, physics, chemistry, materials sciences, etc. Of this group, 58 have taken the FORTRAN class; the others, having had some experience with FORTRAN, will be taught structured programming.
10. Activities such as NSWC should encourage their personnel to attend universities granting advanced degrees in software engineering under government-sponsored programs.
11. An effort should be made among academia, industry, and government to arrive at a consensus of the definition of a software engineer and his duties. This definition should clearly define the differences between a system engineer and a software engineer, similar to the distinction that now exists between systems engineers and electronic engineers.
12. Colleges and universities should move more rapidly in the direction of educating software engineers. Industry and government should cooperate in the delineation of the needs and providing an interchange of personnel and experience with universities.

REFERENCES

1. COMPUTERWORLD, Vol. XIV, No. 45, November 3, 1980.
2. Warner, W. P., R. E. Nance, and J. H. Manley. "Computing Technology: A Seminar for Senior Executives." NSWC TR 79–174, June 1979.
3. Nance, R. E., and W. P. Warner. "Anticipating the Software Engineer: The Academic Preparation." DTIC AD-A086827 (NSWC TR 80–108. May 1980).
4. Hoffman, A. A. J. Personal correspondence, February 18, 1980.
5. Stucki, L. G., and L. J. Peters. "A Software Engineering Graduate Curriculum." *Proceedings of the 1978 ACM Annual Meeting,* Washington D.C., (December 4–6, 1978), pp. 63–67.
6. Fairley, R. E. "Educational Issues in Software Engineering." *Proceedings of the 1978 ACM Annual Meeting,* Washington, D.C. (December 4–6, 1978), pp. 58–62.
7. Fairley, R. E. "Software Engineering Education." *Proceedings of the Thirteenth Annual Hawaii International Conference on System Sciences,* Hawaii (1980), pp. 70–75.
8. Jensen, R. W., C. C. Tonies, and W. I. Fletcher. "A Proposed 4-year Software Engineering Curriculum." *SIGCSE Bulletin,* Vol. 10, August 1978, pp. 84–92.
9. Hoffman, A. A. J. "A Proposed Masters Degree in Software Engineering." *Proceedings of the 1978 ACM Annual Meeting,* Washington, D.C. (December 4–6, 1978), pp. 54–57.
10. Bauer, F. L. "Software Engineering." *Information Processing,* 71 (1971). North Holland Publishing Co.
11. "Report of the Software Engineering Committee." NSWC AP 81–314, August 1981.

An industrial software engineering methodology supported by an automated environment

by MICHAEL S. DEUTSCH
Hughes Aircraft Company
El Segundo, California

ABSTRACT

In recent years, industry and government have sought to formalize software development by constructing automated environments that support the application of modern techniques and methodologies to the production of software. This paper describes the automated software development system being installed at Hughes Aircraft Company. This system is expected to be a major contributor to the orderly management of software development at Hughes.

This software development system consists of integrated development techniques over the life cycle, a set of software tools, and a physical facility for software development and test. Structured methodologies such as structured analysis, structured design, and structured programming are supported by automated tools. The configuration of the software development facility consists of a host software development system, the target machines, and the user display terminals.

Project planning and performance measurement are based on the rate charting technique and earned value assessment.

INTRODUCTION

The "software crisis" of which we are constantly reminded is connected with the vastly increased complexity of contemporary data processing systems and the limited ability of traditional software practices to deal with this complexity. Only within the last several years has this stagnation in software technology been generally recognized and accepted in government/industry circles with the realization that current software practices were of little help in attacking increasingly complex applications.

One of the more positive responses of industry to this situation has been to formalize software development into an engineering practice by developing automated software development environments that support the application of modern techniques and methodologies to the production of software. These software development systems have been built largely as proprietary products with exact characteristics varying from organization to organization. Regardless of the differences, these systems support the same thrust, i.e., that orderly software management and predictable results are based on methodologies of how software is specified, structured, and integrated into larger systems.

In this paper, the software engineering development system being installed at Hughes Aircraft Company's Space and Communications Group is described in terms of the development methodologies being used, the software tools that support the methodologies, and the facility that hosts the tools. An overview of the development system is presented followed by a description of the engineering method within each life cycle phase. The project planning and performance monitoring approach is also described.

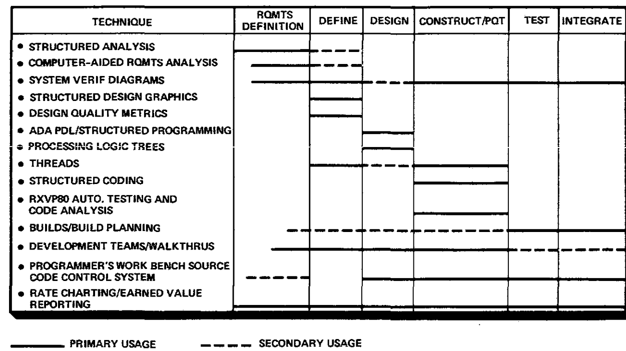


Figure 1—Summary of tools and techniques

SOFTWARE DEVELOPMENT SYSTEM

The three required constituent elements of a software engineering development system are: (1) an overall approach of coherent methodologies covering the entire software life cycle, (2) a set of software tools that supports the consistent application of the methodologies, and (3) a computational facility that houses the tools.

The techniques and tools for the Hughes development system are delineated on Figure 1 for each life cycle phase. Note that several tools/methodologies span multiple life cycle phases and provide a unifying influence. Noteworthy is the system verification diagram (SVD) that is used for requirements and design verification and to guide the construct, test, and integration processes. (The SVD technique was originally conceived by Computer Sciences Corporation.)

The software development facility is shown on Figure 2 and consists of a host software development system, the target machines, and the user display terminals. The host development system consists of several PDP 11/70s and VAX machines. The 11/70s house the programmer's work bench (PWB), which is a facility for source code generation and word processing. The VAX hosts a set of requirements definition and design tools. Requirements engineering, design, and code generation are accomplished on the host system independent of the target machine.

Dynamic execution of code is performed on the target machines. The target machines contain machine peculiar tools including compilers, assemblers, linkers, debuggers, and automatic test tools.

User terminals are linked to an "intelligent" microcode driven switching device called a port contention unit. A terminal may be connected by user request at sign-on time to any of the host system or target machines. There is at least one terminal for every two programmers (located in their offices), thus providing practically unlimited access to computational resources.

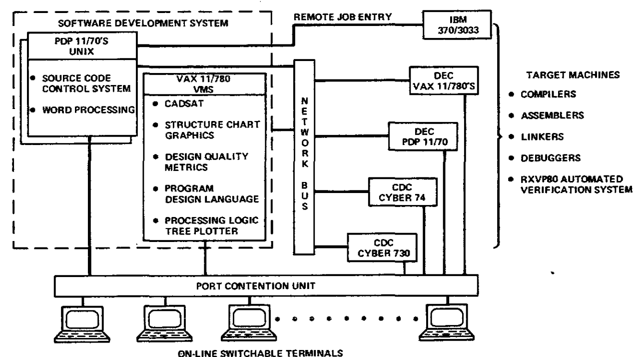


Figure 2—Software development facility

Requirements Definition Phase

This phase normally consists of two constituent activities: (1) a specification activity that generates the system level functional requirements specification and (2) an allocation activity that generates a specification for requirements allocated to each computer program configuration item (CPCI).

The structured analysis methodology is used to analyze the requirements and produce structured specifications. A data flow diagram is constructed to present a logical model of the system functions. By successively decomposing the elements of the data flow diagram, the system is disclosed in an order proceeding from the most abstract to the most detailed. All the data items flowing between bubbles are defined in a data dictionary. Each function in the detailed level data flow diagrams is described by a process description.

CADSAT is a PSL/PSA based tool that supports structured analysis by describing a system in machine-readable form. Objects that play a role in the system, data elements and processes, are named in a requirements language along with the relationships among the named objects. CADSAT consistency checks the stated requirements and reports on possible discrepancies in the defined relationships.

Functional requirements verification is achieved using a technique based on the mapping of functional requirements into units called stimulus/response elements. Each element identifies the stimulus (or input condition) and response (or output condition) of each function in the system or CPCI. The stimulus/response elements are then graphically analyzed showing all stimulus/response flows through the software system. This graphical mapping is the SVD.

The process of preparing an SVD reveals and highlights errors of completeness, consistency, and redundancy in the functional requirements. These errors will appear in the form of incomplete stimulus/response pairings, illogical connections between stimulus/response elements, and contradictions or redundancies among stimulus/response elements.

Preliminary Design

In this phase, the CPCI functional specification requirements are transformed into a preliminary physical design. The Constantine/Yourdon structured design methodology is used.

The data flow diagrams, included in the CPCI requirements specification, are scrutinized to identify certain generic design constructs. These constructs are the basis for converting the logical design (the data flow diagrams) into a physical design. The constructs form the basis of an initial structure chart. The structure chart shows the structure of the program modules, the interface with data modules, and the parameters passed between modules. This structure is successively refined until each of the modules in the architecture corresponds to 100 HOL lines of code on the average.

This design methodology is supported by two automated tools: (1) structure chart graphics (SCG) and (2) design quality metrics (DQM). SCG is a display interactive tool for creating, modifying, and maintaining structure charts. A hardcopy of the created structure chart can be output that is suitable for deliverable documentation. DQM analyzes a structure chart by identifying areas of the design that are overly complex using algorithms based on a hierarchical tree model. Highly complex sections are potential problem areas that are subjects for redesign on the next iteration.

Verification of the design versus requirements is accomplished by "threading" or associating each stimulus/response element of the SVD with a sequence of software modules that implement the stimulus/response pairing. Incomplete, missing, or extraneous associations suggest a nonresponsive design or misinterpretation of requirements. This allocation of modules to threads is the method by which a visible connection between requirements and design is maintained throughout the development cycle.

Detailed Design

Each of the modules identified in the structure charts is expanded into a detailed design. The design of each module is expressed in pseudo-code and input interactively into a programming design language (PDL) processor. Logic tree plots are automatically generated by a tool that uses the PDL syntax as a command language. These logic trees represent the PDL syntax in a graphical form that permits better comprehension of the abstract information and are suitable for design walkthroughs, design reviews, and deliverable documentation.

A development presently in progress will replace the existing conventional PDL with an Ada PDL. This will permit a module design to be developed in two major steps of refinement—specification and implementation. A module specification providing a functional definition of the module procedure and a definition of the visible data interfaces is produced first. The module implementation providing the design of the procedure that operates on the visible data is then generated. One of the major advantages of the Ada PDL is that it will provide automated verification of interface consistency between module specifications.

Software Construction

The software construction phase entails the coding, checkout, and preliminary qualification testing of each CPCI. A build plan for each CPCI is graphically depicted as a calendarized network of threads that were previously defined on the system verification diagram. Because each thread is correlated with specific modules, the coding sequence of modules is defined by the build plan. Each thread undergoes a preliminary qualification test before being baselined.

HIFTRAN, a Hughes developed structured FORTRAN preprocessor, is used wherever possible as the source language.

Each thread is exhaustively tested with the assistance of the RXVP80 automatic test tool. RXVP80 "instruments" the code to determine the extent of the testing coverage. Reports are generated by RXVP80 showing which paths of the code have been covered by previous testing and which paths remain to be tested. Additional test cases are contrived to target on previously untested paths. This sequence is repeated until a

complete (or very close to complete) path coverage is attained.

Actual project experience has shown that this early emphasis on comprehensive testing using the automatic test tool reveals a significant number of errors during the construction phase that otherwise would have gone undetected until some later time. During subsequent periods, however, including operations, the detection rate of latent errors is lower. The cost of rectifying an error later in the life cycle is, of course, higher than if detected earlier. The comparative error detection profiles, depicting testing with and without the aid of the test tool, is illustrated on the left-hand side of Figure 3. Exhaustive testing assisted by the automated tool, while requiring a slight additional level of immediate testing effort, is believed to be a worthwhile investment that pays dividends in reduced life cycle costs.

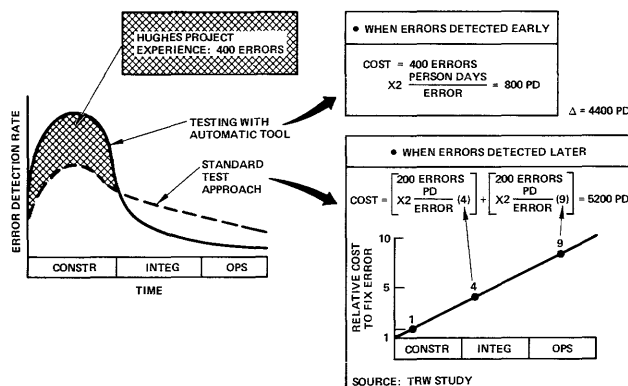


Figure 3—Testing with automatic test tool pays off in life cycle costs

The exhaustive test procedure applied on a recent project at Hughes detected an average of one additional error per thread (approximately 400 additional total errors). The cost, in schedule time, of performing the extended testing ranged from one-half day to three days per thread, with the average schedule cost close to one day per thread. Normally, two persons were involved in testing the thread at this point and, therefore, the incremental cost of the exhaustive testing effort was an average of two person-days per thread. The subject project consisted of about 400 threads. The incremental cost of finding and correcting the 400 additional errors was 800 person-days (400 errors × 2 person-days/error) as shown on the upper right of Figure 3.

The average relative cost to fix an error during the integration activity versus construct activity is four times as great, and the average relative cost to fix an error during the operations phase versus construct activity is nine times as great. To model the life cycle cost benefits, it is assumed that discovery of the 400 errors would otherwise have been evenly distributed over the integration and operations periods. This is probably a conservative estimate since the type of error overlooked during the construct activity is more likely to reappear during operations in which the software would otherwise undergo its first thorough exercise. The 200 errors found during integration would cost 1600 person-days (200 errors × 2 person-days/errors × 4) to correct under this model; the 200 errors found during operations would cost 3600 person-days (200 errors × 2 person-days/error × 9) to correct. The differential savings in life cycle cost achieved by the exhaustive testing strategy on this project is estimated to be 4400 person-days (3600 + 1600 = 5200 person-days less 800 person-days). This computation is depicted on the lower right of Figure 3.

Integration and Test

An incremental integration and test philosophy is based on the "builds" technique. In this approach there is considerable overlap between system integration and CPCI construction. A major emphasis is to segment a complex system development into smaller, more manageable, functionally oriented segments called builds.

Testing at the system level is planned and organized in the same manner as the "thread" testing at the CPCI level. The series of system level tests will integrate CPCI versions and hardware CIs. An SVD derived from the system requirements specification is used to establish the content and order in which partial versions of CPCIs and CIs are developed and introduced into the integrated testing process. Adding only one new element at a time toward a deliverable system capability permits more efficient detection and correction of interface problems.

Builds are incrementally constructed from components of one or more CPCIs and hardware CIs. Each build augments a previously established baseline. Prior to build testing, a preliminary qualification test is performed on components of the CPCI to establish a CPCI baseline version. This baseline is subsequently augmented by additional components which extend the baseline to a complete CPCI. The key objective with this approach is to establish a logically complete system skeleton early in the integration period.

The merits of this approach include: (1) demonstration of key functional capabilities early in the development cycle, (2) early demonstration of the essential viability of the system, (3) early demonstration of key interfaces, and (4) minimization of special test bed environments required for test and integration.

PROJECT PLANNING AND PERFORMANCE MEASUREMENT

At this point, the software development process has been examined from a technical perspective. This development approach is now explored in terms of some of the accompanying management methodologies that complement the technical approach. A project planning approach, based on earned value reporting, that is a natural adjunct to the technical software engineering process will be described. This planning approach consists of methods for scheduling, reporting, and monitoring development progress.

Earned value measurement is directed toward assessment of progress through comparison of actual versus planned expenditures and schedule. The procedure involves decomposing a project into small work packages. Each work package is accompanied by frequent milestones with specific

completion criteria, a situation naturally supported by the engineering process previously described.

At periodic points, schedule and cost variance for each work package is determined and summarized at various levels of the work breakdown structure up to the total project level. Three parameters are used in this determination: (1) the budgeted cost of work scheduled (BCWS), (2) budgeted cost of work actually performed (BCWP), and (3) actual cost of work performed (ACWP). The cost variance is the difference of the ACWP and BCWP. The schedule variance expressed in dollars is the difference of BCWP and BCWS for the effective reporting date.

The basic management tool used in this project planning approach is the rate chart. It is a simple two-dimensional plot of the percentage of work planned and actually completed as a function of time.

As illustrated on Figure 4, planning begins by constructing a master schedule bar chart showing the time-oriented relationships among the various phases of the project. Although the bar chart is an effective tool for initial project planning, it inadequately portrays overall project status and production trends. Instead, a technique called "rate charting" is used. The composite rate chart plan shown on Figure 4 has been derived from the bar chart. The rate chart shows start and end planning dates (derived from the bar chart) and production rates. By weighting the work in each of the phases according to the allocated budget, a total project production rate can be planned as shown here.

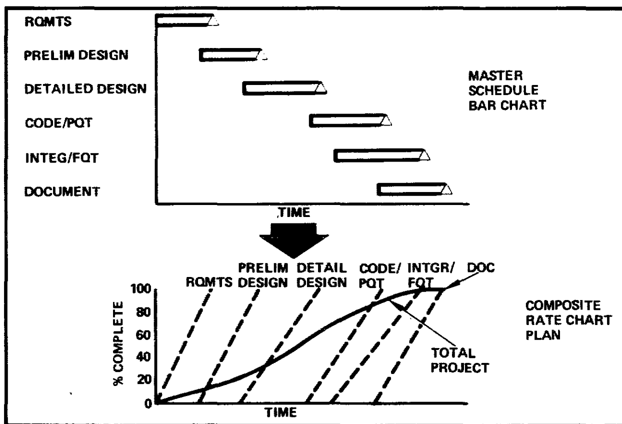


Figure 4—Rate charts: tools for monitoring progress

Rate charts provide visibility for all levels of management—individual work areas, project management, and customer. By evaluating the slope of the actual production rate with respect to the planned rate, a manager is alerted to trends and can consider reallocation of resources.

Each of the development activities is broken down into several or more work packages. Planned versus actual accomplishments are monitored at the work package level and summarized on a composite rate chart. Individual work package contributions to the composite summaries are weighted according to the BCWS that has been allocated to each package. This is depicted on Figure 5.

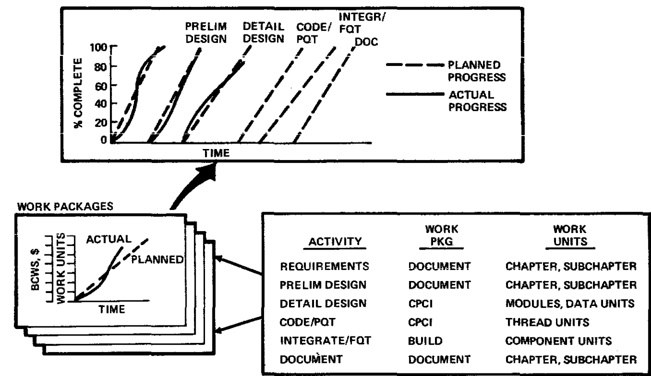


Figure 5—Work package contributions factored into composite rate chart

Work packages are generally defined around the natural products of the engineering process. During requirements and design activities, these products and work packages are documents (specifications, ICDs, test plans, etc.). Progress is planned and measured by allocating points or work units to subchapter and chapters of the document. Work units are accumulated as each subchapter is completed. During coding, the complexity units assigned to each thread serve as work units. In integration, work units allocated to each functional component are accrued as each component is integrated into the baseline system.

The schedule variance (BCWP minus BCWS) and cost variance (ACWP minus BCWP) is computed monthly for each work package and summed up through the work breakdown structure to evaluate overall project status. This performance measurement is supported by an automated tool, the Performance Evaluation and Measurement System (PEMS). PEMS receives and archives actual expenditures weekly, provides an interactive interface for scheduling and recording of accomplishments, and performs earned value assessment. PEMS outputs automatically produced reports that document earned value accomplishment and variances.

CONCLUDING OBSERVATIONS

The software engineering approach described here has emphasized auditable verification and validation events in each life cycle phase that are directed toward early detection of errors. These verification and validation mechanisms are summarized on Figure 6. This reflects a sensitivity to the escalating cost of fixing errors as a function of the time in the life cycle that they are detected as shown on the upper right of Figure 6. The software engineering development process that has been outlined here has emphasized parallel verification and validation in all the life cycle phases as a cost-effective approach to guarantee product reliability and contain life cycle costs.

These parallel verification and validation mechanisms are recapped briefly here. During requirements definition, the CADSAT analysis tool is used to verify interface relationships, while the system verification diagram verifies the functional requirements. The SVD is later employed to guide design verification, construction, and test/integration. Design

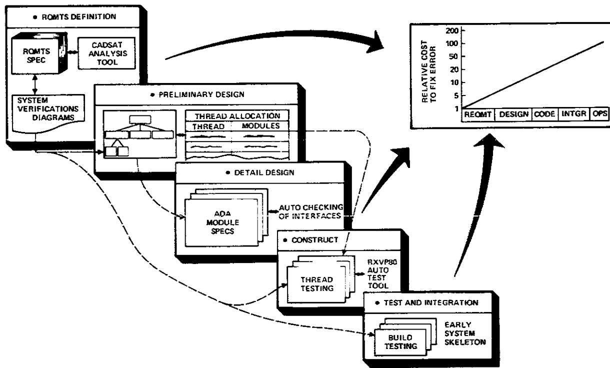


Figure 6—Summary of verification and validation mechanisms

verification is performed by "threading" the design modules against the stimulus/response pairings of the SVD and also by using the design quality metrics tool to evaluate design complexity. During detail design, an Ada PDL processor verifies interface relationships among the various modules by automatically checking Ada package specifications. In the construct activity, CPCI requirements are informally validated using threads defined by the SVD and assisted by the RXVP80 automated verification system to exhaustively test the software and maximize error detection at the thread level. CPCI and system requirements are formally validated during test and integration using the builds approach, guided by the SVD, and directed toward early establishment of a system skeleton.

An approach to the definition and implementation of a software development environment

by JAMES F. ELWELL
TRW Defense & Space Systems Group
Redondo Beach, CA

ABSTRACT

During the past two decades, a marked increase in software costs has been seen. The ingredients are now present to define and implement a software development environment which provides an increase in programmer productivity. The methods used by TRW to identify the goals of such an environment and define the components of the environment are discussed. The resultant TRW Software Office Of The Future is presented and its current status given. Observations relevant to this process are made.

INTRODUCTION

In the past two decades, we have both witnessed and participated in a remarkable expansion of the role of the computer in commerce and industry. For those of us associated with software, it has been particularly significant because during this time the cost for software has overtaken that of hardware (see Figure 1), as predicted by Boehm in 1973.[1]

Another consequence of this expansion has been the steady increase in demand for the trained computer professional—a demand that has not been matched by the university, which is the main source of new computer professionals.[2,3] This trend is expected to continue for the foreseeable future.

The cost implications are clear; unless something is done, software costs will continue to rise at a rapid rate. Increasing programmer productivity is essential to keep software costs down.

During the same period of time, the cost of the computer has shown a marked decrease. For example, in the early 1960's an IBM 7094 cost approximately $1.5 million, while today a computer with comparable power (e.g., a DEC PDP 11/70) costs $100,000.[4] This is the other half of Figure 1, and the prediction by Boehm seems on schedule. In addition to becoming cheaper, the computer is packing increasing power into smaller and smaller packages. In the past few years we have witnessed what can be described only as the dawn of the age of the microcomputer. The computer has now become accessible to the user, no longer requiring the specially equipped rooms of their predecessors. This accessibility, coupled with decreased cost and increased power, suggests that hardware which could be used to support software productivity gains is available. Computer professionals can now offload portions of their work to hardware, thus "automating" themselves as a way to becoming more productive.

Software expertise has also shown rapid expansion during this period of time. A brief summary of the evolution of software tool systems shown in Table I will exemplify this.

Each generation of these tool systems has provided significant productivity gains. For example, the switch from assembler code to a high-level language increased the number of machine instructions produced per source instruction by a factor of 5.[5] There has likewise been a notable advance in software development methodologies, the application of which tends to reduce the cost of software.[6,7]

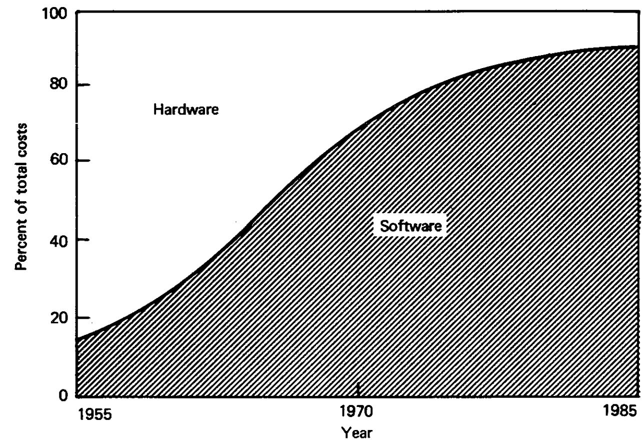


Figure 1—Hardware/software cost trends

The conclusion to be reached is that the ingredients are present to define and build a software development environment that supports programmer productivity gains. The motivation is provided by the increasing software cost and shortage of computer professionals. The means are provided by the decreasing hardware cost, increasing hardware capability, improved software tool system, and the evolution of software development methodologies. In fact, the development of this productivity environment is essential. In the remainder of this paper I would like to describe the steps taken at TRW to define and implement the TRW Software Office Of The Future. I will conclude with a brief status report and some observations which are relevant to this process.

TABLE I—Tool system summary

Time Span	Tool system	Example	Characteristics
1950–	Assembler	IBM map	Mnemonic code, macro capability
1957–	Compiler	FORTRAN, ALGOL	High-level language, functions
1963–	Operating system	System 360	Multi-language support, utilities, simple job streaming
1968–	Time-sharing system	IBM TSO, TRW TSS	Remote job entry, collection of independent tools, multitasking
1974–	Software Environment	UNIX, ALTO	Integrated tool set

LONG-TERM GOALS

The first step in defining an environment that will provide productivity gains is to understand just what gains are desired. To do this, TRW established a Software Productivity Tools Working Group, whose charter was to analyze software tool usage at TRW, assess the current and likely future state of the art in software, and recommend actions leading to improved software productivity within the company. Although the Group concentrated on automated aids to software development, considerations of documentation, facilities, and procedures were central concerns throughout the study. The major conclusion of this study was that software tools are a critical part of an integrated productivity improvement strategy which could increase productivity on large software projects by a factor of two in 1985 and by a factor of four in 1990. In addition to productivity improvements directly related to tools (such as increasing tool usage, increasing interactive development and the avoidance of the cost of retooling), the indirect gains (such as personnel motivation, modern programming practices, tool system experience and stability) were considered.

To achieve these productivity goals, the concept of an integrated TRW Software Office Of The Future (TSOF) was defined. TSOF drew upon the best of environments that are currently defined or operational,[8-10] modified to meet the unique requirements of TRW. Its goals are to provide the following characteristics:

1. Hardware: low-cost, medium-power personal computer each with individual file-storage capability; high-capacity bus which connects the terminals in a local network to a medium-size mainframe (e.g., DEC VAX 11/780), which supports the network and various peripherals; a gateway from each of the local networks to a large-scale mainframe (e.g., CDC Cyber).
2. Software: an integrated tool set which supports the software development life cycle and is user-friendly, easy to use, portable, expandable, flexible, and secure.
3. Facilities: offices which are approximately 100 square feet with floor-to-ceiling walls and which provide adequate work space, storage, lighting and quiet for a single person.

Thus, the following set of goals was established:

1. Implement a TRW Software Office Of The Future
2. Increase productivity by a factor of two by 1985
3. Increase productivity by a factor of four by 1990

APPROACH TO REALIZING THE GOALS

The realization of the long-term productivity goals and the implementation of the TSOF require a significant level of corporate investment. To justify such an investment, an analysis of the effect of the expected software productivity gains on the cost of delivered software was made using the TRW proprietary Software Cost Estimation Program (SCEP). This showed that the productivity goals were achievable.

A further justification was made on the basis of cost. An investment of $15,000 per programmer to provide the facilities and hardware and software tools that will double (or quadruple) productivity is returned the first year they are in place. This return on investment comes from either the value of the additional software which may be built or from the reduced cost of the production of the same amount of software.

The development approach to the realization of the goals was based on three principles: (1) The resultant environment would be built in increments, (2) data would be collected to use as a guide for the next increment, and (3) the environment would be used to support an ongoing project. This approach enables the builder to define and implement something in the near-term, measure progress, then introduce new or improved hardware and/or software, and measure again. Thus a cycle of building, evaluation, and improving is established. The use of the TSOF to support an ongoing project provides a practical focus for the work.

IDENTIFICATION OF NEAR-TERM OBJECTIVES

The result of establishing long-term goals and defining an approach to their realization was the establishment of a Software Productivity Project (SPP), whose charter is to effect those two charges. As a practical matter, the building of the TSOF is a project that will take many years. A set of near-term objectives is therefore necessary to direct the work on a more manageable basis. For 1981, work proceeded based upon the following six objectives:

1. Get something up and running in 1981. This "something" was defined to be Increment One. As with any new concept or project, it is important to show results, particularly to a sometimes skeptical audience. This objective also provides an impetus to the work, as does the support for an ongoing project. The builder has promised, and the project needs some results now.
2. Increment One must be a foundation for the future. What constitutes the foundation clearly must be identified. This foundation must process at least one characteristic; the ability to accommodate change must be built in.
3. Increment One must provide some support for all phases of software development. In addition to the usual phases of software development (e.g., requirements, design, code), tools for managers and necessary support functions (e.g., documentation) must also be provided. Some tools take a long time to build. If they are not started early, they won't be ready by the time they are needed, say in the test phase.
4. Increment One must be amenable to change. With the limited time scale for Increment One, both the hardware and the software will be a subset of that which is ultimately desired. Succeeding increments will add features to that which is built in previous increments. Changes will also occur because of technological innovation and as a result of the measurement process. Ability to accommodate these changes must be built in.

5. The total system aspects of Increment One must be emphasized. Currently, most tools are independent.[11,12] With some exceptions, each was built with little or no knowledge of the other. Productivity gains accrue from the elimination of duplication of effort and the intrusion in the work of one person by another. Tools, then, should be "systematized"; i.e., the incidental information generated in the performance of one task which is valuable to another should be available to the second tool.
6. Certain aspects of potential productivity gain are deferred to later increments. The funding and time limits placed on SPP required the exclusion of some facets of the software and hardware world. In particular, the security aspects of the system (except those that are part of the foundation), graphics, and CAD/CAM were not to be considered as part of Increment One. The gateway from the local network to other networks was also excluded.

CHOOSING THE TEAM

The charter given to the Software Productivity Project (SPP) required building a team with a wide range of talent. This mix of talent is important in providing a perspective on the needs of the potential users of the resultant system and also in providing the expertise necessary to implement the system. Thus, individuals with software, hardware, and facilities expertise who also possessed managerial, project and/or line organization backgrounds were sought. Representatives from various support areas such as configuration management, data entry, and secretaries also participated in the definition and implementation of the TSOF.

Team members were also chosen because they possessed the qualities of creativity and experience, the former because it is essential for the pioneer, and the building of the TSOF is truly a pioneering effort; the latter because each individual is, in effect, a representative of a class of users. A high level of experience is required to effectively represent these users.

There must also be a careful balance between the academics and the practitioner. Both are necessary because they provide complementary talents, but one group cannot dominate the other without serious consequences for the project. With too many practitioners, an inflexible, one-purpose system may be built; while with too many academics, a sand-box project and perhaps even no system at all could result. In this case, the tie to an ongoing project is a great help, because real deadlines require real results.

The team selected for the SPP consisted of 15 full-time and 6 part-time people. Included in this group were 4 individuals with PhD's and 10 with master's degrees. The average amount of experience was 9 years. The team was organized as shown in Figure 2.

Organization is strictly along functional lines, with each functional area responsible for research and development expertise in its area. The system engineer has overall responsibility for the conceptual integrity of the system, its testing, and performance.

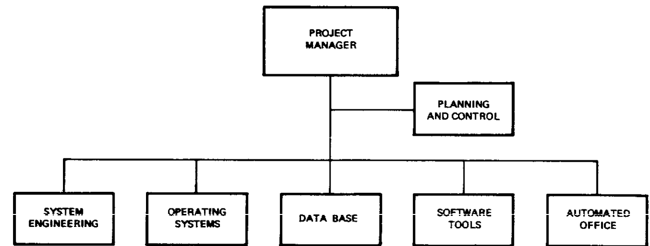


Figure 2—SPP organization

For the team to be effective, each of the individuals must not only be committed to goals of the project, but also be willing to work in an open, cooperative environment. This is accentuated by the unusually high level of interaction between each of the functions.

SYSTEM DEFINITION

With computational capability available directly to users from two different sources ((1) the medium-size central computer, in this case a DEC VAX 11/780, which supports the local network and (2) the personal computer/terminal), the first task of the system definition process is to partition the work. In the TSOF, the personal computer/terminal initially will be used mainly to off-load the word processing and text editing from the central computer. If a tool were able to be operated on both computers, it would be available on both. Thus, the central computer becomes a repository for the text files used to support a software project, the machine on which the large tools are run, and the storage point for the project database. This partitioning has certain hardware implications, the most important one being the requirement for a high-capacity bus to support the rapid transfer (ideally almost instantaneous) of files between the central and personal computer.

The second step in defining the system was to identify the tool set which was to be built. This tool set is dependent upon the software development methodology used at TRW. To define this tool set, the activities required to develop software according to TRW standards were identified. Then a set of tool types which could be provided to support these activities was defined. The result was a matrix of 110 activities and 25 tool types. To choose the tools to be built in Increment One, an "x" was placed at the point at which a tool type supported an activity. For the tool types which supported the greatest number of activities, the actual tools (if they existed) which provided that capability were surveyed. For this survey, the following criteria were applied to aid in the choice of the tools to be built:

1. Consistency with the near-term objectives: The ability of a tool to satisfy the near-term objectives was scrutinized. The tools which satisfied the greatest number of these objectives were retained as candidates.
2. Availability: If a tool which passed Test 1 was available and porting to the TRW system was reasonable, it was given preference as a candidate.
3. The sum total of the resources required to either build or

port the tools chosen must be less than our allocated budget.

Clearly this was an iterative process.

Ten tools were chosen as a result of this process. These tools are described below.

Automated Unit Development Folder (AUDF)

This tool maintains an electronic representation of a software Unit Development Folder Cover Sheet[13] and provides functions for the addition, modification, and deletion of items on that sheet. Each of the nine sections of a UDF is represented, and an Associated Information Pointer is included for each section to provide access to the text content of the UDF.

PDL

The Program Design Language is the software package developed by Caine, Farbe and Gordon, Inc. This provides a structured method for the design of software.

Automated Office

The Automated Office consists of a set of facilities to enhance the communications between people in an office environment via electronic means. These facilities include electronic mail, calendar, document templating, word processing, text editing, and UNIX automated office functions.

Front-End Help

This tool provides an initial high-level view of the tool set through two levels: (1) a menu-driven conceptual view of the collection of tools available and (2) a help capability for each available tool which describes the tool's nature, documentation, person-in-charge, and invoking command.

FORTRAN 77 Analyzer

This tool provides an automated system for the analysis of FORTRAN programs, including those written in ANSI 77 FORTRAN. Analysis data is provided at three levels on the static structure of the code and on the dynamic structure indicated during program testing. It is useful as a code auditor, test effectiveness measurer, and general software development aid.

Software Requirements Engineering Methodology (SREM)

SREM is a set of tools and a technique for defining and analyzing software requirements.[14,15] The technique is built upon a language, Requirements Statement Language (RSL), readable both by computer and by person. The set of tools is collectively termed the Requirements Engineering and Validation System (REVS) and is used to analyze the requirements specified by the user for completeness and display input and output in a variety of ways.

Automated Test Manager (ATM)

ATM controls and supports the testing of FORTRAN software units. This includes the specification of test case inputs and expected outputs for the unit, the generation of a test driver for the unit, and the execution and analysis of one or more units using the generated test driver.

Relational Database Access (RDA)

RDA allows the user to modify a relational database in a user-friendly way. RDA prompts for inputs, provides help messages, validates the inputs, and provides default values. RDA is designed for a user with some knowledge of relational database and enables such a user to add, delete, modify, or show tuples in a relation.

Requirements Traceability (RT)

RT allows the user to trace requirements through software design and test. RT generates reports such as the test evaluation matrix and exception reports.

Distributed System Design (DSD)

DSD supports the designers of software development projects utilizing distributed computer architectures. By providing a central location for the design of hardware and software elements, and the interfaces between them, DSD facilitates communication among software designers, systems engineers, and database administrators and encourages an integrated design.

The third step was the choice of the operating system. There were two systems considered: (1) UNIX as distributed by the University of California at Berkeley and (2) VMS, the standard DEC operating system for the VAX. This was not the first time a choice of operating systems was made between these two alternatives. In fact, it seems that this choice has the power to arouse passionate debate.[16,17] To resolve the problem, an evaluation was performed to determine which of the systems could best support the needs of the large TRW project which would be the first user of the TSOF. A list of 38 capabilities was developed, and the ability of each system to support these needs was evaluated. The major conclusions were that neither system supplied all the project's needs and that each possessed features needed by the project that the other did not. Moreover, with some augmentation, each could support the project. After satisfying ourselves through an industry survey of UNIX use that UNIX could support the software development of a large-scale real-time system, UNIX was chosen. It was chosen primarily because it offered the best approach to achieving the long-term goals of portability, flexibility, conceptual integrity, etc., which were defined for software.

During the process by which the tool set was defined, it was noted that a centralized database was an essential feature of any system that would emerge. Accordingly, a list of ten

candidate database management systems (DBMS) was prepared and, given certain criteria, evaluated. To provide a foundation for future expansion and to provide for flexibility in use, a relational database was deemed superior to a hierarchical database. The choice was further narrowed by the choice of the UNIX Operating System. Again, the industry was surveyed to determine user experience with the now candidate DBMS's. The result of this was the choice of INGRES as the DBMS to be used by the productivity system. In addition to being the best rated DBMS, the potential to port INGRES programs to a machine, the Britton, Lee IDM-500 existed. The prospect of off-loading a large software data processing activity to hardware was very appealing.

Thus far, only the software component of the TSOF has been considered. There are two more segments which were also studied and defined: (1) the hardware components of the system and (2) the office facilities. There are three major hardware components which were chosen: (1) the bus for the local network, (2) the personal computer, and (3) the terminal. The evaluation of the potential candidates for each of these components was performed in early 1981 and has already been dated by the rapidly changing technology of the industry. The results of this evaluation are shown on Table II.

Of more interest is the resultant hardware configuration, which is shown in Figure 3.

The final component of the TSOF is the office facilities in which the programmer would work. After surveying the existing facilities in industry and the universities and visiting some of them, notably the IBM Santa Teresa facility and Xerox PARC, the basic goals for the office facilities evolved. The TSOF would be a single-person office with a closeable door and floor-to-ceiling walls. It would be self-contained; i.e., each office would be connected to the network and have sufficient power, lighting, and air conditioning to support the potential system hardware configurations. Communication with others would be via the terminal over the network and by a telephone. Sufficient internal space would be provided so that two people could meet in any office.

The IBM Santa Teresa facility[18] strongly influenced both the design and the criteria of the office facilities. However, whereas IBM built a building which embodied their ideas, the TRW buildings were already in place. This necessitated adapting certain criteria to reality; e.g., not every office could be 100 square feet because of existing building constraints. The resultant office characteristics are shown below:

1. Facilities—Each office will be from 80–100 square feet with floor-to-ceiling walls, a closeable door, rug, walls coated with sound absorbent material and lights inlayed in an acoustic ceiling.
2. Furniture—Two chairs; approximately 30 square feet of work surface, a portion of which would be at proper terminal height; wastebasket; and two-drawer security cabinet.
3. Storage—Approximately nine linear feet of hanging files and fifteen linear feet of shelf space in close proximity to the work surface; space for tape and disk storage must also be provided.

Table II—Hardware components

	Model	Manufacturer	Characteristics
Bus	SPP	TRW DSSG/S&SO Redondo Beach, CA 90278	• 150K characters/second per channel (minimum) • 50 channel capability • 100 to 300 users per channel • Throughput —960 characters/second (terminal to terminal or terminal to computer) —25,000 characters/second (file transfers between computers)
Computers	VAX 11/780	Digital Equipment Corporation Maynard, Mass.	• Medium scale 32 bit computer (4 M bytes RAM) • Two 250 M byte disks • 30 users • UNIX/Operating System
	DYNASTY	Tymshare/Santa Cruz Operation Santa Cruz, CA	• LSI 11/23 based microcomputer (250 K bytes RAM) • 30 M byte Winchester disk and two 8" floppies • 4 users • DYNIX Operating System (Derived from UNIX)
Terminals	VT100	Digital Equipment Corporation Maynard, Mass.	VT100 or VT100 emulator: • a multitude of keyboard selectable options, e.g., —2 scroll speeds —scroll/no scroll button —80/132 column selectable format —print white on black or black on white —throughput up to 1,920 characters/sec
	VISUAL 100	Visual Technology, Inc. Tewksburry, Mass.	
	CIT 101	C.ITOH Electronics, Inc. Los Angeles, CA	

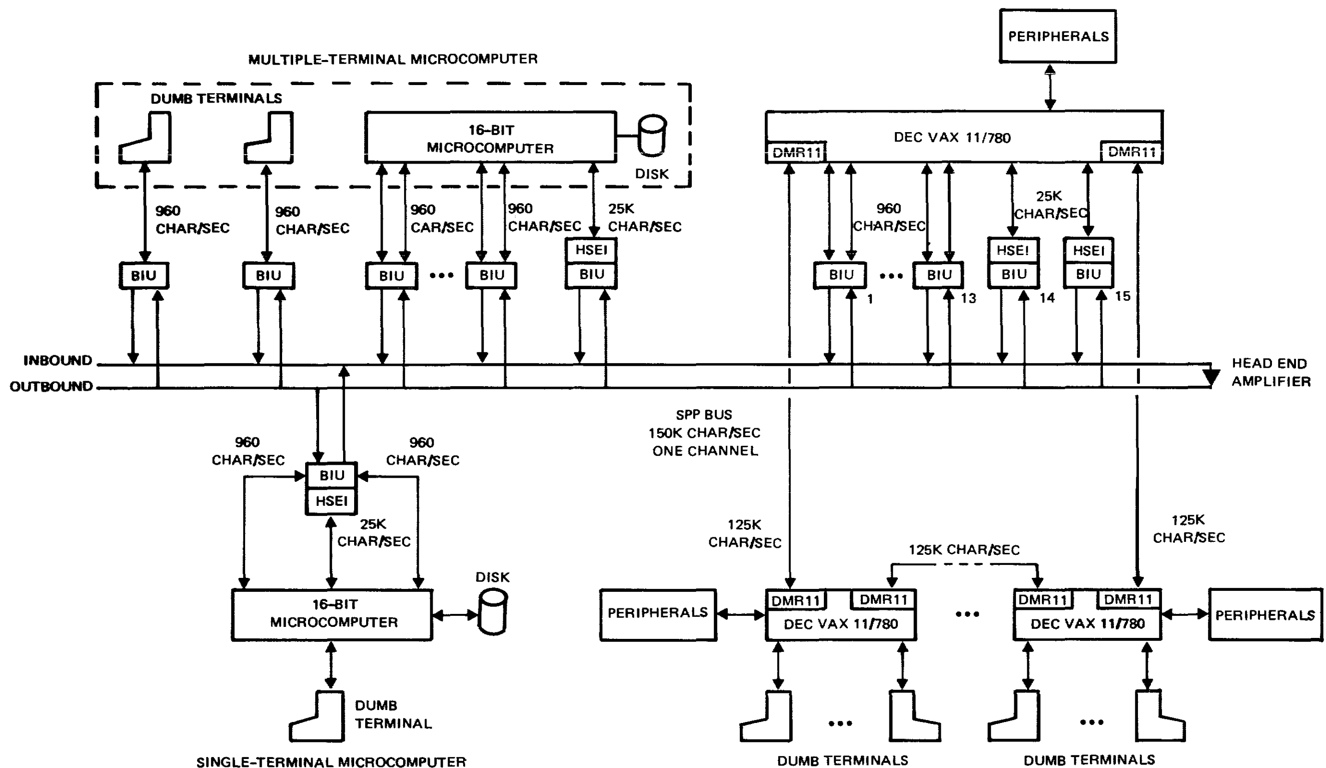


Figure 3—SPP network configuration

4. Lighting—A task light over the work surface in addition to industry-standard overhead lights.
5. Ambience—The office should offer a pleasant work atmosphere and, where possible, be comprised of ergonmetrically correct components.
6. Communications—Built-in wide-band bus connections.

To satisfy these criteria, it was necessary to have some of the components custom-designed. By working with various vendors, a full-scale prototype office was built and furnished. A picture of the resulting office is shown in Figure 4. A full-time programmer was assigned to work in this office as a test of the configuration. The feedback from this experience was used in the design of a much larger area, which would be built to house the contingent of programmers who would be the first users of the TSOF.

MEASUREMENT

One of the most important features of the TSOF is the ability to gather feedback from and measure the performance of the users. By measuring the performance, improvements to the system can be identified and their effects quantified. Measurement also provides the hard data which is necessary to convince management of the need to continue their commitment to this concept because it is providing a return on their investment.

There are two types of metrics which will be measured. The subjective metrics will explore the attitudes and impressions of the users toward the TSOF. Such impressions can be used to rate the morale of the user, identify the best features for reinforcement, and spot weaknesses for improvement. The objective metrics which will be used are: time sheets, which will be used to determine work patterns; system accounting; delivered source instructions; and cost model ratings for the TRW SCEP Program. The latter metrics will be used to provide an object measure of work activity and productivity.

The data will be gathered in the following ways: automatically as the users perform their work, and subjectively, by interviewing the users, by filling out questionnaires, and by observation. Only the approach to performance measurement has been done this year. The actual performance measurement and analysis is an effort for 1982.

STATUS

At the time of this writing (November 1981) an area of approximately 6,000 square feet has been configured in two equipment rooms, two secretarial bays, and 37 offices. Construction of this area is complete. The office furniture for this area has been designed and built and is beginning to arrive and be installed.

Designing, building, equipping, and working in a prototype office during the year was completely successful. The feedback received from the occupant and from the many visitors to this office was used to improve the design of the subsequent offices.

The hardware components, with one exception, have been ordered, have arrived, and have been installed in temporary locations pending completion of the offices. Although the full

Figure 4—Typical office

local network is not yet available, valuable experience has been gained in using the personal computer/terminal in a stand-alone mode. In this mode they are able to support virtually all their assigned functions, the major exception being the transfer of files between the personal computer and the central computer.

The schedule for the delivery of the software tools, again with one exception, is being met. Some of the tools are in operational use on the central computer, others are in integration and test, and one, the Automated Office, is operational on both the personal and central computer.

OBSERVATIONS

The project to build the TRW Software Office of the Future has had a successful first year. However, like all projects, there were deviations from the original plan, oversights, and things done so right that they bear pointing out because of their potential value to others who attempt similar work.

The first observation to be made is that the project expanded virtually from the time it started. The building of the TSOF was a relatively new idea within many areas of TRW, and everyone wanted to have the final product *today*. Because of this pressure, every facet of the project expanded in scope; the tools had more capability than originally planned, more hardware was required to support this expansion, and the size and diversity of the user group was increased. The staff, too, contributed to this expansion with a wealth of ideas and just plain hard work. All of this is good, but it is easy to lose control of the activities and thereby lose your way. The incremental development approach which was used on the project was very helpful in collecting these ideas and channeling this pressure along constructive, controlled paths.

The next observation to be made is that as different groups of potential users view this system, they feel that it rightfully belongs to them and should reflect only their needs. The system is flexible enough to accommodate that view. In particular, it is difficult to resist tailoring the goals and objectives of the project to those of the first user. With different areas having different aims, control of the project becomes difficult. To mitigate this problem, a high-level program office was established to oversee the direction of the project. This central point of both control and contact with the different parties served to coordinate the work. The main advantage of this centralization is that it provides an opportunity to build only

one system, thus saving the cost of duplication if each of the areas built their own.

The importance and scope of the training necessary to support the use of the tools provided by this project was grossly underestimated during the planning phase. It is difficult to overstate the critical importance of a well-planned, well-rehearsed, thorough series of training sessions. This training must address psychological issues which are not part of the usual training class. After all, the user is being asked to accept a new method of doing business which changes, in some cases radically, the previous method. Training must provide the user not only with technical skills to use the tools but also with the psychological will to use them. The truth of this became apparent well into the project's life, and a crash effort to provide the many courses required by an enlarged user group was instituted. The problem was complicated because the user group was no longer homogeneous. Programmers require different training than, say, managers, and the courses must accommodate that difference.

It is essential to have the firm commitment of upper management to a project such as this. Without it there is no project. With it, it is possible to attract the high quality of staff required to do the work, little effort is used finding funds for equipment, and positive direction comes from high levels. Fortunately, we have this commitment at TRW.

There seems to be real benefit in tying the development of this software environment to the needs of an ongoing project. The system being developed is flexible by design. The potential for working a problem to death to find the "right" solution is great. (It may well be that there is no one "right" solution, but several. This suggests that what needs to be built is a system that is easily configurable to each individual's needs and idiosyncracies.) Having a project depending upon the completion of tools to meet their deadlines serves to focus the discussion so that results are produced more rapidly. It is worth the risk of tailoring the system too closely to the needs of the project.

Finally, one of the most fruitful areas for increasing productivity seems to lie in building an integrated system. In this type of system, actions which are incidental to performance of one job are captured and become the essential data needed for another. To date, most of the tools which have been built are independent, one being unaware of the existence of another. To build such an integrated system requires extensive front-end analysis to understand the interrelationships between the tasks being performed and the tools being provided. By establishing tool standards early in the life of the project, individual tools may be built which fit both within the integrated system and support one another.

ACKNOWLEDGMENTS

The author wishes to both acknowledge the help given him by J. Hurvitz, M. Imura, P. Bogle, M. Green, and D. Nunez in the preparation of this paper and to thank them for it. Except for the figures, the manuscript of this paper was prepared using the facilities of the system described.

REFERENCES

1. B. W. Boehm. "Software And Its Impact: A Quantitative Assessment." TRW Software Series TRW-SS-73-04, May 1973.
2. "Science and Engineering Education for the 1980's and Beyond." Prepared by the National Science Foundation and the U.S. Department of Education, October 1980.
3. P. J. Denning. "Eating Our Seed Corn." *CACM,* Vol 24, No. 6, June 1981.
4. T. A. Dolotta, et al. "Data Processing in 1980–1985." John Wiley & Sons, 1976.
5. F. P. Brooks, Jr. *The Mythical Man-Month.* Reading, Massachusetts: Addison-Wesley, 1975.
6. A. I. Wasserman. "Toward Integrated Software Development Environments." Tutorial: Software Development Environments, 1981.
7. D. L. Paster. "Experience With Application of Modern Software Management Controls." Fifth International Conference on Software Engineering, March 1981.
8. R. A. Beach, et al. "Software Productivity Tools Working Group Report." TRW memo, October 1980.
9. "Requirements for Ada Programming Support Environments 'Stoneman'." Department of Defense, February 1980.
10. "UNIX Time-Sharing System." *The Bell System Technical Journal,* Vol. 57, No. 6, Part 1, July–August 1978.
11. "Software Tools: Catalog and Recommendations." Applied Systems Design Section, TRW DSSG, January 1979.
12. "NBS Software Tools Database." Center for Programming Science and Technology National Bureau of Standards, October 1980.
13. F. S. Ingrassia. "The Unit Development Folder (UDF): An Effective Management Tool for Software Development." TRW Software Series, TRW-SF-76-11, October 1976.
14. M. W. Alford. "A Requirements Engineering Methodology for Real-Time Processing Requirements." TRW Software Series, TRW-SF-76-07, September 1976.
15. T. E. Bell, et al. "An Extendable Approach to Computer-Aided Software Requirements Engineering." TRW Software Series, TRW-SF-76-05, July 1976.
16. D. L. Kashtan. "UNIX and VMS Some Performance Comparisons." SRI International.
17. W. Joy. "Comments on the Performance of UNIX on the VAX."
18. G. M. McCue. "IBM's Santa Teresa Laboratory-Architectural Design for Program Development." *IBM System Journal,* Vol. 17, No. 1, 1978.

A JOVIAL programming support environment

by EDITH M. McMAHON
Computer Sciences Corporation
Falls Church, Virginia

ABSTRACT

Programming support environments are developed to provide facilities to be used in addition to, or in absence of, host operating systems. This paper describes a system that incorporates Stoneman principles and provides a transportable Programming Support Environment for the Air Force standard language JOVIAL. The tools provided by this Jovial Programming Support Environment facilitate both software management and development throughout the life cycle.

INTRODUCTION

The Communications Software Development Package (CSDP) is being developed to be used as an automated, portable JOVIAL J73 programming support environment that enhances the production of high-quality software. Programming support environments are being developed and used by both program managers and software implementers. These environments provide tools and comprehensive support for different high-order languages, such as Ada, C, and JOVIAL. The concept of an environment to provide supporting facilities in addition to, or in the absence of, host operating systems has been definitized in Stoneman.[1]

Because of the diversity of operating systems (OS) and the time involved in learning a new OS, a need arose for a transportable system to allow users to shorten the learning curve necessary when transferring to a different computer system. The user would then interact only with the host system during the initial login procedure, after which he no longer communicates with the host directly.

Once fully implemented, retraining requirements for personnel transferring between sites is minimal. CSDP has been designed to be a transportable JOVIAL programming support environment in accordance with the Stoneman design guidelines.

This paper examines the CSDP project as a programming support environment that is also being used to support its own development. Following a brief overview of CSDP and a description of how CSDP has been developed to satisfy the design guidelines enumerated in Stoneman, the functionality of CSDP and the support that it has provided during its own development is provided.

OVERVIEW OF CSDP

CSDP is designed to be used on a large mainframe to develop software for embedded computer systems that do not normally have development facilities of their own. CSDP supports the Air Force standard language, JOVIAL J73. CSDP consists of a set of methodologies and tools to support software development throughout the entire life cycle, a Project Support Library (PSL) for managing development systems, and a user interface for invocation and utilization of both the tools and the PSL.

CSDP is designed and structured to provide standard techniques which will enhance all phases of software development with respect to project management, as well as implementation at the design and programming levels.

To fully realize the benefits of language commonality, a common interface to the host environment is required. This permits programmers to move from one host system to another, continuing to employ the same development tools and user interface.

The key element of the CSDP system design is the perception of CSDP as an abstract machine providing services through automated tools to users through an automated interface. The overall structure of CSDP is modularized, with subsystems being separate and independently replaceable. System tools can be added or deleted as desired, and users may create and use tools privately developed in either the command language or a high-order programming language.

CSDP is comprised of five subsystems as illustrated in Figure 1: Shell, Tool Manager, Tool Kit, Project Support Library (PSL), and Environment. A brief description of each of these subsystems follows.

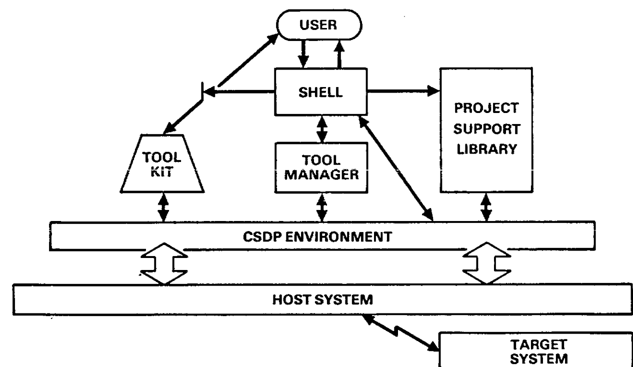


Figure 1—CSDP subsystems

The Shell subsystem functions as the interface for both interactive and batch users. Through the Shell, the user specifies which tools are to be executed and may pass arguments to the tools to control their execution. The tools to be executed may be executable object code or they may be Shell Command Language (SCL) files. An SCL file is composed of standard and frequently used command sequences, consolidated into a single file for repetitive use. The Shell allocates space for any variables which a user may find necessary to declare and use during an interactive session. In addition, the Shell provides a help facility and a statistics collection capability.

The Tool Manager, not visible to the user, interfaces with the Shell, providing the Shell with tool and help file location and access information.

The Tool Kit, one of the most visible subsystems of CSDP, is an expandable set of software tools. All current tools are implemented in JOVIAL, although tools need not be implemented in JOVIAL to be accessible to CSDP users. Capabil-

ities provided by various tools include text manipulation, file management, implementation of software, and development of documentation. It should be noted that the JOVIAL J73 compiler and linker/loader are not incorporated into the current CSDP. These facilities are provided on the host system.

The PSL, which provides primary support for configuration management, performs the following basic functions: management of software documentation, version control, and documentation and file revision statistics gathering.

The Environment subsystem serves as the interface between the other subsystems, the tools, and the host operating system. This subsystem is the critical factor in achieving portability for CSDP; it is the sole machine-dependent subsystem. In cases where the host and target are the same, a subset of the Environment routines provides input/output and file management capabilities for JOVIAL programmers.

CSDP AND STONEMAN

The Stoneman report[1] defines requirements for an Ada Programming Support Environment (APSE) as a set of tools supported by Kernel APSE (KAPSE) functions to provide database and host machine interfaces. A Minimal APSE (MAPSE) is composed of a KAPSE augmented by a minimal set of support tools.

The CSDP design incorporates many of the Stoneman precepts and provides for the basic functionality for a programming support environment as defined in Stoneman. Figure 2 illustrates the relationship of CSDP to Stoneman. Level 0 in the figure represents the host system. The KAPSE functionality is supported in CSDP by the PSL and Environment subsystems. Tool interfaces are provided by the Tool Manager through invocation by the Shell, which incorporates the command language interpreter. Also within the Tool Kit, which corresponds to the formation of a MAPSE, are text editor, text formatter, PDL formatter and file utilities. All of the

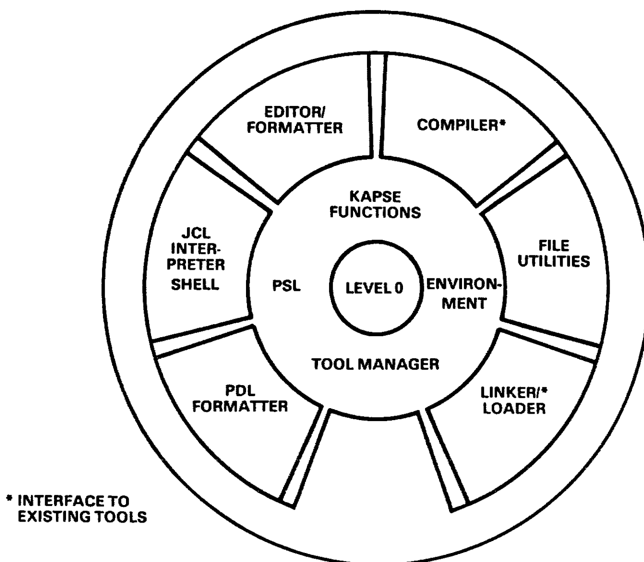


Figure 2—CSDP and Stoneman

tools, the PSL, and the Tool Manager are portable portions of CSDP, although the PSL is made more efficient through consideration of host capabilities.

The Stoneman report identified design guidelines to be used in the development of programming support environments. A discussion of these guidelines and CSDP's conformity to them is given below. Additional discussion of these concepts was presented in "CSDP as an Ada Environment."[2]

1. Scope: CSDP provides development and maintenance support for the development of software for embedded computer systems.
2. Quality: The technologies upon which CSDP is based have been in use for several years at Bell Labs.[3] UNIX has been proven to be very effective and useful in system development for a variety of different mainframes. All the tools that are incorporated in CSDP have been used and proven successful for software development.
3. Simplicity: CSDP has a simple overall structure, involving a user interface (the Shell), a tool kit, a tool manager, and a software database. The command language provides three basic facilities—a standard procedure for tool invocation, a standard method for passing arguments to tools, and programming constructs for building multipurpose command sequences.
4. Life Cycle Support: The Tool Set and PSL have been configured to provide coordinated development support throughout the entire development life cycle. This aids in requirements definition, design, implementation, testing, and operations and maintenance.
5. Project Team Support: All functions required by a project team, including management control, documentation, and configuration control, are provided by the PSL and the Tool Set.
6. User Helpfulness: The CSDP user interface (the Shell) is easy to learn and use. It also provides a help facility for the user.
7. Uniformity of Protocol: All tool interfaces to the host operating system have been isolated in the interface package. The Shell provides standard interfaces between both users and tools and among tools in the same way. The uniformity of the system aids not only in implementation but also in ease of use and transportability.
8. System Portability: CSDP is designed to be as transportable as possible. The major portions of the system are implemented in J73, with all operating system specific functions isolated in the Environment subsystem. When these Environment routines are reimplemented on another host, the rest of CSDP can be transported without changes.
9. Project Portability: Projects developed using CSDP on one host system can be moved to another host that supports JOVIAL J73.
10. Hardware: CSDP has been designed to use low-risk technology; therefore, it can be used on a large number of operating systems with little difficulty and requires only basic support of the JOVIAL language. Each tool has been chosen to be as efficient as possible within the constraints imposed by its function. The number of

actions necessary to invoke a tool has been kept to a minimum. The low-level Environment routines will be made as efficient as possible, although their efficiency depends greatly on what services the host operating system provides.

11. Robustness: CSDP provides meaningful diagnostics to its users in the case of error.
12. Integrated: Inter-tool communications are handled through the Shell, and all tools can access the PSL as a common database.
13. Granularity: New tools can be added from within CSDP as well as without. New tools can be written in the supported high-order language, or can be implemented as a procedure in the Shell Command Language. The new tools are invoked in exactly the same way as standard CSDP tools are from the Shell. The use of existing tools to build new tools is restricted only by the constraints set on the tools a user can invoke.
14. Open-Ended: CSDP's design supports changes and expansion as new requirements arise. The overall structure is modularized, with the Shell, the PSL, the tools and the Tool Manager all being separate and independently replaceable. System tools can be added or deleted as desired, and users may create and use tools privately developed.

CSDP FUNCTIONALITY

CSDP provides a standard interactive interface for JOVIAL users, with a comprehensive set of tools available to software managers and developers. The following paragraphs describe how the subsystems of CSDP and the tools provide a full range of support for both management and development of software throughout the life cycle.

Management support

During the initialization of a project, the project manager allocates all user file storage areas. The project manager is, therefore, distinguished from all users in the project and has full access to all user directories and their files.

The PSL is a repository of information connected with each project throughout the project's life cycle as well as a collection of basic commands and functions for the manipulation of the PSL. Options for tailoring the configuration management program to fit the needs of each particular project are available in the PSL. This includes the specification of file types which will be subject to version control. File types include source, object, relocatable, and text. Each project may specify conventions for naming file types, and these may be selected for version control. The approval cycle for placement of files in the PSL and for making subsequent changes to those files is determined by the project and/or configuration manager.

Managers may examine statistics automatically collected with respect to tool usage. These statistics indicate the frequency of a specific tool's usage, help to pinpoint problem areas with any of the tools or their documentation, and aid in the identification of training needs.

All CSDP tools are available to managers. To determine current status of modules under development, a manager may use tools such as LISTF and PRINT to locate and examine the files of particular interest in users' directories. The tool LISTF identifies all files in users' directories. The tool LISTF identifies all files in a user's directory and provides various statistics for each file. Once located, any file or part of a file may be displayed at the terminal using the tool PRINT. The output from either of these tools may be stored in files, rather than output to the terminal, via the output redirection capability of the Shell, thus saving the information for subsequent examination and analysis.

In addition, tools such as EDIT and FORMAT, for text manipulation, are available to management for use in the preparation of project documentation and reports.

By using SCL files, management can automatically generate current reports which detail progress and statistics pertaining to a given project or a particular user. This type of routine report can be generated at time intervals controlled by project managers, thus enabling the charting of the life cycle progress of any project using CSDP.

Project-specific documentation also can be controlled by using the SCL to routinely update any information contained in a file. As an example, it is possible to routinely update portions of design documentation kept in files associated with the source to include the most current documentation, reflecting the addition or deletion of modules and changes made in module descriptions. This capability allows concurrent control of software and its associated documentation.

Programmer support

CSDP has been developed to facilitate the development of software in the JOVIAL J73 language on a DEC-20 System. CSDP also offers advantages and capabilities for software development in any language on any host, provided the language requirements (for example, a JOVIAL J73 compiler) are available to CSDP users.

CSDP acts as an interactive and user-friendly interface to the host. CSDP contains a wide range of tools and commands to facilitate software development and provides information messages pertaining to current use of CSDP and its subsystems. Additionally, CSDP provides a help facility to encourage full use of CSDP capabilities.

Access to all CSDP standard tools is given to all programmers. In addition, project-specific tools may be developed by project members. Such tools may be stored in a project tool library and made available to the project's developers. Duplication of effort can be avoided through the exchange of project-developed tools when appropriate. The CSDP Environment subsystem acts as an interface between CSDP and the host machine and therefore provides the interface between the JOVIAL programmer and the host machine. The availability of Environment functions for JOVIAL programmers provides for a more unrestrained usage of the JOVIAL language, since the Environment supplies facilities for disk-file management, input/output, and argument and error handling. These facilities enhance the early stages of software development as well as the testing and implementation phases.

Developers and managers are likely to find it necessary to use a sequence of Shell commands repeatedly. SCL files automate this process. In addition to the Shell commands used to execute tools, CSDP offers its users a programming-like language providing variables, decision making, and looping facilities. For example, using SCL file constructs, it is possible to create an SCL file containing test procedures that can be reinvoked in order to perform regression testing.

Batch job capability is provided so that the user is free to work interactively while concurrently executing time and resource-consuming jobs in batch mode. Users have the capability to check the status of batch jobs and to modify batch job execution sequences.

CSDP EXTENSIBILITY

CSDP can be tailored easily to fit user, management, and/or project needs, since specific tools may be added or deleted from the Tool Kit. This flexibility allows growth of a project-oriented tool library, while allowing maximum use of file-storage area through the deletion of unrequired tools.

The portability of CSDP affords the capability to achieve standard software development and management practices for all JOVIAL systems developed under it. CSDP supplies a uniform system with which to pursue JOVIAL programming and management endeavors.

CSDP is designed to be a transportable system. The Environment defines a virtual interface to the host-operating system in order to execute and support the portable CSDP subsystems. The Environment presents a consistent, uniform, machine-independent interface to CSDP and interfaces directly with the host system in its native language. This intentional isolation of host-dependent functions means that only the Environment need be modified in order to rehost CSDP on a new system. All of the other subsystems are implemented in JOVIAL.

For CSDP to function as intended, certain capabilities must be provided by the host system. These include a JOVIAL compiler, linker/loader, and a time-sharing environment.

In some instances it will be possible to ease the amount of effort by taking advantage of host-provided functions. For example, if the host system supports a directory concept as the DEC-20 does, then interfacing routines, rather than a portable directory system, reduce the amount of software necessary to provide this function.

The procedures for extending CSDP by adding a new tool or integrating an existing one are essentially the same. Accesses to specific services of the CSDP host (such as the file system) and to the CSDP framework (such as interprocess-communication mechanisms) are through standard interfaces to primitive routines (i.e., the CSDP Environment); therefore, all tool interfaces look alike to CSDP.

CSDP SUPPORTS ITS OWN DEVELOPMENT

CSDP is designed to support the development and maintenance of software through its entire life cycle and is being used in its own development to provide earlier operation and produce a system easier to use and maintain. This is manifested by the incremental Build approach used for the CSDP implementation. A Build is a group of related functions that forms an implementable subset of the system capabilities. A partial operational capability is provided with the first Build and is enhanced with each successive Build.

Developing software projects following the Build approach (a phased implementation) increases productivity and reduces risk. This incremental approach as applied to the CSDP development not only provided for early demonstration of the system's capabilities but also provided tools that could be used in the continuing development of CSDP.

Using this approach, CSDP was partitioned into a series of three Builds, each of which provided increasing functionality. Each Build consists of modules from CSDP subsystems, and the completion of each Build demonstrated additional capability of CSDP. This incremental development of the system enabled portions of CSDP to be used in later development and testing of the system.

The first Build consisted of the Tool Manager and modules from the subsystems interface (the Environment), the user interface (the Shell), and a subset of the complete Tool Kit, as shown in Figure 3. The virtual file system was provided along

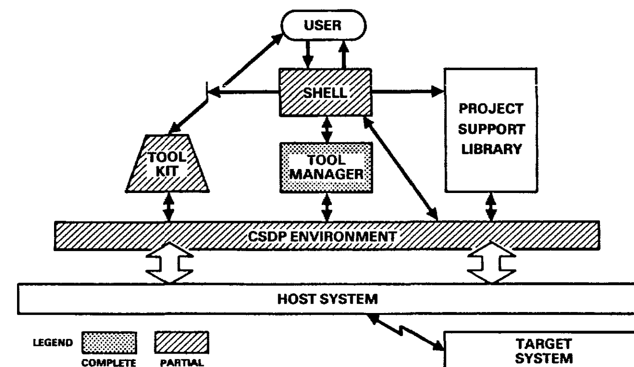


Figure 3—CSDP Build 1

with the capability to ready, suspend, and resume CSDP processes in this Build. The tools included the text editor and formatter, which were then used by the CSDP developers in the continued implementation of the project. This also provided an opportunity for more extensive testing of the tools and other CSDP capabilities through continued exercise of these capabilities. At the completion of Build 1, the CSDP implementers were able to work a minimal CSDP system, using it to develop the remainder of the programming support environment.

In the second Build, the modules from the first Build were expanded and new modules were added to enhance CSDP functional capabilities, as shown in Figure 4. A software database (the PSL) was made available for managing the developing system. System statistics collection function, job submission through batch mode, an interactive help facility, and Shell Command Language processes were implemented. In addition, the rest of the tools to support software development using CSDP were implemented in the second Build.

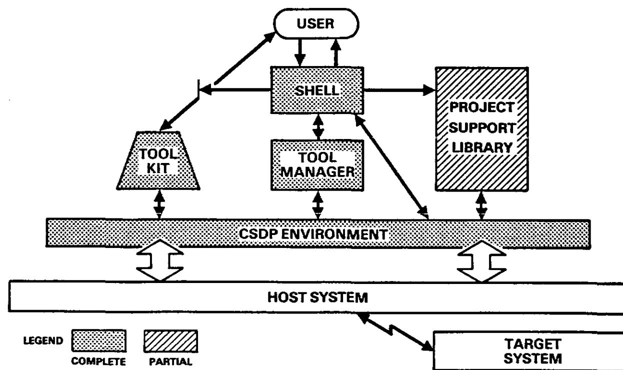


Figure 4—CSDP Build 2

The functionality provided by this second Build enabled even more effective and efficient production of the system. At this point, the Shell, Environment, Tool Manager, and Tool Set were complete and used in the completion of the PSL.

The third Build added the complete functionality of the PSL, including all configuration management support for version and history control. Stubs used in the earlier Builds were replaced, and the full capabilities of CSDP were realized.

CONCLUSION

The CSDP project provided an opportunity for the application of a software development methodology as well as providing automated tools to support the development. In particular, the Build approach to implementation was used to provide a minimal JOVIAL programming support environment used to complete the implementation of itself. That is, CSDP as a support environment was used in its own development.

ACKNOWLEDGMENTS

This work was supported in part by Rome Air Development Center under contract F30602–79–C–0051.

REFERENCES

1. Department of Defense, Requirements for An Ada Programming Support Environment—Stoneman, February 1980.
2. Allshouse, R. A., D. T. McClellan, G. E. Prine, C. P. Rolla. "CSDP as an Ada Environment." *Proceedings of the Ada Environment Workshop,* November, 1979.
3. Dolotta, T. A., R. C. Haight and J. R. Mashey. "The Programmers Workbench." *The Bell System Technical Journal,* Vol. 57, No. 6, Part 2, July–August, 1978.
4. Computer Sciences Corporation. "Communications Software Development Package (CSDP)—System/Subsystem Specification." July, 1980.
5. Fischer, K. F. and M. G. Walker. *Digital Systems Development Methodology.* Computer Sciences Corporation, November, 1978.
6. McMahon, E. M. "Software Development Environment at Computer Sciences Corporation—A Case Study." *Proceedings of the Fifteenth Annual Asilomar Conference on Circuits, Systems, and Computers,* November, 1981.

The impact of Ada on software engineering

by KENNETH L. BOWLES
TeleSoft and *University of California at San Diego*
San Diego, California

ABSTRACT

The term *software engineering* has traditionally been applied to extremely diverse activities, ranging from system programming to managing programmer teams. Ada appears destined to become the first widely used programming language designed to bring these diverse activities together in ways supported by both programmers and managers. Among many important aspects of the Ada language, the most important appear to be (1) its orientation to system construction using interchangeable building-block packages, and (2) strong standardization in the interest of program portability. These aspects should foster the emergence of a new kind of software component industry. A probable result will be an inversion of the traditional view of software as an added value for use on major hardware products. Instead, major machine-independent software systems will emerge, and hardware will be increasingly regarded as an added value.

BACKGROUND

Software large enough to require more than a month or two of design and programming effort and/or the collaboration of two or more programmers tends to be one of the most complex engineering activities undertaken by human beings. Often the complexity is illusory and ill understood by the sponsors or managers of a project—not to mention the project's participants themselves. The problems that result typically include the following:

1. Logical errors and unreliable program execution. Because computer systems are being used increasingly to control equipment or situations on which human life depends, the control and prevention of these errors is assuming great importance.
2. Much higher costs to maintain a software system than to create the system in the first place.
3. Great difficulties in project cost estimation and frequently spectacular cost overruns.
4. Frequent duplication of development efforts to accomplish tasks that are nearly identical on different machines or in differing application settings.
5. Divergence of the program behavior actually achieved from the behavior intended by a project's sponsors.

Characteristically, efforts to solve Problem 1 by increasing the size of one's programming staff results in retrogression because of the rapid escalation of required communication among the staff members.

Engineering is the art of applying relevant scientific knowledge to the solution of real-world problems. In the case of software engineering, the relevant knowledge includes both program design techniques and methods for managing and enhancing the work of groups of cooperating people. The two fields are related because the available program design techniques determine much of the style of interaction among the members of a design team. The Ada language design is the result of a major effort to distill the best current knowledge about programming language design into a single language powerful enough for use in embedded system design. In the second field a large-scale effort is under way to create a comprehensive set of software tools—called an Ada Programming Support Environment, or APSE—to serve both programmers and their managers. Both design efforts were initiated in the hope of alleviating the problems listed above.

Neither Ada nor an APSE will provide automatic solutions to the list of problems cited above. It will still be possible for ill-disciplined programmers to write erroneous and unreliable programs, and in many cases it will still be very difficult for managers to control the errors or enhance the reliability of products developed under their care. On the other hand, the combination of Ada and a disciplined programming style will make it easier for programmers to write correct programs. A good APSE will both enhance the productivity of the disciplined programmers and support effective management decisions affecting those programmers.

The Ada design has been criticized as too large and also somewhat dangerous for use in design of programs for which execution errors can be extremely expensive. The charge of excessive size has been debated extensively in the design process, and the result is an engineering tradeoff that will be tested as the language comes into widespread use. Ada draws its design philosophy from Pascal and from a large number of more recent research languages derived from Pascal. Standard Pascal has been found too small for practical software engineering, and most practical implementations embody a diversity of extensions to the language, thereby making most implementations nonstandard. The Ada design can be viewed as an effort to collect the most commonly demanded Pascal extensions into a single internally consistent language for practical system programming.

Whereas extended Pascal is unlikely to be covered by a widely implemented standard for many years, Ada is backed by a very strong standardization effort. All conforming implementations of Ada are expected to support the same language, with neither subsets nor subsets considered conforming. This approach is intended to permit the reuse of even large and complex Ada programs on many different machines and Ada implementations. Effective program portability was a major goal in the Ada design effort. To this observer, it seems unlikely that true portability can be achieved in a language intended for a very wide range of practical applications without making that language somewhat larger than ideal for any single application area. Portability will be achieved at the expense of a number of generally minor inconsistencies in the Ada design. Acceptance of the remaining inconsistencies is a tradeoff between the time needed to create a fully consistent language and the need to begin using a better software engineering language in the near future.

The charge that Ada may be dangerous[1] seems curious in view of the lack of any other widely accepted language designed to enforce program correctness. Ada permits handlers for run-time exceptions which, in Pascal, might cause abnormal full termination of a program. Much of the charge against Ada is based on the concept that a programmer should be forced to test explicitly for any and all potential abnormal conditions (at least if he/she wishes to avoid full program termination). It is true that sloppy use of Ada exception handlers could result in continued program execution while masking the existence of serious execution errors. On the other hand, disciplined use of exception handlers permits the main

(unexceptional) program flow to be more readable by eliminating the intrusion of a large number of special-condition tests. Careful testing of an Ada module should insure that all possible exception conditions are properly handled. The Ada design encourages enumeration of the various possible exception conditions in each exception handler. Concentration of the programmer's attention on exception conditions in one part of a small module should be an aid to comprehensive checking for all possible errors, but it is not an automatic solution to the problem. One objection to ADA exception handlers is that their behavior is still ill-understood in the context of automated proofs of program correctness. It is debatable whether such proofs will soon be well understood in complex real-time applications, with or without exception handlers.

PROGRAM MODULARITY—PROGRAMMER INTERACTIONS

Perhaps the most important style issue in software engineering is the manner in which large programs are broken into modular pieces. Questions of modularity arise at several different levels of detail, such as the following, starting with the most detailed:

1. Isolation of small program control sequences in such a way that only one entry point and one exit point are used. This is one of the two main ideas on which the structured programming movement has been based.[2]
2. Isolation of related data objects in named data structures definable by the programmer. Again a part of structured programming.[2]
3. The recognition of distinct algorithms which can be isolated in their own subprograms. Reasons for using subprograms range from a desire to improve program clarity through the hiding of unnecessary details to an avoidance of duplicated program sequences.
4. The separation of major groupings of routines and data from other parts of a large program or system. Objectives range from the need to cope with limited main memory to a desire to subdivide the design work among several programmers.

The key idea running through all of these is the hiding of details except in localized areas where they can be handled in limited quantity.[3] We human beings can concentrate on only a limited amount of detailed information at one time in our short-term memory. Other information, stored in our long-term memory, can be retrieved and actively used only at the expense of the displacement of other details from short-term memory. When concentrating on the overall structure of a complete system, or even on a subsystem, we must use abstract names and concepts to represent the details present at lower levels. When concentrating on a detailed level, we must put aside direct consideration of the overall structure.

The need for information hiding, and for isolation of modular groupings of details, arises not only because of the limits of human short-term memory, but also because of the high human communication overhead associated with the division of labor among several people working on a common project. Whenever detailed information needs to be shared by two or more programmers, an effort is needed to insure that all participants have the same view of those details. Any change in the common information by one programmer must be reviewed with the others—often leading either to changes by several programmers to accomodate the revision or to arguments about the best way to confront the new situation.

The Pascal and C languages are probably the most widely used prototypical implementations of the first three program modularity concepts enumerated above. Both suffer in the fourth area, groupings of routines, because efficient communication among such groupings requires the use of shared data objects in common global areas of memory. At best, the shared data objects must be the subject of continuing communication among the several programmers on a team. At worst, the use of shared data objects is an invitation to programmer errors that result from an overload of detailed information, afflicting both individual programmers and groups of programmers. Insofar as it deals with the first three listed areas of program modularity, the Ada language is very similar to Pascal.

For some applications, the UNIX operating system provides an effective solution to the fourth modularity problem through the mechanism of pipes. The standard text stream output of one small UNIX program component can easily be connected via a pipe to the standard text stream input port of another program. From two to many such programs can be connected via a single pipe. This leads to a highly modular style of programming, described by Kernighan and Plauger.[4] The UNIX pipe is a simple, easily described and understood abstraction for the interconnection of otherwise independent program components. All detail shared between these components is conveyed in the text stream passed via the pipe. To be useful, the logical structure of a text stream emitted by a program component must be well understood. The programmer writing a component designed to receive that text stream as input needs no other information about the emitting program. Indeed, many service program tools are designed to cope with text streams coming from a wide variety of sources, with no knowledge of details regarding those sources. The simplicity and generality of the pipe mechanism in UNIX permits the intermingling of program components written in C, Pascal, RATFOR (Rational Fortran, including various structured programming control constructs), and other languages. However, pipes are inefficient for many applications, and they often hide important information on data types that really should be communicated between modules.

Ada modules, i.e., both packages and tasks, are designed to meet needs associated with the fourth style issue listed at the beginning of this section. An Ada package is a collection of related subprograms, data objects, types, and constants, all of which can be separately compiled and stored in a library for later use. Most details regarding a package are hidden within the body part of the package and are not available for use or inspection by a using program. Only those carefully chosen details that the package writer wants used as an interface to the package are placed in the specification part of the package and thus made available to the using programs.

Many of the concepts that led to the design of Ada packages were drawn from research on abstract data types.[5] For exam-

ple, an Ada package can "export" a data type declared to be "private." This allows using programs to declare objects of that type but forces all operations on data stored within those objects to be performed by the package. Indeed, Ada permits alterations (and recompilation) to be made in the body part of a package without affecting the specified interface. Thus, details on possibly changing internal structures associated with the data objects can be isolated in the package—and are of little or no concern to users of the package.

Ada tasks are superficially similar to packages but are intended to run concurrently as semi-independent program components. The formal interface to a task is limited to a set of entries that have the appearance of procedure headings. Ada tasks generally are expected to serve other program components via calls to these entries. Data flow to and from a server task during the parameter passing that takes place in the rendezvous of the server with a client task.

The syntax of Ada module interfaces provides no guarantee that unnecessary details will be hidden to the maximum degree possible nor that the high overhead of programmer interactions will be minimized. Indeed, with very little effort it is possible for a team of Ada programmers to run into most of the structural problems that cause errors using earlier programming languages. On the other hand, it does appear that a style of programming can be developed to make use of Ada in such a way as to accomplish the following:

1. Minimize the interaction of programmers working on separate modules.
2. Maximize the possibility of independent changes for maintaining the bodies of separate modules, without affecting programs that use those modules.
3. Maximize the possibility that suites of test programs can be used to insure correct operation of each substantial module of a large system—and that human designers will understand the implications regarding the whole system.

Early experience in the use of Ada for design of large systems suggests that the necessary new style of programming demands substantially more advance planning before actual coding begins than does more traditional programming practice. The management school of software engineering has argued in favor of advance planning for some time. Early experience with Ada has made the benefits of careful design of module interfaces easily apparent to programmers. Though the effort to produce those interfaces in a consistent way is relatively large, the actual implementation of the underlying module bodies then turns out to be relatively simple. Moreover, the module interfaces are typically written at a level of detail easily understood by managers. As a result, the system granularity resulting from this style of Ada programming should lead to better programmer/manager interactions.

SOFTWARE COMPONENTS INDUSTRY

Most software products sold today are complete programs. The idea of a software components industry, or marketplace in which building-block software components are sold independently, was suggested by M. D. McIlroy.[6] Since that time a flourishing marketplace has developed in building-block system components on printed circuit boards for use with several popular interconnecting bus standards. McIlroy's idea was that a similar marketplace should be available for the interchange of software system components roughly equal in complexity to the board components. Implementation of that idea in connection with uses of the UNIX operating system within the Bell Telephone System (see, for example, Kernighan and Plauger[4]) appears to have been very successful. However, the interchange of UNIX program components among users in general has remained informal and largely noncommercial. As described earlier, interchangeable UNIX components are generally connected by pipes—a useful mechanism for transmitting streams of text between components, but of limited utility in many other applications. In spite of that limitation, the rapid acceptance of UNIX for the current generation of desktop work stations may well encourage the emergence of a commercial market in UNIX components.

In principle, Ada modules can be interconnected in a wide variety of ways, including pipes. In practice, a flourishing market in Ada program components will only grow if most implementers adhere to a small number of conventionalized interconnection designs. The software components industry will require common bus designs similar to the standard bus designs that support the printed-circuit-components industry associated with single-board computers (SBCs). A hardware system design based on SBC components represents a trade-off minimizing development time at the expense of optimized performance. Similarly, a software system composed of building-block components connected by a general-purpose interconnection design will usually be completed in much less time, but perform somewhat less efficiently, than a system constructed specifically for the application at hand.

Thus the economics surrounding the software components industry can be expected to be similar in this respect to that characterizing the SBC components industry. Frequently, the decision whether to use building-block components or a specific design will depend on the number of copies of the system projected for delivery to customers and on the time available for completion of the design. A small manufacturer of hardware systems generally chooses a specific design if the number of system copies is projected to be in the thousands. The higher investment made in the design, compared with the use of building-block components, is then more than offset by the larger spread between manufacturing costs and prices paid by customers. If the expected number of identical system copies is only in the hundreds, the design cost per copy is a relatively large part of the potential sales price, and a design using commercially available building-block components becomes preferable. This suggests that system integrators will be willing to pay per-copy royalties for the right to distribute hundreds of copies of building-block software components. For projected duplication in the thousands (or more), integrators either will prefer to buy full rights to the software components they use, or will choose instead to develop the equivalent software in house. If the current shortage of qualified system programmers persists, as projected, integrators are likely to buy rights to fully developed software components rather than choosing in-house development, except for small parts specifically related to the purpose of the final design.

Among the most important domains for interconnection of building-block software components will be that associated with user-defined data records. One of the strong principles behind the design of Ada (and before it, Pascal) is compile-time checking to insure that incompatible data objects cannot be intermingled without explicit conversion instructions by the programmer. To be generally useful, a library of modules designed for use with data records will have to cope with data records of many different user-defined types. The following list gives examples of frequently needed modules:

Sort and merge packages
B-tree record storage and retrieval packages
Index handler packages
Data display and data capture packages
Report writer programs

Ada offers an alternative to the common method of using program generators to provide library modules like these. A program generator maps a user's input of specifications into a complete program written in one of the widely used programming languages. The generated source program is then compiled normally, as if handwritten. This is a relatively clumsy method. Either a substantial repertoire of specialized program preparation tools is needed, or the user needs to master a substantial volume of detail to make use of a program generator.

In Ada, these modules will commonly be provided as generic library packages. The user-defined record type of information will be partially supplied as parameters passed to the generic packages when they are instantiated in the user's program. However, an Ada generic library package will not be given direct access to the individual data fields within a user-defined record. That access can be provided through a Record_Access package supplied by the user (i.e., by the programmer working as system integrator). Subprograms such as Store, Retrieve, and Compare will be provided by the Record_Access package interface for indirect access to the fields of a record. These subprograms will also have to be passed as generic parameters to each library package. Although this implies processing overhead for accessing each field, it is a method that allows hiding details on how the data fields are stored except with the Record_Access package itself. Indeed, several different mappings of the same data items within different record formats could be used with the same set of generic library packages without change.

It may be seen that specifications for the interface part of the Record_Access package will provide the common meeting ground for both writers of the general-purpose library packages and writers of specific Record_Access package bodies. These specifications will take the role of a software bus for applications in the record domain.

Easily understood interface conventions will also be needed in several other domains where building-block Ada components are likely to be used extensively. For example:

1. Message passing among the several layers of a handler for a communication protocol such as X.25 or Ethernet. Here the messages are likely to be passed as complete units rather than as character streams, as in a pipe.
2. Packing and unpacking of data objects in the fixed-size information "containers" of a paged virtual-memory management scheme.
3. Interface controllers for peripheral devices ranging from disks, printers, and CRT display terminals to laboratory instruments.

REFERENCES

1. Hoare, C. A. R. "The Emperor's Old Clothes." *Communications of the ACM,* 24 (1981), pp. 75–83.
2. Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare. "Structured Programming." New York: Academic Press, 1972.
3. Parnas, D. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM,* 15 (1972), pp. 1053–1058.
4. Kernighan, B. W., and P. J. Plauger. *Software Tools.* Reading, Massachusetts: Addison-Wesley, 1976.
5. Shaw, M. "The Impact of Abstraction Concerns on Modern Programming Languages." *Proceedings of the IEEE,* 68 (1980), 1119–1130.
6. McIlroy, M. D. "Mass Produced Software Components." Software Engineering, Report on NATO Conference, Garmisch, Germany (DBR), October 7–11, 1968.

The importance of Ada programming support environments

by THOMAS A. STANDISH
University of California
Irvine, California

ABSTRACT

In this paper it is argued that even if we assume the most optimistic scenario we can think up for the introduction of the Ada* language, the language alone, in the absence of an Ada Programming Support Environment (APSE), is insufficient to achieve the gains in programming productivity and software reliability with which use of Ada tantalizes us.

Moreover, it is argued that the level of support envisaged in the Minimal Ada Programming Support Environment (MAPSE), specified in the STONEMAN, which provides a rudimentary level of capability incorporating a text editor, compiler, linker/loader, and symbolic debugger, is also insufficient; and that it is time to seize the opportunity to conceptualize what sort of advanced programming support tools should populate a mature APSE of high utility and effectiveness. In this context, consideration of support tools for software project management, interactive programming, modern programming practices, software reuse, and improved program understanding techniques arises.

*Ada is a trademark of the United States Department of Defense (OUSDREAJPO).

WHY THE ADA LANGUAGE NEEDS AN APSE

Suppose that the most optimistic scenario we can dream up for the introduction of the Ada language actually comes to pass. Will this be enough for us to reap the benefits we are hoping Ada can provide?

In this paper, it shall be argued that the Ada language, considered as an isolated tool, cannot solve all of the problems of reliability, performance, and productivity that must be addressed if Ada is to succeed in realizing the high hopes some have for it. Rather, it is argued that Ada must be buttressed by powerful programming support environments that provide the means for performing a variety of essential tasks lying beyond the reach of programming languages.

Defining an Ada Programming Support Environment (APSE) as the collection of tools, resources, procedures, and policies that support the development, repair, and upgrade of Ada software, it is argued that even more than the rudimentary level of capability envisaged in the STONEMAN's Minimal Ada Programming Support Environment (MAPSE) is necessary if we are to achieve the significantly improved levels of programming productivity and software quality with which use of Ada beckons us.

Thus, the central question that the paper addresses is: could the Ada environment be even more important than the Ada language in helping to achieve the benefits we seek from the use of Ada?

First, for the purposes of setting an appropriate context for the subsequent discussion, let's conjure an optimistic scenario for successful introduction of the Ada language.

An Optimistic Scenario

Our optimistic scenario for the introduction of Ada consists of the assumption that we succeed in accomplishing the following steps:

1. There would have to be success in establishing precise, comprehensible standards for the definition of Ada. If we don't know what Ada means, we can't write certified compilers which implement the common meaning and provide a framework for exchanging Ada programs.
2. There would have to be success in writing certifiable Ada compilers which produce efficient, reliable running programs. (Note: the author is not so naive as to be unaware that the risk that we might fail in getting this far isn't entirely trivial; but the rest of the paper would be rather uninteresting if we don't assume we can reach at least a state of affairs where we have certified Ada compilers running on many computers of reasonable size. That is, if we lose the game by failing to define a standard and by failing to implement certified compilers that achieve performance and reliability, there is no point in discussing what must follow for the whole game to be won.)
3. If certified Ada compilers run standard Ada on most machines of reasonable size, the best we can hope for is that a framework will be established permitting a flourishing commerce of Ada programs. (Few people these days are so ill-informed as to think that having a precisely defined and implemented standard programming language will solve the program portability problem 100%. After all, programs are written containing dependencies on operating system calls, device characteristics, and other interface requirements that lie outside the scope of definition of a programming language. Nonetheless certification, standardization, and widespread implementation of Ada compilers could help reduce the cost of program transfer, since machine dependencies could be isolated in Ada packages with invariant external interfaces, and since some machine dependencies could be expressed using Ada representation specifications. This tells us that we might be able to reduce the cost of program transfer with Ada to a point at which it costs less to do it with Ada than with other approaches.)

To continue by optimistic speculation, if Ada becomes widespread in use, begins to function as a medium of exchange, and opens up a substantial market for the sale and exchange of programs, what response might one assume from the free enterprise system?

Perceiving that a wide market will exist for sales, here is a list of possibilities: (a) computer manufacturers would develop certified Ada compilers for new computers; (b) software firms would develop and sell Ada programming support tools; (c) publishers would publish books and educational materials on Ada; (d) educators would introduce and teach Ada in programming courses (perceiving that Ada, in addition to being popular and useful, could be a good carrier of modern programming principles); and (e) enterprises could get Ada programmers, Ada compilers, and Ada programming support tools in the marketplace and could import and export Ada programs.

To complete the optimistic scenario, we assume that some (but not necessarily all) of these economic consequences of the introduction of Ada take place.

Ada is Still Not Enough!

Even if an optimistic scenario such as this comes to pass, it is argued that the Ada language alone is not enough. More is needed. Here's why.

Wonderful as they are, programming languages play only a

small role in the software life cycle. It is estimated, for instance, that in software projects of substantial size, coding the design in a programming language accounts for only 15% of the total pre-release cost, and that the total pre-release cost may be only 10–30% of the total life cycle cost.

Furthermore, many activities in the software life cycle are supported by tools, procedures, or policies that are not directly connected with the programming language(s) employed by the project.

For instance, software project managers devise project schedules; manpower loading plans; milestone charts; and budgets for machine cycles, memory, and monetary resources. They monitor tasks on the critical path, report on progress and resource consumption, incrementally shift resources in response to perceived needs, and oversee the hiring and training of new project personnel. Recent evidence[1] suggests that upward of 80% of software project failures are software management failures as opposed to technical failures. Few if any of these management activities depend in any essential way on the choice of the programming language.

The system requirements and the system design may be expressed in natural language or in design representation notations separate from the programming language; and activities such as requirements tracing and design reviews may take place using notations, language, and procedures entirely separate from those given by the programming language.

Maintenance of current system documentation, module test sets, test completion status, and system configurations may depend more on database, word processing, and file system tools than on the programming language or on programming language support tools.

Let's take a glimpse at some software economics, for a moment, to try to establish a framework in which we can discuss the relative importance of trying to introduce various kinds of support tools and policies into a programming environment.

In the first place, the demand for computer instructions appears to be increasing rapaciously, and a serious shortfall of programmers to produce them exists in relation to demand.

For instance, the number of instructions NASA used to support the Mercury, Gemini, Apollo, and Space Shuttle programs has been growing at 24–25% per year for a couple of decades. While Gemini support took 1 million support instructions, and Apollo took 10 million, the Space Shuttle now takes 40 million. Most of the Space Shuttle instructions support ground launch and pre-launch check-out procedures and were designed to avoid the necessity of employing a ground launch support crew of 20,000 people. What is true for NASA appears to be true for the economy in general; namely, automation is being employed to avoid inefficient, labor-intensive production, and computers are being used to enhance product versatility and market appeal. Thus, the overall demand for computer instructions appears to be increasing in the neighborhood of 10% per year in many industries, and the national demand for computer instructions may, in general, be growing somewhere near 20% per year.

However, the supply of programmers is increasing perhaps only in the neighborhood of 5% per year, and programmer productivity has been falling! During the 1960s when high-level languages were replacing assembly languages, programmer output (in delivered instructions per person year) was increasing at perhaps 8–11% per year; but recently, the annual increase has been estimated to be in the range of 4–5% per year.

In short, given the poor prospects for increasing the output of new programmers from the educational system, there may be no alternative but to increase software productivity if the demand for production of computer instructions is to be met and if the shortfall in programmers will be a condition we will have to live with.

How then do we address the problem of increasing productivity? One approach is to analyze the cost drivers that correlate with the cost of software projects. In his new book, *Software Engineering Economics*[2] Barry Boehm introduces the COnstructive COst MOdel (COCOMO), which is a good fit to a database of measurements on 63 software projects spanning a range of different application areas. Briefly, one starts with a baseline estimation formula such as

MM = 2.4(KDSI)**1.05

giving an initial unadjusted estimate of the number of man-months (MM) to complete a project as a function of the number of thousands of delivered source instructions (KDSI), and one multiplies by coefficients that determine whether the estimated man-months will increase or decrease as a function of measurable software project characteristics (which can be thought of as cost-drivers). The ratio between the best increase and worst decrease in productivity for each cost-driver forms a *productivity range.* Examples of such productivity ranges are as follows:

1.20—programming language experience
1.32—turn-around time
1.49—software tools
1.51—modern programming practices
1.57—applications experience
2.36—product complexity
4.18—personnel/team capability

Software Productivity Range
(from cover of Boehm[2])

Some of these cost-drivers are controllables. That is, by investing to provide software project resources or by following certain project disciplines, we can control factors that enhance productivity.

For instance, we could invest in good programming support equipment to give programmers excellent turn-around time. We could provide good software tools. We could train programmers to use the programming language well. We could adopt modern programming practices as a software project discipline, and we could attempt to select programmers with proven track records and applications experience, if possible. The cumulative effect of these measures on productivity could be very dramatic (e.g., factors of 4, 8, or 10 could be achieved using the short list of measures just given), and these could easily dwarf any effects of choosing to use Ada or not.

In summary, we see that software productivity may depend heavily on the characteristics of the environment employed

and not so heavily on the characteristics of the programming language employed.

What is true for software productivity may be true to a lesser but still significant extent for software reliability and software performance.

Software performance will obviously be influenced critically by whether or not it is possible to compile Ada source programs into compact, fast-running object programs. However, performance may also be influenced critically by whether or not the Ada Programming Support Environment provides effective tools to perform measurement and optimization. Frequently, upwards of 90% of the execution costs are attributable to 7 to 10% of the code. Identifying and optimizing the critical sections has been found to be an effective way to improve performance. If the name of the game is measurement and tuning, both the programming support environment and the compiler must work together to provide the solution, with the environment furnishing performance measurement tools and the compiler providing optimizations. Where compiler optimizations are insufficient, the environment may make the key difference by providing source-to-source program improvement transformations or by making manual program rewriting more manageable and systematic.

Where software reliability is concerned, the programming language can play a key role in promoting reliability. Proponents of Ada have argued that Ada will promote reliability because it supports clean module interfaces and information hiding (through packages) and because it permits clear expression of control and data (through exceptions, tasking, and an extensive data type system). Opponents have argued that Ada programs may not be reliable because the language is too complex or may have ill-defined interactions between its features. But we have all known reliable programs written in unreliable languages and unreliable programs written in reliable languages. Promoting reliability may have more to do with assuring clean designs and thorough testing than with the characteristics of the language in which the program is written. It is the environment, not the language, which must provide tools and disciplines to perform design, design review, and testing.

Thus, it could be that the reliability of Ada programs will be more dependent on the characteristics of the Ada Programming Support Environment and the programmers who write them than on the characteristics of Ada itself. In summary, the success of Ada in promoting software reliability, performance, and productivity depends critically on the characteristics of APSEs. While Ada is clearly *necessary* for success, even under the most optimistic scenario, Ada alone is *insufficient.*

A BRIEF VIEW OF THE STONEMAN PHILOSOPHY

The STONEMAN[3] requirements document for APSEs specifies three levels of structure: (a) a KAPSE or Kernel Ada Programming Support Environment, (b) a MAPSE or Minimal Ada Programming Support Environment, and (c) the APSE itself. In a nutshell, the KAPSE provides basic operating system services and database capabilities and is intended to provide a machine-independent set of kernel services on which APSEs may be built. The MAPSE provides minimal Ada programming support services such as: (a) a text editor, (b) an Ada compiler, (c) a linker/loader, (d) an Ada debugger, and (e) a command language interface (for logging on, calling tools, manipulating files, etc.).

The STONEMAN philosophy, expressed in its so-called "strategy for advancement," envisages that the KAPSE can be implemented as a standard operating system kernel on many machines to provide a standard foundation for APSEs. If the KAPSE could be standardized and expressed as an Ada package, then all the KAPSE services and capabilities could be made available to Ada programs, and a major deterrent to program portability could be overcome.

The MAPSE, if successful, could provide a means for using Ada as a systems programming language for implementing not only Ada applications programs, but also the tools that constitute the full APSE. Thus, one could get going by supplying a rudimentary Ada programming environment (the MAPSE), and one could bootstrap out of the rudimentary environment into an advanced APSE by using Ada as the systems programming language for populating the APSE with environment tools. Such tools could be compiled, loaded, and run on top of the MAPSE to form a highly portable APSE.

APSE tools would thus be supplied in an Ada library available as a companion to the MAPSE. Current design efforts (the Army's ALS or *Ada Language System* and the USAF's AIE or *Ada Integrated Environment*) focus on providing the MAPSE-level capability specified in the STONEMAN but do not call for design of full APSE toolsets.

While STONEMAN provides some guidance on what APSEs must support effectively (such as maintenance and configuration management), STONEMAN does not attempt to present an extensive or very refined view of how to populate an APSE.

At the moment, therefore, an important opportunity exists for conceptualizing what an advanced APSE should contain.

Thus it is important to do our homework on what a full APSE should look like, and a number of important targets of opportunity come to mind.

POSSIBLE TARGETS OF TECHNOLOGICAL OPPORTUNITY FOR APSES

Interactive Programming

Although it is hard to cite credible experiments that demonstrate that interactive programming is more productive than batch programming, some experiments suggest an improvement of roughly 33% if interactive programming is used in place of batch.

Good interactive languages, such as APL and LISP, permit sophisticated and powerful actions to be taken by programmers while interacting with their programs. For example, at a point of suspension of a running program, a user at a terminal can do such things as: (a) print formatted values or texts of defined procedures; (b) define new procedures or assign new values to new variables; (c) perform queries ("Where am I?, Who calls this procedure? Who can read and set this variable?"); (e) resume program execution at the point of sus-

pension (or at other valid points of control); (f) call for explanations to be printed from online manuals; and (g) set breakpoints, traces, and performance measurement probes.

It is rare that such interactive services are available to the user of a high-performance systems programming language, such as JOVIAL, CMS-2, BLISS, C, Pascal, or MESA. In order to support queries and incremental changes characteristic of interactive programming, programs must usually be represented in a somewhat elastic (and thus incrementally updatable and explicitly queriable) representation. Usually this implies that program representations must be interpreted to be executed. On the other hand, to get high performance, programs must usually be compiled into rigidly efficient machine code. Such machine code does not conveniently support incremental editing in source program terms, and it usually does not contain symbolic information discarded by compilers yet needed at run-time during interactive sessions to reply to user queries in source program terms.

To the author's knowledge, nobody has succeeded satisfactorily in providing the combined advantages of compiled systems programming language performance with the power of interactive language query and incremental change. There may be a considerable technological challenge in providing this kind of support for Ada (or for any other high-performance systems programming language for that matter).

Management Support

If the evidence suggests that upward of 80% of software project failures are management failures and not technical failures, and that these failures result, in large measure, from ignoring software practices of proven effectiveness, what might we do to support project management so it can avoid well-known pitfalls?

Might we have online management interviews at the time a project is being organized to remind managers about software practices of proven effectiveness and to enable them to select thorough, effective project disciplines well-matched to a particular organization's characteristics?

What sort of management support tools might help managers devise project schedules, estimate resources required, make required reports, track project activities (monitoring especially the activities on the critical path), and adjust resources incrementally to fit changing needs?

Programming Methodologies?

Should an APSE support a programming methodology? If so, should it try to support a standard one and encourage its use? For example, should an APSE provide for use of an Ada-based program design language (PDL) together with some sort of discipline for design composition, design review, and requirements tracing?

Any suggestion that an APSE should support a standard programming methodology usually engenders heated opposition and dire warnings about the evils of premature standardization, and the points about such evils are usually well-taken. However, it may be possible to phrase the policy on the use of such methodologies in order to overcome most of the objections. One might say, for instance, "Here is a recommended methodology which is provided in the APSE library, and here are its abstract characteristics: (a) it provides a clear, comprehensible design representation, (b) it is accompanied by effective ways of getting design review by independent teams, and (c) one can determine which design modules are responsible for implementing which items of the system requirements, and so on." An RFP might then specify the following: "You can propose either to use the recommended methodology or you can propose to use your own, but if you choose to use your own, you should give some justification as to how using your own meets the essential abstract characteristics of the recommended one."

Software Reuse

Since the biggest cost-driver in software projects is the size of the software, any method that permits successful reuse of software to implement portions of a system dramatically increases productivity. Although the idea of software reuse has been around for a long time, and although it works in limited application areas (typified by well-defined interface and composition paradigms and by libraries of useful, well-indexed, well-explained components), we do not generally build software by assembling catalogued, prefabricated components. Getting software reuse methods to work as a general program composition technique may involve surmounting challenges such as finding ways to reuse designs and higher-level program abstractions and finding how to generate concrete refinements of the abstractions that meet the extraordinary variety of concrete usage constraints encountered in practice. Nonetheless, the payoff for finding an effective software-reuse technology would be dramatic, especially if performance measurement and certification were performed on all components entered into a component catalogue.

Program Understanding

Software maintenance accounts for 70 to 90% of the cost of the software life cycle for many large, long-lived systems. If, as recent evidence suggests, upward of half of the software maintenance time is devoted to trying to understand how a program works and what the effects of a proposed alteration would be, the activity of trying to understand programs and the effects of incremental program changes could be a dominant cost-driver in the system life cycle.

If this is the case, there might be an inviting technological target of opportunity in trying to devise ways of making it vastly less expensive and more effective to go about understanding programs. We might ask the following questions:

1. How can we write comprehensible program descriptions?
2. Is paper a good container for program documentation, or can we do better by using a computer to store explanations appropriate for different intended audiences and by computing various appropriate views for the different audiences?
3. Given a projected software lifetime (and other appropri-

ate unit costs), what level of capitalization is appropriate for developing program explanations?

4. Are there any techniques for "program archaeology," wherein, if we are confronted by an undocumented or poorly documented program, we could systematically go about trying to develop an understanding of it and whereby we could estimate the cost of doing so ahead of time?

Advanced APSE Tool Sets

What kinds of tools could an APSE provide the system builder? (Unfortunately, there is a great variety of answers to this question, and space does not permit the author to do more than provide a pointer or two to the literature. Two good sources that provide a variety of views and excellent bibliographies are Hünke[4] and SIGSOFT.[5])

RISK AREAS IN APSE DEVELOPMENTS

What are some of the risk areas that confront the development of APSEs? Since this is highly speculative, the author prefers to give just two areas where he perceives risk:

1. *No KAPSE Standardization:* What if the KAPSE never gets standardized? The KAPSE in STONEMAN is envisaged as a machine-independent Kernel operating system and database support system. If it can be standardized (with a machine-independent interface, given, for example, as an Ada package), one can write machine-independent Ada programs which call on KAPSE services in a standard notation (much as package Standard functions in Ada now), and such Ada programs will transfer to every machine on which an Ada compiler interfaces to a standard KAPSE. A special case of program transfer of great interest is a full APSE with tools written in Ada and depending on KAPSE services for support. Thus, KAPSE standardization holds the key to the machine independence of APSEs and to providing a powerful conduit for portability of Ada programs and environments. If the KAPSE cannot be standardized, can the market for APSE tools, which depends on having a viable method for the exchange and portability of Ada programs, ever become an effective reality?
2. *Too Little and Too Late:* What if some of the thirty or so current efforts to write Ada compilers succeed and serious Ada programming begins before MAPSEs and APSEs can be designed, built, and used? If serious Ada programming begins starting with a compiler, does one not then tend to use the available tools in the *de facto* environment surrounding that compiler (meaning the text editors, file system, linkers, and so forth), and do not critical dependencies then develop which inhibit program transfer to other different environments (with other different file system and operating system conventions)? To what degree is the timeliness of APSE development a critical factor in its possible success?

CONCLUSIONS

In conclusion, this paper argues that the benefits some seek for the introduction of Ada cannot be realized effectively without also introducing advanced APSEs that provide capabilities well beyond the STONEMAN MAPSE level. Furthermore, the time is ripe to do our homework on what a full APSE should look like, and a number of inviting targets of technological opportunity present themselves.

ACKNOWLEDGMENTS

This work was supported by the Defense Advanced Research Projects Agency of the United States Department of Defense under contract MDA-903-82-C-0039 to the Irvine Programming Environment Project. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

REFERENCES

1. Boehm, Barry. "Software Engineering as it is." 4th International Conference on Software Engineering, Munich, September 1979.
2. Boehm, Barry. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
3. Buton, J.N. and Druffel, L. E. "Requirements for an Ada Programming Support Environment, Rationale for Stoneman." *Proceedings COMPSAC 80,* Chicago, Ill., October 1980.
4. Hünke Horst. *Software Engineering Environments.* North Holland, Amsterdam, 1980.
5. SIGSOFT. NBS Workshop Report on Programming Environments, *Software Engineering Notes,* Vol. 6, No. 4, August 1981.

Challenges and requirements for new application generators

by ALFONSO F. CARDENAS
University of California, Los Angeles
Los Angeles, California
and
WILLIAM P. GRAFTON
Continental Airlines
Los Angeles, California

ABSTRACT

The need for application generation as an alternative to the aging programming languages (COBOL, FORTRAN, PL/1, Pascal, etc.) is put forth. Major technologies that participate in the movement toward application generation are reviewed. A variety of technical problems that plague these individual technologies are identified. The problem of lack of standards in the participating technologies is discussed. A satisfactory application generator (AG) should synthesize the overwhelming variety of services and individual technologies into a cohesive whole. The desired characteristics of an AG are indicated. Last, the resistance of programmers, analysts, and other specialists to embracing a new level of application development beyond handcoding in programming languages is indicated.

INTRODUCTION

The last fifteen years have seen a gradual introduction of new techniques in application development. High-level procedural languages have largely replaced assembler languages for application code. Complex operating systems have assumed most of the computer system management tasks, freeing application programmers to concentrate on solving business problems. Standard sort programs, subroutine packages, utility programs, access methods, and such have solved common, recurring problems once and for all.

Generalized file-management systems have simplified report programming and data manipulation for simple to moderately complex applications. Generalized database/data-communication systems have made it feasible to put a whole enterprise on line. Data-dictionary/directory systems have offered the potential for storing the definitions, relationships, and usage of all data in an organization. Database modeling technology permits an organization to be described in terms of its need for and use of information.

Software development practices have emerged under the banner of software engineering. Structured programming, modular programming, top-down design, and so on are some of the development methodologies that have been advocated as enhancing software development productivity. However, the actual productivity enhancement realized has rarely been in terms of orders of magnitude.

The Basic Approach Remains Unchanged

These innovations, while encouraging, are merely refinements of a basic approach to application development that has remained unchanged since the early days of data processing.

We still analyze requirements, conduct a feasibility study, identify and evaluate alternatives, select a solution, prepare cost and schedule estimates, obtain budgetary and staffing approval, design the external and internal system specifications, write and check out detail procedural code, publish program and system documentation, conduct exhaustive testing, train the user and operators to run the system, and finally implement. By this time the environment has often changed, the schedule has slipped, the budget has been exceeded, the user is unhappy, the DP staff has turned over, and we are ready to begin again.[1,2]

The Basic Approach Is Unsatisfactory

A strong case can be made for the proposition that this approach is unsatisfactory in several important ways:

1. *It is not responsive.* The process just described generally requires months or years to produce a usable application system. The time required from the recognition of a user requirement until a computer-based solution is available is perceived as excessive and unacceptable by many users, and the data-processing organization is universally regarded as being unresponsive to user requirements.
2. *It handles changes poorly.* The basic application-development process is inherently weak in its ability to handle changes in design. The further along in the development cycle an application project progresses, the greater the impact of a change. Environmental changes, misunderstanding of problems, new or improved ideas, and errors in communication will inevitably lead to the need for changes; the result is rework, schedule slippages, cost overruns, and frustration. The response of data-processing organizations has been to lengthen and formalize the front end phases of the process in an attempt to obtain specifications from the user that amount to a signed contract.[3] While this may help in avoiding unnecessary changes and may assist in fixing responsibility for changes that do occur, it does not address the underlying problem.
3. *It results in a backlog of undeveloped systems.* The process is so labor intensive, and the skills involved are so scarce and so expensive, that most organizations simply cannot put together an application-development staff of the quality and quantity needed to meet user requirements. This results in a backlog of undeveloped applications, failure to exploit the potential of the computing system, and a general dissatisfaction with data processing.
4. *It produces systems that are difficult to maintain.* The difficulty of change noted above does not only impact development; it follows a conventionally developed system throughout its life. Maintaining production applications occupies a major part of a data-processing organization's time. It is often estimated as requiring 70 to 80 percent of programming resources. The culprit seems to be the procedural coding technique, which requires human definition, in extreme detail, of exactly how the application is to be processed. Designing ease of maintenance into a procedurally coded system requires time, talent, and foresight, and these resources are usually in short supply in a development project.

A New Approach Is Necessary

What is needed is a new capability that will overcome the problems and break the logjam in application development and maintenance. Application systems should be produced directly from their specifications without the need for pro-

cedural coding and lengthy testing. This capability is of course the subject under discussion—the application generator (AG). The call for automatic production of applications directly from their specification is not new. Various authors and efforts in the past have worked toward this.[4, 5, 6, 7, 8, 9, 10]

CURRENT APPLICATION-GENERATION TECHNOLOGY

Under the broad category of application-generation technology we can identify the following major technologies and emerging product lines.

Turnkey Applications Software

A variety of proprietary software packages are available, usually from software houses (for example, most of the bread and butter business applications such as payroll, order entry, inventory control, and invoicing).

Turnkey applications usually offer only a limited range of customizing options in the final application. The range of features is very fixed and is mostly limited to cosmetic changes (e.g., names of fields in files but no changes in the structure of the files). Significant changes required in the application necessitate hand coding via "user exits" or recoding of the code "generated." Thus, turnkey software cannot be classified as a full-fledged application-generation technology.

As a consequence, the prime clients for such software are organizations that, since they do not wish to develop their own special systems, are willing to adjust or even change significantly their usual systems or procedures to fit the turnkey software. A small company today can obtain a set of related turnkey applications for as little as $15,000 to $25,000.

Self-customizing Applications

Self-customizing applications or application customizers have the characteristic that the *structure* of the programs that make up the applications can be changed significantly to fit a customer. It is not a matter of just changing a few parameters or making rather cosmetic changes that may be required by a client. It is the capability of allowing significant structural changes that is required. This is a fundamental capability of self-customizing applications.

The changes are to be done automatically by the customizer without programmer intervention at the programming-language level. A few customizers have been reported in the literature: for example, IBM's Application Customizer Service (ACS) for System /3 environments,[11] Distribution System Simulator,[12] and "programming-by-questionnaire" efforts.[6]

Most special-purpose languages fall in the category of self-customizing applications. For example, there are GPSS and SIMSCRIPT for discrete system simulation and CSSL and CSMP for solving ordinary differential equations.

Generalized File-Management Systems (GFMS's) and Sophisticated Report Writers

This technology excels in report writing, sorting/merging, and related tasks, usually against existing files and databases. In contrast to turnkey and application-customizing technologies, this technology is not application specific, that is, it can apply to completely different application areas, such as sales/marketing analysis and real-time spacecraft data analysis. The GFMS adapts itself to the application by means of detailed data definitions and high-level, nonprocedural instructions provided by the user at execution time (e.g., "select all invoices greater than 6 weeks past due"). Prime examples of such technology are Informatics' Mark IV, ASI's ASI-ST, and Cullinane's Culprit. Experience has shown that GFMS's provide productivity gains over using conventional programming languages (COBOL, PL/1, FORTRAN, etc.) of on the average 8 to 1 in a significant percentage of the bread and butter applications of an organization. By productivity is meant the elapsed time and person/days required to do the job. GFMS's are mostly used by medium to large organizations with a major investment in EDP. There are probably from 3000 to 4000 installed GFMS's.

Query Language Processors

This technology has emerged with generalized database management systems (GDBMS's) to provide high-level and intelligent data retrieval and update (write, modify, delete) capabilities over a database. Query languages are transaction/record oriented as opposed to GFMS's, which are batch report/full file oriented. They are particularly effective in online, interactive applications not requiring the data volumes or complex report writing, sorting/merging, and so on where GFMS's excel. Without a query langauge, a nonsimple request for data from a database would require a number of statements in the programming language and in the procedural data manipulation language of the GDBMS (e.g., DL/1 in IMS, DML in CODASYL GDBMS). The more intelligent query processors, and in particular those furnished with the newer relational GDBMS's such as IBM's QBE[13] and SQL/DS,[14] automatically generate the commands that would otherwise have to be hand-coded by a programmer-user to obtain the answer. An example of a nonsimple data request that illustrates this point is the often-quoted "list the names of all employees whose salary is more than that of their managers, and list also the corresponding manager names."

Application Development Systems (ADS's)

This is a nebulous area of more recent activity than the previous technologies. ADS's are attempting to automate some of the tasks involved in developing more advanced applications involving online, screen-dialogue, and database environments. Specific examples are the American Management Systems Generation Five (possibly),[13] IBM's Application Development Facility,[16] Cullinane's Application Development System,[17] and Informatics' Mark V.[18] These systems provide a variety of tools that have shown significant gains in programmer productivity in developing applications for a database-communications environment.[19,20] Libraries of precanned code for frequently used database I/O, screen formatting, auditing, and so on can be invoked by the application developer, thus reducing total programming time and effort.

Thus far, these new systems have been developed for use with specific GDBMS and data-communications facilities.

Other Generators

Data-dictionary/directory systems (DDS's) have emerged recently as central tools in a database environment. Descriptions of data residing in databases, of application programs, of users involved, of the terminal network, and so on are concentrated in the DDS. The DDS can then generate the record descriptions and file descriptions (FD's) for application programs, as well as schema and subschema descriptions in the language of the particular GDBMS to which the DDS interfaces.

A number of other less encompassing code-generation aids that do not fit well in the previous categories have been developed and are in some use.[8] Two of these are

1. Decision table packages that can generate procedural code based on the logic defined in a decision table
2. COBOL aids and COBOL precompilers such as ADR's METACOBOL

However, their program-generation capability is usually limited to only narrow portions of the complete application required.

At the R&D and futuristic end of the spectrum lie efforts on automatic programming conducted by the artificial intelligence community. However, such exciting possibilities continue in the research stages, are highly speculative and still long range, and remain to be proven in practical commercial situations.

PROBLEMS OF CURRENT APPLICATION GENERATION TECHNOLOGY

A number of problems are observed in much of the current application development software.

1. The large majority of turnkey applications software and self-customizing applications do not use database management systems. Neither the generators themselves nor the applications that they generate enjoy the benefits of GDBMS, such as data independence, relatability between files, access flexibility through query languages, and performance and efficiency, etc. This is rather surprising, since a number of the flexibilities provided by a GDBMS would tend to enhance the range of services and customizing provided by the turnkey or self-customizing applications software. A major reason for the current situation is that conversion of software to a database environment usually requires reprogramming of the software. In addition, if the vendor wishes to penetrate the full database market, it must either provide a version for all of the popular GDBMS's (e.g., IMS, TOTAL, IDMS, ADABAS) or utilize a generalized GDBMS interface with a translation bridge for each GDBMS supported, accepting some loss of function and performance as a result. This is a very expensive process that most vendors of such software have not been willing to undertake or able to justify. Nevertheless, it is expected that the bulk of such software will eventually evolve toward databases, just as the bulk of hand-coded applications software is evolving (actually, being largely reprogrammed) toward databases.

   User companies that have entered the world of databases will very likely find it undesirable to make use of application generators if the code generated is not database oriented. Only in the case that the applications generated have no connectivity to other database applications and do not use or interact with data from existing databases will non-database application generators be attractive. However, there may be various applications that can have such isolation, so non-database application generators can not be automatically ruled out.
2. The few turnkey applications software systems and self-customizing applications that do use or interface with GDBMS's usually select the one or two most widely used GDBMS's to work with, namely, IBM's IMS and Cincom's TOTAL. The CODASYL standard is followed by a significant number of GDBMS's but not by IMS or TOTAL. Applications software targeted at non-CODASYL systems is thus tightly cemented to a specific GDBMS and its vendor. Vendors of turnkey software and application generators face the same question that everybody else faces: which GDBMS should be used? For the vendor it makes sense to interface with the GDBMS in widest use.

   The same can be said of GFMS's regarding their use of database technology. They usually provide interfaces to work only with databases under IMS and TOTAL. There are only a few exceptions at this stage of evolution, significant ones being Cullinane's Culprit and IDMS (a CODASYL GDBMS).
3. Applications are evolving toward not only database but also data-communication environments. Users are increasingly demanding online terminal access and remote access through communication lines. This leads toward the need for generalized software for data communication, for example, IBM's CICS and Cincom's ENVIRON. Such generalized data-communication systems (GDCS's) are unfortunately incompatible with one another because no standard exists. Thus, any choice by a vendor of the turnkey software or of a customized application generator essentially cements the software to the particular data-communication system chosen. There is a trend to choose IBM's CICS because it is the most commonly used of the data-communication systems.
4. Turnkey software and application generators may claim to have built in their own database and data-communications (DB/DC) facilities. This means that part of the software generated for a client includes the DB/DC facilities. It is very doubtful that these generated DB/DC facilities match up to the power of the acknowledged generalized DB/DC systems. However, this approach may (a) enable the generated software to be independent of the variety of full GDBMS' and GDCS's in the market place, and (b) enjoy custom DB/DC horsepower. But what if, as most likely is the case with medium to large

organizations, the application software generated is to use a shared database under a GDBMS and not be isolated from other hand-coded applications using the database? Now we have an interface problem to solve, and it is not a minor one.

5. There is no standard for report writers and the more sophisticated GFMS's. Consequently, all GFMS's differ in syntax, semantics, and features (although functionally they pursue the same goals). The architect of an application generator faces the same problem that the architect of an application faces: which GFMS to use. Once one is chosen, the generator or application is cemented to it. To no longer use the specific GFMS would mean reprogramming for another GFMS, or reprogramming by hand-coding in a conventional programming language, which might cost much more than reprogramming for another GFMS.

   It is observed that many millions of dollars invested in hand-coded applications and also in turnkey software and application customizers could have been saved by the use of report writers and, particularly, sophisticated GFMS's. What applications do not involve report writing, sorting/merging, and so on? We seem to be reinventing the wheel all the time by hand-coding these tasks. The lack of a standard for generalized report writers and GFMS's has contributed much to the problem.
6. There is no standard for query languages. Even the CODASYL standard followed by a significant number of GDBMS's does not specify a query language. Practically all GDBMS vendors now market a query language with their GDBMS's. Unfortunately, all of the query languages are incompatible with one another, even among CODASYL GDBMS's. It would make much sense to be able to use the same high-level, nonprocedural query language, or maybe two or three at the very most, to access data from databases with little or no concern for the particular GDBMS managing the database. Unfortunately, in the 1970's we fell into the practice of developing essentially *n* query languages for *n* GDBMS's, rather than 1, 2, or 3 query languages for most of the GDBMS's.

   The trend toward relational query languages and GDBMS's may be a way out of the dilemma. Unfortunately, no standard for the relational architecture has been developed yet. Consequently, it is to be expected that incompatibility will predominate even among the new relational GDBMS's or relational query languages that may be developed on top of enhanced existing GDBMS's.

DESIRED CHARACTERISTICS OF AN APPLICATION GENERATOR

1. *It should be a system.* As indicated in previous sections, we have a significant and rather overwhelming variety of tools that go beyond the aging programming languages. Worse still, the standards for such tools are few. As a result, questions of overlap, incompatibility, what complements what, what interfaces are needed from whom, and so on make the task of the individual application architect most difficult.

   A satisfactory AG should synthesize all of the available services and productivity tools into a cohesive whole. It should be the driving force that ties the development process together. It is thus a system, not merely an application package.
2. *It should be databased.* A reasonable requirement for generated applications is that they be databased systems. If an organization's data and relationships are predefined and the data may be accessed by a GDBMS, then the problem of generating systems to manipulate the data is greatly simplified. If an enterprise intends to use an AG, it should also do some sort of information modeling, design major subject databases, install a GDBMS, and develop detailed "families" of operational databases in parallel with the implementation and use of the AG. A data-dictionary/directory is also indicated, either as an integral part of the GDBMS, as a part of the AG, or as a separate package.

   Subject databases are beginning to attract attention, and products are beginning to emerge. An example is Cullinane's Integrated Manufacturing System (CIMS), which embodies the structures of a manufacturing database.[21] Subject databases should be a good start and a more attractive alternative than starting from scratch for a significant portion of the market.
3. *It should interface with GDBMS.* The AG should use the GDBMS but should not be dependent on any specific GDBMS product. This means that the AG developer will have to build the system to be "GDBMS independent." An interface module will have to be supplied to bridge between the AG and the more popular GDBMS's such as IMS, IDMS, and TOTAL. Interface module specifications should be made available so that software vendors, user groups, and so on may develop interfaces with other GDBMS products. An interesting possibility in this direction is the Informatics' TAPS interface package,[22] offered to software vendors as a means of making their products DB/DC independent.

   Unfortunately, current application-generation software packages that embrace databases seem to be tied either to a specific GDBMS or to their own database facilities.
4. *It should emphasize terminal-based applications.* The AG should recognize that more and more applications will be on line and terminal based, and its architecture should emphasize this type of development. Fortunately, report-based batch type applications can be viewed as essentially a subset of the more complex terminal-based transaction type applications, and a system that has been designed to produce terminal-based applications can be extended to produce report-based applications with minor extra effort on the part of the system vendor. Specifically, the following facilities can be envisioned.
   a. Screen-format designer aid. Terminal-based application generation should begin with a user-oriented definition of the input and output terminal dialogs (i.e., screen formats) involved. The AG should include a screen-format design aid in its architecture.

This facility would permit the application designer to enter on line to the AG the exact formats desired, including data names, screen placement, and attribute characteristics. The AG would in turn obtain data characteristics from the data dictionary and/or database schema, perform editing, error checking, and standards verification, and return a "picture" of the desired formats to the designer, repeating the process interactively until the design is satisfactory. The AG would then create correct input data for whatever screen-format generation process is used by the DC system involved (e.g., Message Format Service in IMS/DC, Basic Mapping Support in CICS).

b. Batch application I/O descriptions using GFMS. The input/output specifications for batch applications should be developed by the AG using analogous techniques. A GFMS facility should be part of the AG, and it should use input transaction record descriptions and output report formats in the same manner as input and output screen descriptions.

5. *It should include data-flow fallout from user I/O specifications.* The information developed in the input/output specifications should be used directly by the AG as the foundation of the input-process-output data-flow specifications of the application. Data requirements that cannot be inferred from the I/O specifications should be added by the application designer in terms of what data are needed. Given the data-element names the AG should be able to determine the data format, where it resides, and how to access it. It should be noted that all information supplied by the application designer is machine readable and nonprocedural.
6. *It should include nonprocedural data manipulation.* Data-manipulation requirements should be defined to the AG through a menu or decision table. The associated data-manipulation code should be generated by the AG from a library of "canned" modules. Perhaps the AG should include a standard global set of such routines, with extra-cost options being available for various industry groups. The AG standards and interfaces should encourage easy library expansion of data manipulation capability by in-house development, user groups, software houses, and the product vendor. User exits should also be provided for custom-coded procedural language modules as required.

PROTOTYPE APPLICATION DEVELOPMENT SUPPORT

One of the greatest potential benefits of AG's is in their ability to reduce the impact of change on the application-development process.

Heuristic Development

It is unarguably true that users often do not know precisely what they need, that system developers often design a less than optimal solution, and that the application environment often changes during or soon after development. It is also true that great benefits would often accrue if an application could be implemented quickly, albeit inefficiently, and optimized later. It would thus be desirable to develop applications on an iterative heuristic basis if feasible. The AG offers the potential to do this.

"Prototype" Development

AG architecture should include heavy support for "prototype" application development for "quick and dirty" implementation, followed by migration later to more efficient implementation, where indicated. Features of such support might include:

1. DB space "for rent" in standard formats
2. Interpretive processing
3. Global screen formats
4. Application invocation of query facility
5. Effective monitoring and reporting of resource utilization, so as to highlight hogs and assist cleanup

Prototyping Scenario

In a typical scenario, the user and the application designer would discuss requirements and specifications. They would then use a terminal to define data elements, relationships, and space requirements in a temporary generalized database that was already in place. Next they would customize a few global screen formats to meet application needs. This would be followed by definition of data-manipulation instructions to an interpretive processor and the query facility. Preparation and input of data would follow. The application would now be ready to check out and use. Elapsed time would vary from a few hours to a few days, depending upon complexity. Probably response time would be slow for high-volume production, and resource consumption would be relatively high. But the user would be up and running and happy. He/she could easily make changes or do the whole system over if necessary. When the user was satisfied, the system could be regenerated on a permanent basis, perhaps as part of an integrated system and/or database plan.

IMPROVED APPLICATION MAINTENANCE

Another major benefit of the AG is in the area of application maintenance. Since there is little or no procedural code involved, application changes as a result of specification modifications should be greatly simplified. The AG vendor should design maintenance aids into the system, including automated documentation support, change-control and audit-trail facilities, and data-dictionary interface.

HUMAN BARRIERS AGAINST NEW SOFTWARE PRODUCTION TECHNOLOGIES

The introduction of new software-production technologies faces various challenges. One major challenge is the resistance of programmers, analysts, and other specialists whose

activities are directly affected by such technologies. Years ago one frequently heard the words "assembly programmers die hard" as we were leaving behind assembly programming and evolving toward higher-level languages such as COBOL and FORTRAN. Now we can perhaps observe that "COBOL programmers die even harder" as we are trying to leave behind programming in the conventional COBOL, FORTRAN, and such and evolve toward another level of software development.

One significant technology that is part of the wave toward replacing conventional programming languages is GFMS technology. GFMS's excel in report writing, sorting, and so on. As noted earlier, productivity gains of GFMS's over conventional programming languages have been publicized as averaging 8 to 1 in a significant percentage of the bread and butter applications of an organization. In spite of these productivity advantages, GFMS's are used in only a fraction of such applications. All too frequently one of the reasons why this is so, and sometimes the major reason, is the resistance of EDP staffs to abandoning conventional programming languages, or the inertia of continuing to do business as usual.

We should realize that many EDP staffs have been in existence for almost two decades by now. Bureaucracy has solidified in many EDP shops. Worse still, the "Peter Principle" is already present in a growing number of cases. EDP staffs used to be among the most dynamic and innovative groups in organizations, but are now at times exhibiting reactionary behavior like many established professional groups. There is the reality that in many shops EDP staff have invested 10, 15, or 20 years in becoming proficient and capable with conventional-language programming. To suddenly ask them to put aside the tools that they have learned so well over so many years is bound to raise problems in many cases. Resistance to change is a strong human tendency with which we must cope. It takes time to educate, change attitudes, and gain proficiency in significantly new ways of doing business. If it took several years for most assembly programmers to abandon their old tools in favor of conventional programming languages, it will take more years to abandon the latter in favor of a new generation of software production tools.

Such human resistance to change will have to be added to the technological difficulties of developing new application-generation technologies to replace the aging conventional programming languages.

Recent history is filled with examples of rapid public acceptance of profound changes in lifestyle, religious, social, and political attitudes, foods and consumer products, the arts, medical care, and other fundamental components of our culture. We should therefore be able to effect technological change in software-production techniques with equal success.

CONCLUSIONS

A number of important technologies participate in the wave toward application generation, away from the aging programming languages. The technologies range from

1. the inflexible turnkey packages, to
2. intelligent self-customizing packages, to
3. generalized file-management systems that can replace programming languages for batch I/O, report writing, and sorting/merging tasks, to
4. query processors and associated database-management systems (GDBMS's) that can replace programming languages for all I/O from databases, to
5. the newer application-development systems that integrate a number of tools to develop more quickly online, terminal-oriented applications in a database/data-communications environment.

Unfortunately, none of them individually fulfills the role of a complete AG. Besides constituting an overwhelming variety, these technologies exhibit various limitations and suffer from the lack of standards. Complementing one technology with another becomes a difficult interfacing problem.

There is the need for AG's that synthesize the variety of services and individual technologies into a cohesive whole. The AG should exhibit the following characteristics:

1. It should support a database environment.
2. It should interface with popular GDBMS's.
3. It should emphasize terminal-based applications.
4. It should include data-flow fallout from user I/O specifications.
5. It should include nonprocedural data manipulation.

The AG should include heavy support for "prototype" application development for "quick and dirty" implementation followed by later migration to more efficient implementation where indicated.

A major benefit of the AG should be in enhancing application maintenance by greatly simplifying the introduction of inevitable application changes.

Not all the problems and challenges of evolving toward future generators are of a technical nature. A major problem is the resistance to change of programmers, analysts, and other specialists toward embracing a new way of doing business, away from hand-coding in the traditional programming languages.

REFERENCES

1. Fisher, D. A., "DOD's Common Programming Language Effort," *Computer*, March 1978, pp. 25–33.
2. Brooks, F. P., *The Mythical Man-Month*. Reading: Addison-Wesley, 1975.
3. "Introduction to SDM/70 Systems Development Methodology." Atlantic Software Inc.
4. Tiechroew, D., and H. Sayani, "Automation of System Building," *DATAMATION*, August 1971, pp. 25–30.
5. "Problem Statement Language, User's Manual." IS-DOS Project, University of Michigan, Ann Arbor, Michigan.
6. Low, D. W., "Programming by Questionnaire: An Effective Way to Use Decision Tables," *Communications of the ACM* 10 (1973), 5, pp. 282–286.
7. Hammer, M. M., W. G. Howe, and I. Wladawski, "An Overview of a Business Definition System," in *Proceedings, ACM SIGPLAN, Symposium on Higher Level Languages*, Santa Monica, Calif., March 28–29, 1974.
8. Cardenas, A. F., "Technology for Automatic Generation of Application Programs—A Pragmatic View," *Management Information Systems Quarterly* 1 (1977), September, pp. 49–72.
9. Winograd, T., "Beyond Programming Languages." *Communications of the ACM* 22 (1979), 7, pp. 391–401.
10. Zollicker, M. L. (ed.), "Proceedings of a Conference on Application Development Systems," *Data Base* 11 (1980), pp. 1–20.

11. "Application Customizer Service, System /3 Application Description Manual." IBM Reference Manual GH 20–0628.
12. "Distribution System Simulator (DSS) for System 360/370." IBM Systems Guide LB-21-0980.
13. "QBE, Query-by-Example." IBM Reference Manual G320-6062.
14. "SQL Data System." IBM Reference Manual GH24-5013.
15. "Generation Five." American Management Systems Inc., Arlington, Va.
16. "IMS Application Development Facility, General Information Manual." IBM Manual GB 21–9869.
17. "Application Development System/OnLine." Cullinane Data Base Systems, Westwood, Mass.
18. "Mark V Concepts and Facilities Manual." Informatics, Inc., Canoga Park, California.
19. Holtz, D. H., "A Non-Procedural Language for On-Line Applications." *DATAMATION,* April 1979, pp. 167-176.
20. "Programming Aid Reduces Cost 1.5 Million Dollars." Information Processing, 1 (1982). Information Systems Group, National Accounts Division, IBM Corporation.
21. "Cullinane Integrated Manufacturing System, Summary Description." Cullinane Data Base Systems Manual TXCM-110-10,Westwood, Mass.
22. "Terminal Application Processing System, Concepts and Facilities." Informatics Inc., New York, New York.

Program generators and their effect on programmer productivity

by RICHARD L. ROTH
Information and Systems Research, Inc.
Coraopolis, Pennsylvania

ABSTRACT

This paper investigates the concepts and utilization of source-code program-generating systems within the business programming sector; and, using a "standard" system development framework as a guide, discusses the advantages and disadvantages of program generators to software organization and their personnel.

INTRODUCTION

The topic of software tools is becoming an increasingly popular one within the software community. Programmers' salaries are constantly increasing, in line with increasing demand for these technicians; and with the steady decline of hardware prices, the software industry is constantly seeking new ways to cut costs, increase productivity, and maintain the quality of the software produced.

Many different kinds of software tools have come into existence since the introduction of the digital computer. Of course, depending on the definition of the term *software tool,* the nature of these various tools may be less than obvious, or may even be taken for granted, in today's world of high-technology software.

For example, computer hardware provides access to magnetic storage such as memory, tapes, and disks. The operating systems software provides a simplified method of storing and retrieving data from these devices.

Computer hardware provides a set of machine-level instructions to guide its internal actions; higher-level languages and language compilers exist to provide a method of communicating algorithms with a vernacular consistent with the problem to be solved.

The use of single-user batched systems has been to a great degree superseded by interactive, multiple-user systems that provide the programmer with tools to ease the burden of creating and debugging programs. For example, these systems allow the use of interpretive languages, which bypass lengthy compilation procedures and allow immediate communication of problems and errors to the user during the entry of the program; on-line diagnostic aids have replaced unreadable memory dumps and allow the user to examine, step by step, the actions of the program during execution; on-line text editors have replaced the key-punch, allowing easy access to the source program for immediate correction and retesting of incorrect source code; and other utilities have been designed to simplify the task of communicating a problem-solving procedure to the machine that will eventually perform that procedure.

Another tool in use today is the concept of relational databases and database management systems.

Aside from the technical considerations, these types of tools provide an almost unrestricted access to data stored on an information processing system. Coupled with sophisticated query languages and report generators, a database management system can provide an unsophisticated user with easy access to information while simultaneously providing a powerful tool to the software designer.

However, the design of most query languages and report generators requires the inclusion of routines for all possible requests that can be made. Obviously, the more routines that are included in the query language, the more powerful it becomes.

Unfortunately, users who would most benefit from the *power* of this type of system are, for the most part, excluded from using this tool. The user of a small- to medium-sized system is unable to support, by virtue of machine size and software cost, the overhead of a database management system. It is a fact that the *needs* of the users of the small systems are no different from those of the users of the larger computers; regrettably, the "scaled-down" versions of these tools cannot provide the type of access that most users need.

However, none of the tools mentioned really accommodate what is generally the largest, most time-consuming task. Problem-solving procedures must still be designed and hand-coded prior to the use of any of these other programming tools. As a result the programmer invariably spends less time thinking about the problem and more time thinking about communicating the problem to the computer by a method the computer will ultimately understand.

One of the most recent additions to the class of programmer tools has been the program generation system.

PROGRAM GENERATORS AND THE SYSTEM PRODUCTION PROCESS

A program generator is, very simply, a *program that writes programs.* Directed by series of parameters entered into the system by means of a user-oriented front end, the program generator actually creates a source code program, which can then be compiled and executed in the same fashion as an equivalent hand-written program. This is very different from *table-driven* application generators, or even program generators that produce machine level code directly, in that neither of these methods allows for the manual modification of generated code.

The introduction of a program generation system into the system development environment has a number of effects on the individual aspects of the process of system design and programming.

Using the typical system implementation process as a benchmark, one can examine how the introduction of a program-generating tool would affect each of the six segments of the procedure and simultaneously determine the effect of such a tool on programmer productivity.

The first step in the creation of a software system is the system design stage. Normally, during this stage, the analyst would discuss the needs of the systems end user, and attempt to translate these needs into terms to which the programmer can relate. During this stage the programmer or systems ana-

lyst must organize a vast amount of information concerning the structure of files, the interactions of fields within these files, and the operation of the actual programs.

The introduction of a program-generating system has a profound effect on this task of designing a software system. In order to use the program-generating tool fully, the analyst must be fully aware of the philosophy of the program-generating system.

The typical program-generating system creates programs within a very small class of all programs. It is necessary for the system designer to keep this in mind while performing this analysis: that is, if this tool is to be used efficiently, the programs (or at least the functional procedures) designed during the specification stage must be as close as possible in internal operation and in form to the class of programs that can be generated. If the analyst strays too far from these prototype programs, the increase of productivity with the use of the program generator would be negated in proportion to the routines that would be "nongeneratable" and would therefore have to be created by conventional means.

It is the analyst's responsibility to use the features of the program-generating system within the application system design; in fact, the degree to which the program-generating system aids in the production of the programs is totally dependent upon the initial design of the system.

The existence of a program generator may also have ramifications in terms of the types of individuals who would actually perform the analysis. Because of the fact that the program generator would be addressing most of the computer programming issues that normally arise during an analysis, the individual actually performing this analysis should not necessarily have to be as sophisticated in terms of computer programming. This indicates that the individual performing the analysis could be more oriented to the *application;* in fact it may be that the end user could perform this analysis without the aid of a technician. As such, many of the communications problems that arise between nontechnical users and technical analysts would disappear, providing an analysis that would be more oriented toward the application area rather than the technical side of systems development. Finally, many of the difficulties that arise between the user and the designer would necessarily disappear, since they could very possibly be the same person.

The impact of a program-generating system can also affect the actual end result of the software systems design process. Most analyses yield a systems design communicated in the form of file layouts, including descriptions of fields, data types, field lengths, and allowed value ranges; data entry screen layouts, handwritten to show the position of fields and computer responses; report layouts showing the positions of fields on the printed page and totals and subtotals given on the report; flow charts and narrative flow descriptions, indicating the actions of individual programs and routines; and other information communicated by the printed page.

However, when considering the existence of a program-generating system, many of these written documents could be eliminated, thereby decreasing the amount of time the analyst would have to spend in preparing them, not to mention the clerical time and cost necessary to produce them. In fact, since the program-generating system would require the input of its parameters in some predetermined manner, the actual form of the end product of the system design could very well be the data input sheets for the program-generating system.

The second step in the system development process would be the actual programming, according to the specifications developed and accepted during the previous step. The role of the programmer during this phase of the project would also be dramatically altered by the introduction of a program-generating system. Assuming that during the previous step the systems analyst took pains to adhere to the philosophy of the program-generating system, and further assuming that the end result of the analysis was recorded on the data input sheets for the program-generating system, for the most part the programmer's task would become clerical. In fact, depending upon the user-friendly nature of the front end to the program-generating system, entering the results of the systems analysis into the program-generating system might not require the services of a programmer but rather those of a clerical worker. The effects on productivity here are obvious. Routines that are inherently simple and are repeated many times within the series of programs making up the application system would most probably be those created by the program-generating system. Programs such as formatted screen data entry programs, sorting programs, formatted report programs, and less complicated file update routines would most probably be targets for the program-generating system. In fact, depending on the application system, it can be shown that programs that fall within these categories represent anywhere from 50% to 90% of most business-oriented systems in existence today. In this regard one can further extrapolate to say that programmer productivity would then increase by similar percentages on the typical project.

In any system there are programs that because of their application area or complexity of design fall outside the class of programs that could be generated. The programmer would necessarily have to produce these programs in a conventional manner. However, because the program-generating system has been used to write many of the more mundane programs, these generated programs actually set up guidelines for the internal structure of any other program that would have to be handwritten. For example, portions of source code that defines file structures, performs initialization functions, opens files, performs input/output functions, and performs other system-wide utility routines could be extracted from the generated programs. Even in the cases where programs would have to be written by hand, a major part of these programs could be borrowed from the source code generated in other, automatically created programs.

In terms of program text editing time alone, the savings would be quite significant. In terms of more important factors such as program correctness, debugging, and organizational source code programming conventions, the savings would be substantial.

The third of these six steps in the system development process would include the actual testing and debugging of the software system. Assuming again that the 50% to 90% figures are accurate for the average software system, it follows that the same percentages would hold for the number of programs which, by virtue of their being generated rather than hand-coded, could be assumed to be error-free (at least at the

programming level). One could then assume that the amount of time necessary for a programmer to test individual routines for desired functionality would be reduced by a comparable amount. The major portion of the testing and debugging process would only involve checking the interaction between program modules, checking any routines that had been hand-coded, and verifying the mechanically generated routines that for one reason or another might have been modified by the programmer after their generation.

The testing and debugging process is all too often skipped over in the interest of getting the system up and running. With the use of the program-generating system this technique may in fact become more acceptable as more and more of the programs can be considered bug-free at the point at which they are generated.

The fourth step in the system implementation process involves the creation of the software documentation, both from a technical and from an end user standpoint. The program-generating system can increase productivity in this critical area as well. From the fact that the programs that would be generated exist in a class of similar programs, it follows that the documentation describing the actions of these routines would also be fairly regular in their form. This fact holds true for technical-level as well as end-user-level documentation. The only variables within this documentation would be based on the actual parameters input during the initial program generation process, which could be preserved during their entry. The program-generating system would generate technical documentation in the form of file layouts, screen formats, report formats, and flow charts of the program's activities, complete with machine-generated English-language narratives describing this flow. The clerical time required to produce this documentation would be reduced to the level needed to generate technical documentation for the hand-coded portions of the system, overall system philosophy, and other nongeneratable documentation.

In addition, because the activities of these routines are also defined within a narrow class, the user level documentation would be similarly generated. For example, English-language narratives describing the activity of a screen data entry program, complete with field-by-field descriptions of data to be entered, range checks and edit checks performed, and legal versus illegal values, could all be produced during the process of program generation. In this fashion pages for the users' guide could be generated, and the clerical time needed to develop the complete users' guide would be reduced to the task of developing users' instructions for the hand-coded portions of the system.

The use of program generators also adds a new dimension to the fifth step, which is training the users of the software system. This area is vital to the success of any software installation. Many excellent techniques using the computer's power for CAI (computer-assisted instruction) have been developed, but for the most part they have been ignored because of the time and expense necessary for incorporating these techniques into the design and programming of an application system. In addition, the added size and disk capacity needed to store this added information could become prohibitive on a small computer system. However, program-generating systems could be used to solve these problems. During program creation, dual sets of programs could be generated, of which one would be a normal application program for production use and the second would be a temporary program containing the additional code and textual information required for computer-assisted instruction. This secondary set of programs would be maintained in lieu of the first set during the initial training and installation and then removed from the system and replaced with the production programs once the operators were trained sufficiently. Again, the amount of personnel time required to do training at the user's site would be significantly reduced by the existence of computer-assisted instructional programs, without the added burden of writing these programs and the permanent burden of maintaining these routines in a production environment. The programs that interface directly to the user (for example, screen data entry programs and query-level report programs) would be excellent candidates for computer-assisted instruction code created by a program-generating system.

The last of these six steps generally occurs sometime after the final installation of the software system. This step, of course, is the modification and enhancement of the software system once it has been placed in production use. In many instances, and especially where a database management system has not been used, the addition of fields to data files and the addition of new features to application programs represents a major problem for the software systems organization. Individual programs must be individually modified to acknowledge the existence of new data and functions, documentation must be updated, and in some cases users must be retrained. Even in instances where COBOL-like libraries have been used throughout a system of programs, the procedural sections of each individual program must still be modified in order to use this new data. In this situation the program generator becomes an invaluable tool. Unlike the COBOL-like libraries and other source code management techniques, the parameters entered into a program generator carry with them information concerning the *actions* of the programs, relationships between fields, and specific descriptions of individual fields within data files. In the event of post-installation modifications these parameters can be updated to include new fields and concepts and the program generator can be used to create updated programs, training materials, technical documentation, and user level documentation for the newly modified system. It is not difficult to foresee instances where it will be more cost effective to automatically recreate entire subsystems that have been previously handwritten, even when the updates to that system are not dramatic. In cases where the original software was automatically generated this type of update would be only an operational (rather than a technical) matter.

In summary, then, the existence of a program-generating system dramatically increases productivity in all the major areas of the system design and development process. Assuming that the system design is consistent with the philosophy of the program-generating system, the systems installation process—including the system design; the actual specifications; and the programming, testing, debugging, installation, and training—would all enjoy significant increases in productivity.

This new mode of systems development is based to a great

degree on the existence of a program-generating system containing features in all the areas of system development, including systems analysis, actual programming, technical documentation, user level documentation, computer-assisted instruction, and end user training. In addition, it assumes that the capability of this program-generating system extends into many subapplication areas, including screen data entry program generation, sort program generation, file update program generation, reports program generation, and the existence of a user-friendly front end to access the program-generating system. But even with a program generation system that does not contain all these features, it is evident that the role of computer professionals in light of these program-generating systems, and the activities that will be performed by these professionals, have been dramatically altered. In essence the program-generating system would cause a type of professional migration. Systems analysts would strive to become experts in one or more application areas, and programmers and program analysts would strive to become algorithmic engineers, spending less time thinking about explaining problems to the computer system and more time thinking about sophisticated solutions to the actual problems at hand.

ACCEPTANCE WITHIN THE PROFESSIONAL COMMUNITY

It may be, however, that some programmers are fearful of this attempt at automating their jobs; in fact, many become resentful. This is unfortunate for two reasons: First, the concept of program generation is truly in its infancy and requires the input of a majority of the data processing community to become a truly effective tool. Program generators must be created for a much wider range of application areas than they exist for today. And, of course, these program-generating systems are not today themselves generated; in that they are also programs, they are today handwritten. Second, it is usually these types of people who resist the professional advancement that program generators offer—who are either fearful of or unable to make these changes within their professional community.

It is conceivable that the introduction of a program-generating system into a software organization where these types of professionals exist would not cause a dramatic increase in productivity. The impact of the system would be lessened by individuals who resist the use of the product.

ADVANTAGES AND DISADVANTAGES

A program-generating system is not a panacea; and, depending on the method of its use within a particular environment, the program-generating system has several advantages and disadvantages to the organization using it. In terms of the organization, the existence of a program generator, properly used, would necessarily decrease the need for programmers and programming support individuals involved in the creation of systems software. For the most part, the low-level programming activities would be handled cost effectively by the program-generating system. In addition to the actual programming, the organizational overhead would also be reduced in terms of documentation, training, and technical support personnel. The need for systems design and analysis staff within the organization would probably not be reduced, but the skills that these personnel would have or acquire would be dramatically different from those of their counterparts in organizations not incorporating a program-generating system. Systems personnel would necessarily begin to acquire skills in application areas rather than systems programming areas to supplement the effect that a program-generating system would have on the organization. Although this is not a direct productivity gain, it does have positive ramifications for the software organization. It is apparent that the most successful software houses concentrate on a small number of application areas, and the quality of their products is in direct proportion to the amount of knowledge concerning the *application* (not systems) that the software house and its personnel command.

In examining the disadvantages of a program-generating system it is possible to draw several analogies between program-generating systems and the concept of structured programming. For either of these techniques to be used effectively within an organization, the organization must be organized to use the tool properly and effectively in the day-to-day business of creating programs. One of the most obvious disadvantages of a program-generating system is its inherent inability to generate all the programs required in a particular software system. Because no program-generating system can accomplish 100% of the required tasks, there are gaps in the program-generating process that must be filled by conventional programming techniques. If the organization is not structured in such a way as to monitor this activity, a long-term trend of increasing numbers of conventional programs and decreasing numbers of generated programs may occur. Obviously, then, this one disadvantage of program-generating systems would effectively negate all the aforementioned advantages.

Another disadvantage of program generation systems is that the source code is generated with a single, immutable style. Obviously a given program-generating system will generate programs with a single, consistent internal structure; and as such, this internal design will probably not be consistent with that which the organization has been producing previously. Again, the degree of acceptance of a program-generating system would be based on how well the organization could change its techniques and internal standards to match those of the program-generating system. Unlike a human programmer whose techniques may be modified, the program-generating system cannot be so easily retrained. For the same reasons as mentioned above, any roadblock in the path of acceptance of a program-generating system decreases its effectiveness within the organization. However, this disadvantage might also be considered an advantage in organizations having no internal standards for the generation of conventionally written programs.

A third disadvantage, along these same lines, has to do with the actual operation of the programs that the program-generating system would produce. In terms of highly visible, user-interactive programs such as screen data entry programs, and in terms of printed output such as that produced by report programs, the output of the automatically generated pro-

grams might not conform to the standards already accepted within an organization or might not match those formats created by canned packages used by the organization. This could lead to extensive modification of generated programs, abandoning the program-generating system as an in-house tool or massive updates of these canned routines. Obviously, any of these actions would have a severe effect on productivity gains.

It is interesting to note that all the disadvantages mentioned could be (for the most part) eliminated by either modifying standards to include the program-generating system or designing the program-generating system to match widely accepted standards. Similar to the structured programming techniques mentioned above, the organization's ability to accept these techniques is directly proportional to the benefits gained by using them. Many programmers within an organization may resist the installation of new techniques or programming tools, stating reasons such as "We've always done it this way." As Edward Yourdon states in his book, *How To Manage Structured Programming,* "It's literally all they can do to write programs in the disorganized fashion to which they've been accustomed; to suggest they should introduce some organization, some structure into their work is literally beyond their ken." (P. 172)[1]

Unlike structured programming techniques, however, the program-generating system can, in effect, replace this type of individual within the organization instead of attempting to modify the individual's behavior to fit some predetermined pattern.

OTHER USES OF PROGRAM GENERATORS

As previously discussed, program-generating systems must continue to grow, both in their capabilities and in the scope of generated programs, in order to be an effective tool for the future.

For example, program-generating systems should be expanded to cover more of the tasks involved before and after the actual generation of a program. Tools designed to simplify the task of systems analysis, including natural-language input of system parameters, would be a very desirable addition to the front end of a program-generating system. In fact, within a certain restricted set of application programs, it may be possible to design a table-driven decision tree system that would actually guide the unsophisticated user through a systems analysis of the desired application. This system would simultaneously gather the parameters to be used by the program-generating system. Coupled with computer-assisted instruction and even a natural-language interface, this technique would provide for direct interface with the end user and eliminate the need for computer expertise during the design stage.

The program-generating system could be used as an instructional tool for teaching proper programming techniques within a classroom situation and as an instructional tool for new members of the programming staff of a software organization. Obviously, if the internal standards of the organization were closely related to the structure of the generated programs, an instructional program could be based on the program generation system, teaching new programmers the desired structure of conventionally written programs. In a classroom situation dealing with first-time programmers, the program-generating tool could be invaluable in instructing students in the use of proper programming techniques. The student would simply describe a desired program to the program generator and would be able to examine the final results. By imitating these programming techniques, the student programmer would learn these techniques by example.

Another natural migration path for a program-generating system would be the generation of programs in other languages, as well as the generation of programs for other manufacturers' computers and operating systems, as an option during the generation process. For the sake of ease of migration from one manufacturer's computer to another, and especially in the case of languages such as COBOL and BASIC, this migration would be a fairly trivial task when compared to the task of creating a new program generator for the new computer system. However, the task of translating from one language to another is not as simple.

For example, consider the case where a COBOL program is written to generate COBOL programs. Simply translating this COBOL program into another language, say BASIC, would result in a BASIC program, which would still generate COBOL code. It is therefore necessary to translate not only the program itself, but the class of programs that program would generate as well.

There are many ramifications to consider regarding the possibility of an easily transportable program-generating system. Unlike any other transportable feature of an operating system or language, the idea of a transportable program-generating system implies the existence of transportable programming techniques across manufacturer lines and the software organizations using the manufacturer's computers. It is reasonable to assume that if a single program-generating system were to become popular in the industry, and further, that the program-generating system created source code in a transportable language, for a variety of popular computers, these conditions would define a standard of programming *technique* unparallelled in the industry today.

The existence of such a portable program-generating system also has some implications for the microcomputer side of the industry. By simply reading today's microcomputer journals it is easy to see that the average cost of a software package, be it operating-system or application-level, is generally a small percentage of the hardware price. As hardware prices continue to drop, the prices asked for the software may drop to almost ridiculous levels. If one adheres to the concept of getting what one pays for, it becomes obvious that the quality of conventionally written customized programs existing on microcomputer systems would be less than desirable. However, the existence of program-generating systems on these microcomputer systems would be a natural way to increase the quality of the custom software products without increasing their price (beyond that which the typical microcomputer user would be willing to pay).

SUMMARY

It is obvious that, unlike any other tool, the program-generating system is today and will continue to be a major

factor in changing the face of software organizations. As these tools become more sophisticated in terms of their internal structure for the generation of programs, expand their scope in terms of the types of programs they will generate, and improve their interface to the end user via user-friendly natural-language front ends, the program generator will literally reorganize the whole structure of the design, installation, and implementation of computerized software systems. Many of the problems besieging our industry today, including lack of good-quality personnel, lack of widely accepted standards, problems with reputation and acceptance by the general public, and the rapidly decreasing price of hardware with respect to the labor-intensive costs of software production, are problems that can be solved today and in the future by the program-generating tool.

REFERENCES

Yourdon, Edward. *How To Manage Structured Programming.* New York: Yourdon, Inc., 1976.

Application generators at IBM

by AARON M. GOODMAN
IBM Corporation
San Francisco, California

ABSTRACT

This paper discusses the reasons for the great interest in application generation. Because of the growing backlog of user demands on data processing, a new technology with an order-of-magnitude increase in productivity is suggested. IBM has two families of application generation products. One, DMS, operates in a CICS/VS environment; the other, IMSADF, operates in an IMS/VS environment. An overview of the techniques involved and the benefits of using these systems is discussed.

The most important problem facing data processing management today is the backlog of applications. There are a number of studies done by IBM and other groups which show the backlog to be many years long. Figure 1 shows the results of some studies on application backlog. The trend for the future indicates an even greater need for data processing solutions. The labor force is growing more dependent on information which must be provided to do their jobs. We, as individuals, demand more and more information in a timely fashion; it would be inconceivable, for instance, to go back to the days where airline reservations and ticketing were not immediate. As consumers, we ask for more and better service.

Application Backlog Growth

	1977	1978	1979
Have Application Backlog	94%	96%	97%
# of Applications in Backlog	5	15	20
# to be Operational by End of Year	3	3-4	5
# of Online Applications in Backlog	—	—	10

Figure 1—Statistics on application backlog

One reason for the large and growing backlog is, as mentioned, the increase in service that we demand from data processing groups. There is another important reason, i.e., the availability of skilled data processing professionals. Figure 2 shows the expenditures for programming, as reported by Diebold, across industry. We see that 24% of the data processing budget is for programming, but 69% of that 24% is used to maintain existing systems. Therefore, only 8% is available to develop new applications. The backlog of applications can be addressed in only a small way, since the bulk of the programming resource must be used to maintain current systems.

The difficulty of finding and attracting qualified programmers is well known. Figure 3 shows some results of studies and indicates that the future is bright with prospects in the programming field. Put another way, it's going to be more difficult and much more costly to satisfy our data processing needs by simply hiring more programmers. This is already evident when we recognize the decline in the percentage of DP budget due to hardware costs and the increase due to software costs. Due to the educational philosophies of various countries, this problem may be more or less important, depending on geography.

So, we find a situation where more data processing solutions are required, but with severe constraints of people and budget to get the necessary systems.

The DP Dollar

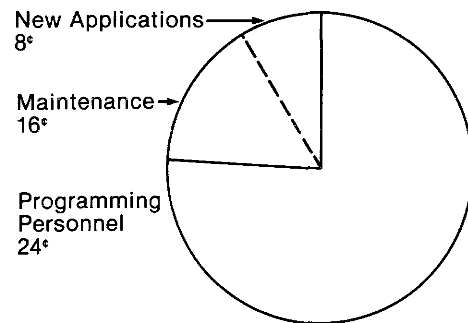
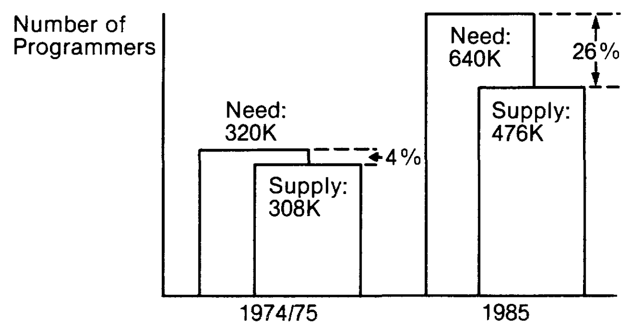


Figure 2—Programming expenditures

In addition, the advent of major DB/DC systems such as IMS and CICS brought two long-existing problems into sharp focus. One is data security, a topic of much discussion in data processing. Now it is a more significant topic since more data are located in a single or a few repositories. The data, being more centralized and more complete, are more valuable. The results of destruction or unauthorized usage are more profound. In addition, sharing of the data processing resources, database, and programs has caused more attention to the topic of change isolation. How to share resources, while still retaining independence for individual systems and being able to change systems without widespread effects on other sys-

Programmer Availability



Source: U.S. Department of Labor, SHARE SILT Report.

Figure 3—Statistics on programmer supply and demand

tems, is a significant problem which is accentuated by DB/DC systems.

Many of these problems can begin to be addressed today by using a new tool—Application Generators. This "new" tool is, in fact, a continuation of a trend we've had in data processing since the 1950's. Remember machine language? We've come a long way since then, and Figure 4 will allow us to reminisce a bit.

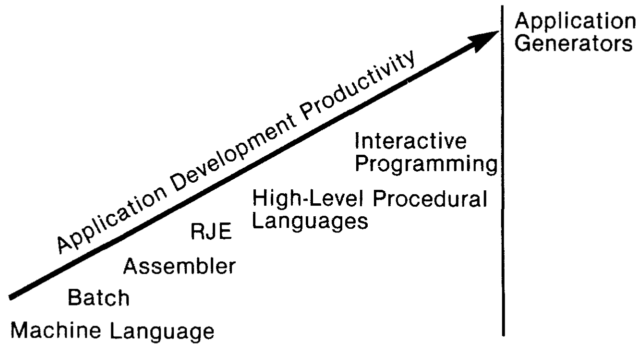


Figure 4—Trends in programming technology

IBM has developed two families of Application Generator products in the large system environment. One is based on usage of IMS/VS; the other is based on CICS/VS. In the early 1970's, people in the Rochester, Minnesota, manufacturing plant of IBM developed a system to assist in solving a number of problems with application delivery at that plant. That was the granddaddy of IMSADF. At about the same time, people on the East Coast were developing a system to assist the development of online systems (particularly screen design and development), and this was the start of DMS/CICS/VS.

We now have a number of years of experience with these systems and have worked with customers to support more and more requirements. From an architectural point of view, a generator is feasible because it makes it possible to identify and separate out functions which are common to many applications, code them separately, and put them together in a way that supports a particular application need. The programmer, then, needs to work with specifications which relate to the business needs to be addressed, not primarily to the programming required.

Let us look at the functions in an online program. Architecturally, most business data processing applications will fit into the framework of Figure 5. We have, in addition, coded the logic of each of these functions in a generalized manner and have developed a technique to supply specifications through a source external to the common program modules. Therefore, we have decoupled the FUNCTIONS which satisfy the application requirement from the PARTICULARS of that exact transaction.

The usefulness of a particular generator depends, in large measure, on its richness of function. It is clear, I think, that we'll not get to the point where all application requirements are satisfied with common code alone. Both ADF and DMS allow for EXITS and Special Processing, where the programmer gains control and programs in a traditional manner. Where this is required, the increased productivity promised by generators is reduced. Therefore, the intent is to, over time, put more and more of these functions into common code.

The benefits of this technique accrue throughout the development cycle—during design, during programming, in test, and during maintenance. During design, the ability to work closely with a user and prototype the solution quickly, enables the user and the programmer to understand the needs and agree on the system. The wide gulf between user and the DP function is often crossed by simply showing a proposed solution, or at least an approach.

During programming, much of the work is reduced because much of the code is supplied. The programmer often works at the external level of the system, specifying the particular requirements for this particular application. If security is to be enforced, for example, the programmer now supplies DATA about the authorization, rather than reinventing another security module. Even if some code must be written because all the needs are not addressed, it is much less than if the whole application were written conventionally.

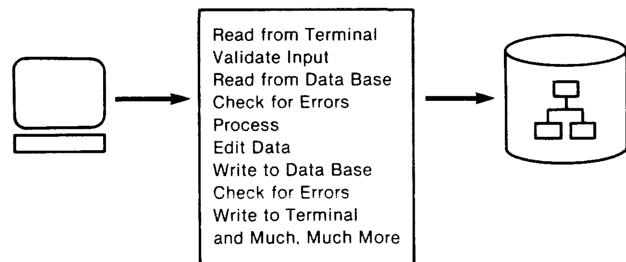


Figure 5—A traditional application development framework

The approach to testing can be quite different if a generator is used. The common modules of code are pre-tested, and if problems occur, they are to be corrected by the vendor. But, even more important, hundreds of customers are running these exact common modules; therefore, they tend to be thoroughly debugged. Testing of a particular application involves, by and large, verification of the parameters and business information that the programmer provides to the generator.

All of the above contribute to the improved maintainability of generator-produced solutions. But in addition, a person can easily maintain their application because changes are usually external, and the common code is left alone. It is also much easier to maintain someone else's application, since the need to understand the data processing techniques used is lessened. Parameters, constants and business rules change, and these in fact are held separate from the common modules.

Because of these benefits, we can now begin to approach the growing backlog with a tool which offers an order of magnitude of improvement on our productivity.

Application generators: a case study

by JAMES H. WALDROP
Hamilton Brothers Oil Company
Denver, Colorado

ABSTRACT

Hamilton Brothers Oil Company recently implemented a complex accounting and finance system. This system contains 157 reports, 53 online screens, and 1,320 data elements. The system was designed, developed, and implemented in 20 months. Most of the approximately 300 programs contained in this system were developed with an application generator. Using an application generator allowed the project team to complete the coding and unit testing in three months. Based upon the experience acquired in this effort, many benefits appear to be gained by using an application generator in the system development process. These include reduced coding, faster testing, and enforced structured development. However, these benefits must be weighed against the constraints associated with an application generator. These include potential file and processing inefficiencies, limited language syntax, and minimal debugging facilities.

INTRODUCTION

Hamilton Brothers Oil Company recently developed a new, comprehensive accounting and financial information system. The development of this complex system was accomplished under an extremely difficult schedule. A key ingredient of this project was the use of an application generator language to address the programming requirements. An application generator is a software tool designed to anticipate standard user requirements in a functional area such as accounting and finance. It provides data management facilities, automated common functions, and a specialized language. The following narrative presents a brief project background, the project's goals, the impact of an application generator language on the system development process, and several key areas to consider for future use of application generators.

PROJECT BACKGROUND

Hamilton Brothers' senior management decided in January 1980 to terminate its service bureau computer support and implement an inhouse Financial Reporting System (FRS) consisting of five integrated subsystems: accounts payable, accounts receivable, general ledger, joint interest billing, and budget. After a thorough four-month evaluation of 19 different vendors, it was determined that Hamilton Brothers' requirements could only be met either through a total custom development or through significant modifications to an off-the-shelf software package.

Amplifying the magnitude of the challenge, the company simultaneously began a major expansion effort. This expansion led to an increase in new key users who would determine the design of the system. In addition, the company had to establish and staff a complete medium-size computer installation. Furthermore, the data to be used in the new system would also require a complex conversion effort from the previous service bureau environment. The target date for implementation of the new system was January, 1982—20 months elapsed time.

The plan to accomplish this substantial effort encompassed several critical phases. First, a detailed evaluation of the request for proposal (RFP) responses from the qualifying vendors was accomplished. The selection criteria focused on several topics: optimum use of off-the-shelf software to address user requirements; utilization of the most current software architecture; maximum software flexibility; and a financially and technically sound software vendor. This phase was accomplished in two months, culminating in the selection of the Corporate Financial System (CFS) marketed by American Management Systems, Inc. This system was developed using their application generator (Generation Five) as the underlying software. It would become the core of FRS after considerable custom modifications. Second, a system concept definition phase was performed to refine the general requirements presented in the RFP. This phase also required two months. Next, a general system design stage was completed requiring six months. With the completion of the general design, there were only 10 months remaining to accomplish the detail design, programming, testing, and implementation phases. It was obvious that new techniques must be evaluated to maximize the efficiency within the programming, testing, and implementation stages. This requirement, coupled with the fact that CFS had been developed using an application generator, led to the consideration of using an application generator to perform the custom development work.

PROJECT GOALS

As with any project, the ultimate goal is to provide a sound system which meets the user's requirements. However, because of the special nature of the FRS project, there were several ancillary goals to be addressed. First, FRS must be user friendly. To accomplish this, we introduced the concepts of online processing, user controlled parameters, and user report development with minimal support from the data processing staff. We also required a table-driven architecture to provide flexible data modification by the users. Second, FRS must be capable of easy technical modifications and provide a solid base for future expansion. This consideration introduced the need for a structured development methodology and self-documenting code. Third, the system development process must take advantage of key development productivity tools. This aspect included online program development, simple report writers, online screen generators, a data dictionary, and a project task plan.

Since this was the department's first project, its success was crucial to establishing the department's credibility. The following synopsis of the characteristics of the system provides an insight into the magnitude of the project:

- 1,300 pages of general design narrative
- 6,000 mandays for concept definition, general design, detail design, and implementation
- 25 people assigned fulltime at the peak point in the project
- 3,000 pages of automated technical documentation on a data dictionary
- 1,320 data items
- 285 programs (most of these were developed in Generation Five or its associated report writer)
- 157 reports

- 53 online screens
- 102 master tables
- 132 data files
- 48 input forms

The selection and use of an application generator contributed to the achievement of the project's goals and its timely implementation. The FRS project was completed within one month of the plan which had been developed nearly two years earlier.

IMPACT OF THE APPLICATION GENERATOR ON THE PROJECT'S DEVELOPMENT

The programming effort was started six months prior to implementation. The project plan allocated approximately 30% of the elapsed time (50% of the total mandays) for all technical facets of the project. At this point, the project leaders reviewed the application generator (Generation Five) marketed by AMS. It appeared to offer several programming advantages over a standard programming language such as COBOL or PL/1. The syntax appeared easy to code and did not seem to require significant training. General housekeeping functions such as opening and closing files were eliminated. Several application unique keywords were included in the syntax. For example, REJECT-BATCH and REJECT-DOCUMENT offered easy techniques to reject an entire batch of data or only an individual document within the batch. The online screen generation facility of Generation Five also appeared to provide a relatively simple, high-level technique for developing input/output screens without acquiring the knowledge necessary to develop native teleprocessing code. The structured methodology inherent in Generation Five coupled with its editing facilities offered the potential of eliminating the occurrence of a program dump. The "forced" structured methodology should also require fewer compiles to complete a program developed in Generation Five. The obvious benefit to be gained was increased programmer productivity and reduced machine utilization.

The above expectations were developed by the project team prior to using Generation Five for the first time. As the project progressed through the remainder of the development effort, the actual experiences encountered using Generation Five were somewhat different from the expectations. The following narrative is a synopsis of the benefits gained from using Generation Five as an application generator, a discussion of the key problems encountered in its use, and a review of the project team's expectations of Generation Five compared with its actual performance.

Problems encountered

The initial obstacle encountered by the project leaders was the project members' occasional resistance to using a new software language. The programmers preferred to use COBOL, a familiar tool, to do the development. This issue and the lack of previous experience with any similar product presented many unknown problems in the design process.

Without any previous Generation Five experience, the staff was not aware of the constraints inherent in the software. As each design issue arose, such as file merging/sorting or data element storage and retrieval, the options and capabilities supported by Generation Five had to be reviewed. The conclusions reached from the review of supporting documentation occasionally did not match the actual required implementation of the design issue. For example, a file that required both random and sequential processing could be accomplished with only one copy of the file in a COBOL designed system. Under Generation Five, the random file must be copied to a sequential file for sequential processing. For small files, this was not a critical issue; for large files, it introduced excess processing requirements.

The thought processes used by the programming team for such typical data processing functions as file access, edit error message generation, and online screen generation also had to be altered. The file access method used by Generation Five is a non-standard technique. This becomes critical when other program languages are used to access the data stored by the application generator. For example, when non-Generation Five software is used to update Generation Five data files, those updates are not integrated with the Generation Five recovery facility, thereby increasing the complexity of the backup and recovery function.

One of the unique aspects of Generation Five is its ability to automatically generate online and batch edit error messages based upon the section (or paragraph) name used for the portion of code performing the edit. However, this feature eliminates the dynamic characteristics of edit error message presentation. (For example, "COMPANY CODE xx IS INVALID" where xx is replaced by the invalid company code.) The online screen generation capabilities provided by Generation Five allow extremely easy and fast development of either input only screens or output only screens. It does not provide online interactive screen facilities.

Other limitations encountered using Generation Five included constraints on the number of data elements allowed in a random file (master table) record plus the inability to allow multiple record types in a file. These two constraints required an increase in the number of random files that had to be defined in the system leading to suboptimal use of storage space. Also, the architecture of Generation Five presented critical throughput problems during compiling and testing. Simultaneous compiles/tests occurring against the same database were prohibited. This forced the project team to create multiple copies of the database to avoid single thread compiling of Generation Five code. While the limitation of simultaneous executions against a single database is inherent in many non-Generation Five systems, the single thread compile requirement is unique to this software (because compilation accesses the database). This fact has had a negative effect on the simultaneous development of multiple programs using Generation Five.

Benefits gained

While several unexpected problems were encountered using Generation Five, it is also true that several benefits were

gained from its use. The structured architecture and selected keywords enforced programmer standardization. It also provided an excellent technique for developing common code included in many programs. These characteristics reduced the amount of code that otherwise would have been required for a typical program. The online screen generation facility eliminated the need to train the project team to program in native teleprocessing code. The code developed to process the online input data could also be used to process the same data in a batch mode. This eliminated the need to develop two sets of processing code, one for batch and one for online. This was a significant benefit given the 53 online screens that were required. Simple "read-a-record, process, write-a-record" applications were coded and tested rapidly. These last two facets of the system allowed very junior programmers to be immediately productive on the project.

A major benefit provided by Generation Five is an automatic transaction suspense processing facility. This is implemented through its document database feature. Input transactions captured through online or batch processing techniques reside in a holding file (the document database) awaiting subsequent processing. After processing, the rejected transactions are retained on the document database until corrected, while valid transactions proceed through the remainder of the processing cycle. The invalid transactions can then be corrected as necessary. This feature eliminates the sizable effort required to design, develop, and implement complex transaction-master file processing requirements.

A subtle feature of Generation Five is the ability to use identical names for multiple data fields. This feature offers several advantages. First, data fields with the same name are moved automatically from the input record to the output record (without regard to their relative position within the records). This eliminates encoding numerous MOVE statements to accomplish the same objective. Second, data fields can use standard naming conventions not only among multiple programs but also within a program without the use of qualifying syntax. This reduces the coding required and also greatly simplifies tracing the life cycle of a data element throughout all of the software acting upon it. Third, the use of a data dictionary to document Generation Five code is greatly simplified. With standardized data element names enforced through the Generation Five architecture, the number of unique data element names are minimized. This reduces the information to be captured within the data dictionary and enhances its consistency and accuracy.

Coding in Generation Five also eliminates the need to open and close data files. While this is advantageous, a greater benefit is achieved by not coding end-of-file logic. Depending upon the application, this may be a very complex function to accomplish in a traditional programming language.

Expectations versus actual performance

The actual experiences encountered in using Generation Five when compared to the project team's original expectations highlighted areas where Generation Five could be improved. The language syntax did turn out to be relatively easy to code and was no more difficult to learn than another high-level language. However, while the structured aspect of the language made it self-documenting, the syntax was occasionally confusing to follow. Portions of the software allowed a PERFORM capability (the EDITOR segment of Generation Five) but did not allow nested IF logic. The remainder of the software (the report writer) allowed nested IF logic but prohibited the use of a PERFORM.

Several general housekeeping functions were supported and directly contributed to improved programmer productivity both in the coding and testing stages. The use of selected application unique keywords such as REJECT-BATCH or REJECT-DOCUMENT also eliminated the need to code and test these types of routines. Other application unique keywords such as POST or GENERATE (another form of WRITE or PUT command) offered virtually no productivity gains but did relate the function (WRITE) to application oriented terminology (POST).

The online screen generation provided the greatest increase in programmer productivity. Fourteen of the sixteen programmers on the project team had never used a teleprocessing monitor and, therefore, had no experience in generating online screens. The Generation Five screen facility provided each programmer with an easy-to-learn tool to perform that function. However, it does lack the interactive characteristics desirable in some applications such as receiving input parameters, retrieving data from multiple files, performing a calculation, and displaying the results.

Perhaps the most criticial disappointment in the use of Generation Five was the frequency with which program dumps were encountered. Unlike using COBOL or PL/1 where the programmer knows how to interpret the dump, this generally was not the case under Generation Five since it would have required highly specialized training. Unfortunately this facet removes a valuable tool from the programmer's arsenal for solving problems. Unless the problem is fairly simple, it may be very difficult to track down under Generation Five. The obvious solution to this problem is to incorporate significant debugging aids within the application generator language to address this area.

Perhaps the best measure of the productivity gains provided by *this* application generator was the relatively brief time required to develop the code through unit testing. The elapsed time for this task was approximately three months, certainly a significant testimony to the potential value of an application generator.

Key considerations in selecting an application generator

The Hamilton Brothers FRS project team derived several factors that should be considered prior to the future use of an application generator. The initial point to be evaluated is the size and complexity of the impending project. A complex project requiring unique access methods, specialized processing techniques, and/or sophisticated online requirements may not be suitable for development using an application generator. Whereas for less complex systems, project teams should carefully consider the beneficial contribution of an

application generator to the system development and testing phases of a project.

Once a project team concludes that an application generator is an appropriate tool for the project, a careful evaluation of its technical capabilities should be performed. Application generators are inherently oriented to specific functional areas. The selection of the correct application generator for a given functional area is paramount to the ultimate success of the project. The design of a system is directly correlated to the technical capabilities of the underlying software. A limited software tool generally leads to an inefficient system design. Once an understanding of the application generator's capabilities and constraints is developed, the project team must be willing to live within those constraints. The following list identifies several key characteristics desirable in a mature application generator:

- Provide code usable for both online and batch processing
- Incorporate facilities to eliminate or minimize program dumps
- Provide integrated program looping and nested IF processing with self-documenting code
- Provide online inquiry/retrieval facility with data manipulation and selection features
- Support a structured development methodology
- Support standard file access methods
- Enforce consistent data element naming standards
- Provide high-level application oriented features such as automated batch balancing and file open or closing facilities
- Offer easy-to-use debugging aids
- Provide thorough and accurate documentation with sets of complete examples
- Provide a simple facility for interfacing with programs developed in a traditional language such as COBOL
- Offer flexible data storage techniques (i.e., variable length records with multiple record types per file)
- Provide an integrated online recovery facility for all software (including non-application generator programs) which updates an application generator data file
- Provide table processing features

CONCLUSIONS

Hamilton Brothers' use of Generation Five as an application generator language provided several key benefits. It did, however, introduce inefficiencies into the design process that could have been avoided under a COBOL or PL/1 based system. How effective was the project team in meeting the goals of FRS as outlined above? The user friendliness of the system is superior to what existed in the old batch system. Users have complete control, primarily through online techniques, of system parameters. The user report development facility (using the report writer associated with Generation Five) is yet to be attempted. However, it appears that simple reporting can be accomplished by selected users with minimal data processing support. More complex reporting will require direct data processing support. The table-oriented design of Generation Five provides extremely flexible data file modifications, thereby dynamically changing most edit criteria without programmer intervention.

Future technical modifications can be easily incorporated into FRS if the code is developed in Generation Five. Non-Generation Five online routines will still require extensive technical evaluation prior to implementation. The structured coding requirements of Generation Five were beneficial but were somewhat offset by limitations in the syntax.

The implementation of key development productivity tools such as online program development, report writers, screen generators, a data dictionary, and project task planning exceeded the original goals. These tools were critical factors in the success achieved by the project team.

The concept of application generators is an innovative technique for improving the quality and timeliness of system development. The facilities provided by Generation Five are beneficial as they exist today. However, the product could be greatly enhanced with a few additional capabilities such as more data manipulation and retrieval flexibility, interactive online processing, and more variable file processing options. American Management Systems has indicated their intention to address these areas in future releases of Generation Five. Generation Five offers a good tool for accounting applications. Its use for other application areas appears, by design, to be limited.

Requirements definition and its interface to the SARA design methodology for computer-based systems

by JAMES W. WINCHESTER
Hughes Aircraft Company
Fullerton, California
and
GERALD ESTRIN
University of California
Los Angeles, California

ABSTRACT

This paper presents results of efforts during 1979–1981 to integrate and enhance the work of the System ARchitects Apprentice (SARA) Project at UCLA and the Information System Design Optimization System (ISDOS) Project at the University of Michigan. While expressing a need for a requirements definition subsystem, SARA had no appropriate requirements definition language, no defined set of requirements analysis techniques or tools, and no procedures to form a more cohesive methodology for linking computer system requirements to the ensuing design. Research has been performed to fill this requirements subsystem gap, using concepts derived from the ISDOS project as a basis for departure.

INTRODUCTION

Research into requirements definition and design methodologies for Computer-based Information Processing Systems (CIPS) has been extensive. Some fundamental concepts have emerged:

1. *Hierarchical decomposition* from abstract descriptions to refined detail[1]
2. *Verification* analysis to ensure that each level of description is consistent with and traceable to adjacent levels[2]
3. *Simulation* as a legitimate analysis aid in detecting incomplete and inappropriate designs[3,4]

Many methodologies utilize notations that ease the burden of analysis and decomposition as well as provide a vehicle that enhances understanding and design freedom. Of principal interest are *pictorial* and *graphical modelling,*[5] specialized *textual* languages,[6,7,8] and *database* management of information.[9]

Less prevalent before the SARA research[10–13] was recognition of the following needs: (1) describing *attribute* requirements along with *process* and *function* requirements, (2) modelling CIPS *structure* as well as *behavior,* (3) separating models of the *environment* and the CIPS and modelling the environment along with the CIPS, and (4) constructing *tests* for requirements satisfaction as a necessary adjunct to defining the requirements. Nearly all methodologies concentrate separately on either the requirements phase or the design phase of the CIPS development cycle. That narrow concentration creates an artificial gap in notation and analysis between the requirement and design phases, generally resulting in ad hoc methods to bridge this gap. More coherence between the requirements definition and design methodologies is needed, not only to bridge this gap but to close or eliminate it.

In this paper, the authors discuss the results of efforts to relate requirements definition and design methodologies by integrating and enhancing the work of the System ARchitects Apprentice (SARA) Project at UCLA and the Information System Design Optimization System (ISDOS) Project at the University of Michigan. SARA[12] offered support to a designer in creation and analysis of multilevel models. While expressing a need for a requirements definition subsystem, SARA had no appropriate requirements definition language, no defined set of requirements analysis techniques or tools, and no procedures to form a more cohesive methodology for linking requirements to the ensuing design. The ISDOS Project's PSL/PSA System[9] offered support to problem statements, problem analysis, and management of resulting information but had no other support to give to the design process. Research has been performed to fill the SARA requirements subsystem gap, using concepts derived from the ISDOS project as a basis for departure. Neither the PSL/PSA nor the SARA systems were looked on as models of perfection in supporting computer-aided creation of complex systems whose behavior would satisfy customers' and designers' intents. They each offered some unique strengths but also needed each other's strengths.

This paper is organized as follows. The system development life cycle, specifications and requirements categories, and analysis aspects of requirements definition are first summarized.

The SARA methodology and the PSL/PSA system are then briefly described. The framework of SARA, augmented with a requirements definition subsystem derived from the concepts of PSL/PSA, is proposed as a viable approach to an integrated development methodology.

The requirements definition subsystem can be characterized by three components: (1) a *Requirements Definition Language (RDL),* (2) *Requirements Analysis Techniques and Tools,* and (3) *Requirements Definition Procedures.* Due to the limitations on publication length, the emphasis of this paper is on the RDL and its semantic foundation. Details of the complete requirements definition subsystem and its interface with the SARA methodology are found elsewhere.[13]

Finally, a brief summary of experience using RDL and the state of the support tool development is described.

OVERVIEW OF REQUIREMENTS DEFINITION ISSUES

The requirements for a CIPS must be recorded in some fashion to provide a means of communication between individuals and supporting tools involved in its design. This record consists of a set of specifications comprising language statements (natural or some special language), graphs, diagrams, and tables. Determination of the form and format of these specifications is an important issue. Its resolution is affected by the desired interface between the requirements specification and the succeeding design processes.

The requirements specification must include three categories of requirements. First are function requirements. Function requirements specify the transformations that a system must perform. Second are process requirements. Process requirements specify coordinated sequences of functions. Third are attribute requirements. These are statements of constraints and performance parameters imposed upon elements of the CIPS.

A means must exist to analyze the requirements specification to ensure that certain criteria for a well-formed specification are satisfied. The specification should be *understandable*

to those who are providing the requirements information (the "customers") as well as those responsible for developing the proposed system (the "designers"). The information within the specification should be *consistent*; i.e., no subset of requirements should be incompatible with any other subset. The specification should be *complete* so that unintended value judgements can be avoided during the design process. The information within the specification should be *traceable* to the resulting design and implementation to verify that the resulting CIPS has addressed all requirements. The requirements should be *testable* to validate that the resulting design satisfies all of the requirements. The requirements should be *realizable* in the sense that there are no unattainable requirements which are detectable. Finally, the requirements should be specified so that there is *design freedom* allowed wherever possible.

SARA METHODOLOGY

The SARA methodology uses a set of tools and procedures to design computer-based information processing systems. The SARA methodology has evolved from research and development continued since the early 1960's at UCLA.[11]

An overview of the SARA methodology is illustrated in Figure 1. The methodology is characterized as requirement-driven; that is, requirements that the CIPS must satisfy are specified, and the design activity proceeds to create a system that can meet those requirements. The environment with which the CIPS is assumed to interact is explicitly defined at the beginning of the design process. Validation of a CIPS' design is meaningful only in the defined environment.

To describe and evaluate CIPS (and the set of decomposed subsystems) a collection of modelling tools is used.[10,11,12,14] A *structural* model identifies subsystems and their interconnections. A set of *behavioral* models[14] expresses the behavior of subsystems and their behavioral interrelationships. The Graph Model of Behavior (GMB) consists of two separate but interrelated *control* and *data* graphs to express behavior. An interpretation model is associated with each processor and data set. A GMB model can then be exercised through an interpreter that simulates the behavior represented in the model, providing a means to evaluate ensuing CIPS designs. A control flow analyzer[15] is used to detect pathologies such as deadlock.

Once a CIPS has been partitioned to the point at which the designer feels confident in understanding each subsystem and knowing how to fabricate it, the *composition* process begins. The designer composes the subsystems using validated building block models of existing hardware, software, and other elements that may be used to enhance analysis. The specifications for the building blocks should be consistent in form with top down requirements. In the limit, if a building block exists whose specification satisfies a requirement which was generated top down, its acceptance should be simple. Research is ongoing to discover canonical forms for specification of existing hardware and software elements.[16,17] The composed models of the subsystems are then analyzed and tested using the

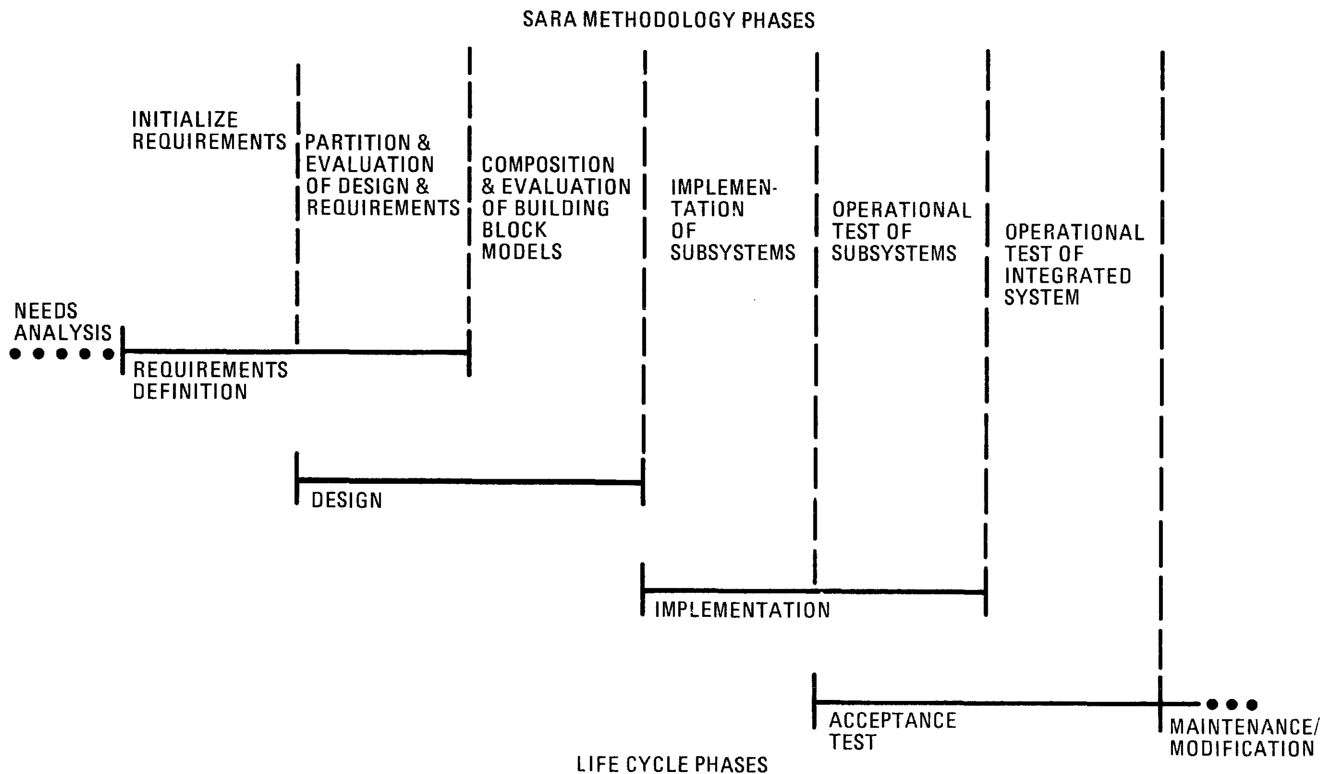


Figure 1—The CIPS development life cycle phases as constrained by the SARA methodology phases

modelling and simulation tools to verify that the subsystems and, ultimately, the complete CIPS satisfy all requirements. The physical implementation of the CIPS can then be fabricated directly from the building blocks used in the model of the CIPS.

PSL/PSA SYSTEM

The PSL/PSA system is a product of the Information System Design Optimization System (ISDOS) Project. The ISDOS Project is an ongoing research effort at the University of Michigan.

The Problem Statement Language (PSL) consists of a syntax and semantics for describing requirements according to a structured format of objects and relationships. Aspects of system structure, size, volume, dynamics, properties, data structure and derivation, and project management can be described. Included in the language is the capability to add descriptive English language comments and definitions of object attributes.[9]

The Problem Statement Analyzer (PSA) consists of all the computer software to process, analyze, and manage the PSL statements and the resulting database of PSL information. Final complete documentation of the PSL database can be produced by PSA semiautomatically in desired formats.

REQUIREMENTS DEFINITION FOR SARA

The *SARA methodology is based upon accurate specification of requirements for a CIPS being designed.* The methodology includes tools and procedural steps for decomposing, composing, and modelling CIPS to create a design that tries to meet the desired requirements and can be directly implemented. Figure 1 illustrates how the system development life cycle phases can be defined in terms of the SARA methodology phases. The requirements definition phase is seen to form the basis for all of the decomposition (refinement) activity. Thus the requirements specifications form a continuous stream of documentation of the CIPS from the most abstract customer-defined need for the CIPS down to the detailed refinement of subsystems, so that the designer can construct the subsystem from existing building blocks. In this context, the concept of distinct design specifications is not necessary; the design specifications can be a refined level of the requirements specifications.

A REQUIREMENTS DEFINITION LANGUAGE FOR SARA

The Requirements Definition Language (RDL) is used to express the requirements of a CIPS and to interface those requirements to the ensuing SARA oriented design. To determine what elements of information must be included in a requirements definition, one must have an appropriate model, or representation, of a computer-based information processing system (from requirements definition and design definition viewpoints) and an appropriate model of a requirements specification.

Computer-based Information Processing System Semantic Model (CIPSSM)

RDL's semantic model of a computer-based information processing system is based on SARA's structural and behavioral models of a system.[10,12,18,19] Most requirements for a system deal with conceptual information as opposed to physical realization. Therefore a semantic model, from a requirements viewpoint, should be concerned with conceptual constructs *onto which physical constructs can be mapped as part of the design activity.* The CIPSSM is derived from this basis.

CIPSSM primitives

The CIPSSM consists of six primitives that can be combined to model the structure and behavior of a CIPS and its environment. This representative framework allows the requirements, design, and implementation information to be associated with the model as the system development proceeds from requirements definition through implementation. Inherent in this modelling approach is the ability to perform a controlled refinement of the primitives to create a multilevel representation of a CIPS. At each refinement, more descriptive details are added to effect a progression from the conceptual requirements to physical realization.

The CIPSSM structural model primitives are *systems, dataflows,* and *connectors.* The CIPSSM behavioral model primitives are *functions, data-uses,* and *processes.* Figure 2 illustrates the graphical representation of the primitives and provides a brief description of the meaning of each primitive.

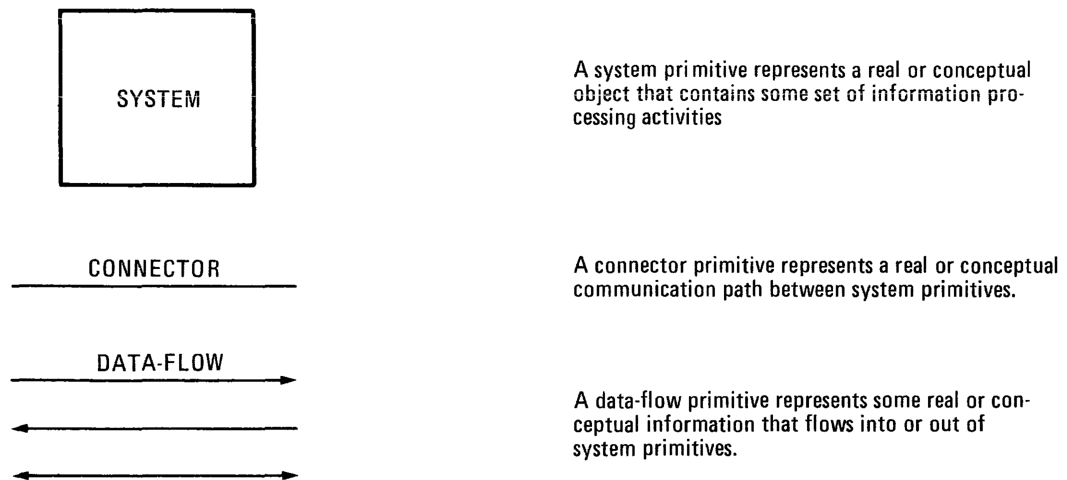
CIPSSM rules

The CIPSSM primitives interact with each other according to well-defined rules. The semantic rules are organized into three categories: (1) allowed relationships between the environment and CIPS domains of the design universe (the only allowed interface between the environment and the CIPS is through connectors, data-flows, and processes); (2) allowed relationships between primitives within each domain (e.g., a function can derive any number of data-uses); and (3) allowed decomposition relationships (e.g., a function can consist of any number of subfunctions). The RDL is designed to implement these semantic rules while the Requirements Definition Techniques and Tools are designed to enforce them.

Requirement Specification Model

The Requirement Specification Model (RSM) is a definition of the form and format of the requirements specifications for a CIPS. RDL is the principal language that will document the information that must be included in the RSM. The RSM ensures that the criteria for a well-formed specification are achievable and that the three categories of requirements are discernible. To perform this task, the RSM is set up to provide a means to describe a CIPS at all stages of its development and then automatically extract the function, process, and attribute requirements identified in the description. After the require-

CIPSSM STRUCTURAL PRIMITIVES



CIPSSM BEHAVIORAL PRIMITIVES

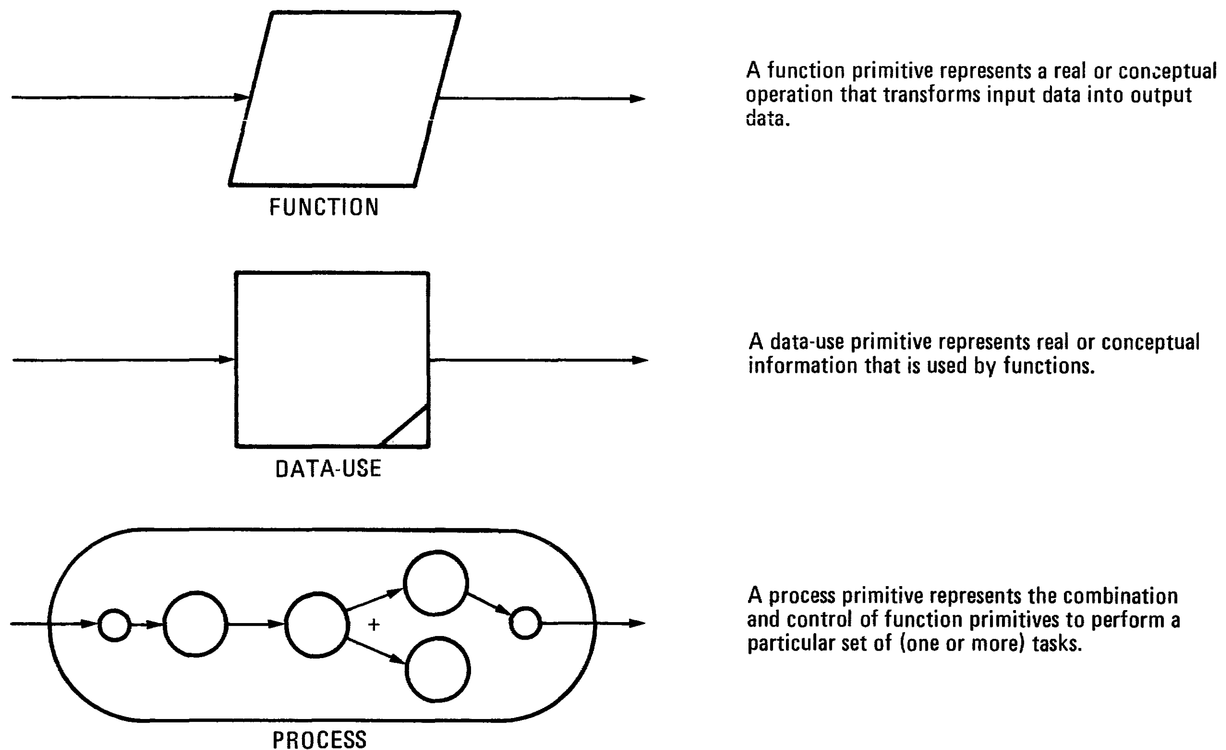


Figure 2—The graphical representation and definition of the CIPSSM primitives

ments are extracted, the tests for requirement satisfaction can be defined.

CIPS views

The RSM provides a description of four interrelated views of the CIPS. These views are decomposable in a structured fashion so that, within any particular view, the information at level (i + 1) is related to the view from level (i). The views adhere to the semantic rules of the CIPSSM. An example of the graphical representation of the four views using CIPSSM primitives is illustrated in Figure 3. The four views are designed to form a composite of the CIPS and its environment. All information expressed graphically in the four views and

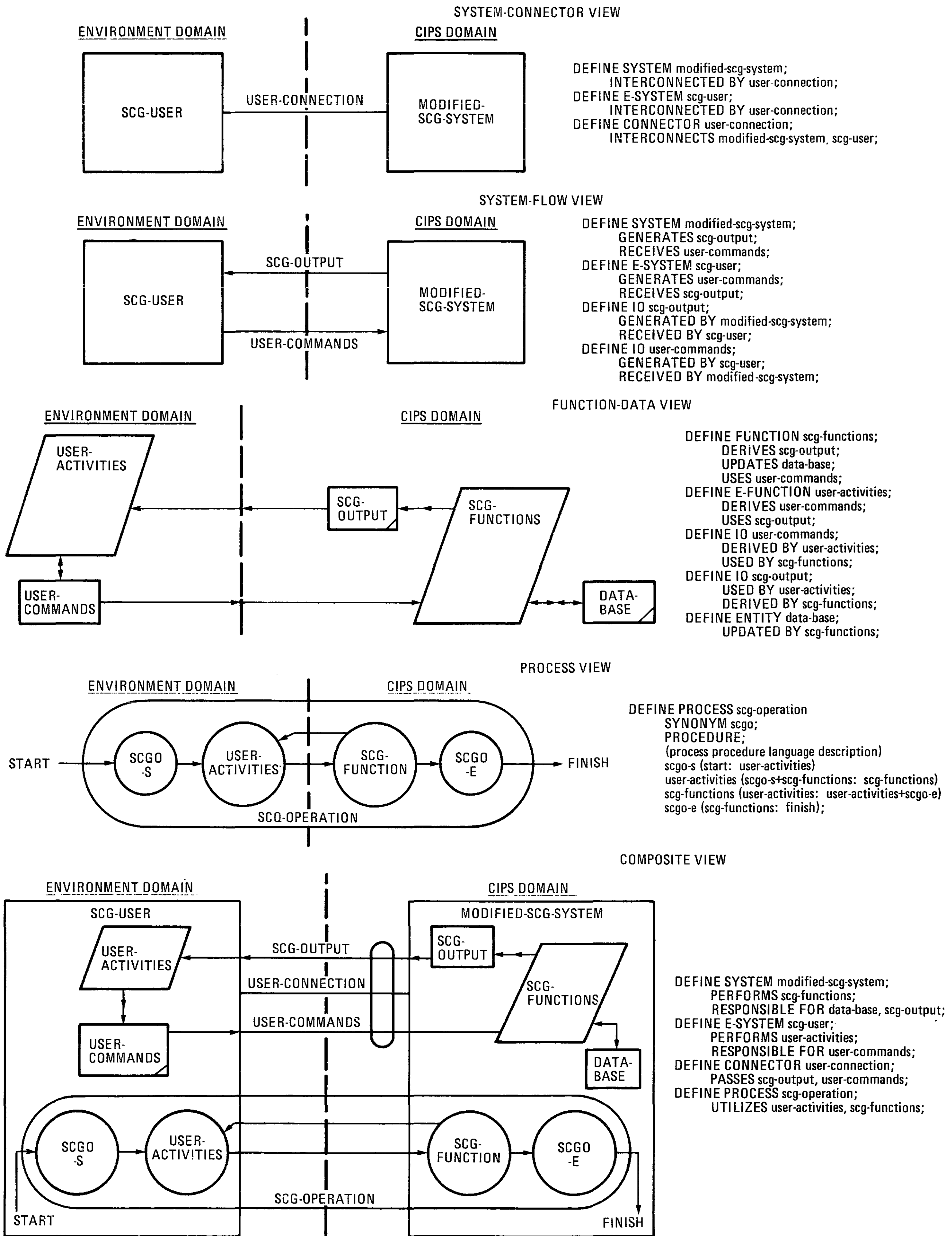


Figure 3—An example of the four CIPS views and the corresponding RDL representation for a modified structure chart graphics (SCG) system

their composite have a corresponding RDL textual representation. RDL also allows a means to express detailed information about CIPSSM primitives that cannot be represented graphically.

CIPS view decomposition

Requirements decomposition proceeds from a primarily logical description of a CIPS to a physical description of the CIPS that ultimately represents the actual design. The four descriptive views of a CIPS are oriented toward gradiated levels of logical versus physical description, between the views, and within the views.

The requirements decomposition process proceeds in parallel with the design process after the initial requirements specification is completed. The requirements decomposition is made *as the result of design decisions*. Figure 4 illustrates the graphical representation of a decomposition of the function-data view shown in Figure 3. At each level of decomposition, the designer gets a new set of requirements to respond to; however, since the new set of requirements were derived and documented from the previous level of requirements, continuous traceability is maintained between one step of decomposition and another. The RSM, built upon the CIPSSM primitives and documented by the RDL, is appropriate for the description of the CIPS at all stages of the requirements definition phase of the development cycle, as displayed in Figure 1.

Requirements extraction

Once a satisfactory *requirements specification level* is defined, using the CIPS graphical views and corresponding RDL descriptions, a complete set of function, process, and attribute requirements can be extracted. The function requirements are the RDL descriptions of the function primitives that are portrayed in the function-data view. The process requirements are the RDL descriptions of the process primitives that are portrayed in the process view. The attribute requirements are the RDL descriptions of attributes associated with all of the primitives in all four views. The requirements extraction concepts are illustrated in Figure 5. The function requirement extracted from the initial CIPS description of Figure 3 is presented in Figure 6(a).

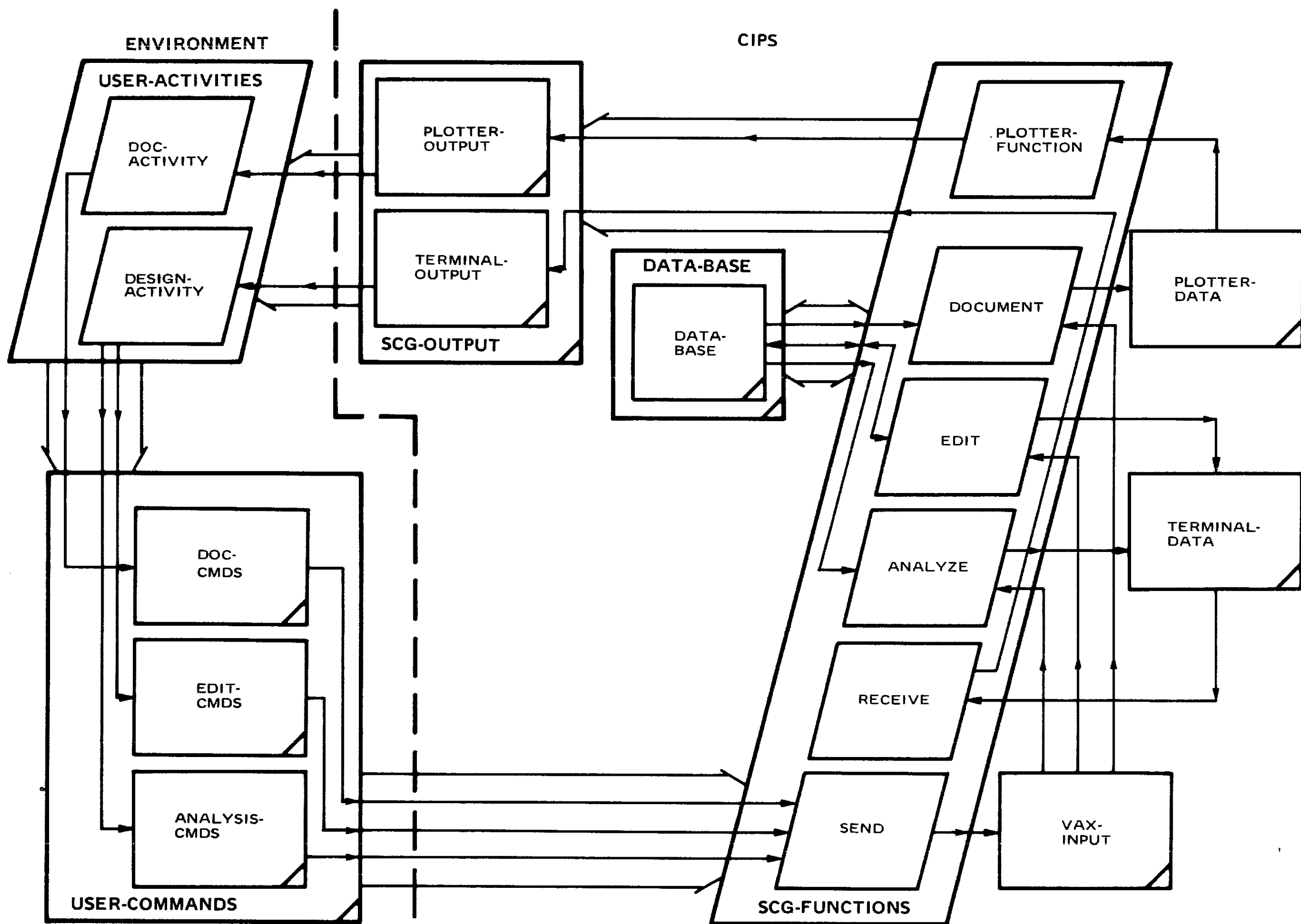


Figure 4—An example decomposition of the function-data view shown in Figure 3

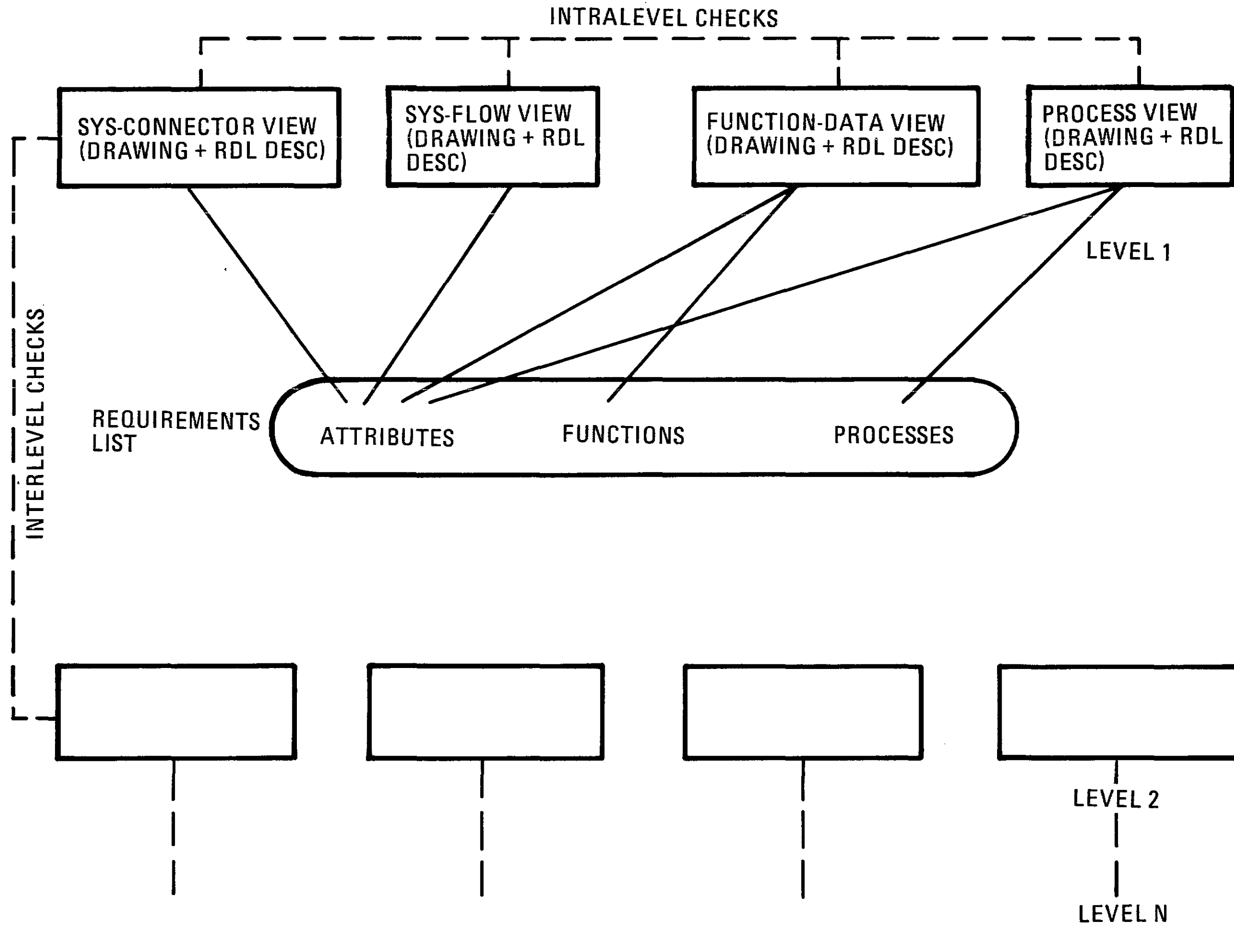


Figure 5—At every level of specification, requirements are extracted from the complete CIPS description as represented in the four views

Requirements tests

At every requirements specification level, a list of function, process, and attribute requirements is generated. At every level, each unique requirement makes one or more associated tests mandatory, so that the requirement can be verified and validated. The tests consist of test procedures and criteria to judge the outcome of the tests (i.e., whether the requirement is satisfied or not). In addition, the tests should reflect the multilevel decomposition of the requirements by becoming more detailed during refinement.

The nature of the test procedures and criteria depends upon the category of requirements being tested. RDL provides constructs permitting definition of initial conditions, final conditions, and procedures for defining how the test cases should be executed and acceptance criteria for determining how the outcome of each test case will be evaluated. A test for the requirement extracted in Figure 6(a) is described in Figure 6(b).

Requirements specification outline

The body of the requirements specification is organized by requirements specification level. Each level of the document consists of the four drawings of the CIPS views; a composite drawing of the four views; the function, process, and attribute requirements and tests; and any supporting RDL sections referenced by the requirements descriptions (e.g., definition of the 'user-commands' of Figure 6(a) and 'scg-ftl-ac' of Figure 6(b). At the end of the requirements specification, documents referenced by the requirements specification body that are not expressed in RDL are found.

Requirements Definition Language Characteristics

The Requirements Definition Language is based upon the same syntactical constructs as the Problem Statement Language (PSL) of the ISDOS project.[9] The language consists of *objects, relationships, descriptors,* and *associators.* Objects are essentially equivalent to nouns in English—they represent the things being described (CIPSSM primitives and associated information elements) when describing the CIPS. An *object-type* is a generic class of objects. The object-types are categorized according to what aspect of the RSM and CIPSSM they support. In Figure 6(a) the object 'scg-functions' is an example of the object-type FUNCTION.

Relationships are the "verbs" of RDL—they define the as-

(a)

```
DEFINE FUNCTION                  scg-functions;
   USES              user-commands;
   DERIVES           scg-output;
   UPDATES           data-base;
   PERFORMED BY      modified-scg-system;
   UTILIZED BY       scg-operation;
```

(b)

```
DEFINE FUNCTION-TEST             scg-function-testl;
   TESTS             scg-functions, user-activities;
   DESCRIPTION;
     This test sequence is designed to provide
     a customer acceptance test for creating
     structure charts, as one of the necessary
     functions of the modified-scg-system.;
   INITIAL-CONDITIONS ARE        scg-ftl-1, user-cmd-seq;
   FINAL-CONDITIONS ARE          scg-ftl-2;
   ACCEPTANCE-CRITERIA ARE       scg-ftl-ac;
   PROCEDURE;
     DO.
       set initial-conditions to 'scg-ftl-1'.
       DO UNTIL final-conditions='scg-ftl-2'.
          execute initial-condition
                 'user-cmd-seq'.
       ENDDO.
       check acceptance-criteria 'scg-ftl-ac'.
     ENDDO.;
```

Figure 6—(a) A function requirement extracted from the initial CIPS description shown in Figure 3
(b) A test for the function requirement

sociations between objects based upon the allowed relationships between object-types as determined by the CIPSSM and the RSM. In Figure 6(a) USES is an example of a relationship associating 'scg-functions' and 'user-commands.' Relationships are designed to be complementary in the sense that if the object-types are interchanged in an RDL statement, an equivalent relationship can be formed.

Descriptors are RDL constructs which fall outside of the object, relationship character of most RDL statements. Descriptors are associated with objects but are not objects themselves. Descriptors are an important source of redundancy which is counted on to help reduce the gap between intent and designed behavior. The most common *descriptor-type* is the comment-entry. This descriptor-type consists of English language text, or any more formal language of the users' choosing, that can be used to further describe an object outside the realm of the object's relationships with other objects. Descriptors are associated with objects via *associators.* Associators and descriptors allow one type of extensibility within RDL by permitting any desired language (e.g., SARA's GMB expressions, program description languages) to be included in the RDL specification. The associator DESCRIPTION and its comment-entry is illustrated in Figure 6(b).

The RDL syntactical constructs are patterned after the constructs that are supported by the ISDOS Project's Metagenerator and Generalized Analyzer.[20] RDL *statements* are grouped together by *sections.* Each section defines an object name and the relationships of that object to all other objects in a specification, plus the descriptors associated with that object. Figure 3 illustrates simple RDL sections that describe CIPS views.

RDL implementation of CIPSSM

RDL implements the CIPSSM by providing constructs that allow all CIPSSM primitives and properties to be described, as well as maintaining a basis for the semantic rules. There are unique RDL object-types for primitives that exist in the environment domain versus the CIPS domain. Their purpose is to provide a mechanism for maintaining the universe partition in a CIPSSM description.

RDL support of RSM

RDL supports the RSM by providing constructs that allow the following: (1) the four CIPS views to be incrementally developed and then rigorously coupled (Figure 3), (2) easy extraction of requirements by category (Figure 6(a)), (3) association of tests with each requirement (Figure 6(b)), and (4) easy extraction of RDL sections for the organization of the requirements specification.

The RDL syntax permits the designer/customer to describe the objects and relationships associated with any one particular CIPS view and then later to add the relationships that couple the views.

REQUIREMENTS ANALYSIS TECHNIQUES TOOLS AND PROCEDURES

Analysis techniques have been defined as a set of checks on the information within the specification to determine its compliance to the criteria of being understandable, consistent, complete, traceable, testable, realizable, and allowing design freedom. These checks are driven by the CIPSSM rules, heuristics of the design activity, and the modelling and analysis power of SARA. The requirements definition activity is designed to be extensively supported by computer-aided tools. The tools can be categorized into five functional areas: (1) an RDL interpreter and editor (plus associated database storage mechanism), (2) database query, (3) analysis checkers, (4) graphics support of the CIPS view constructions and translation to corresponding RDL, and (5) CIPS view to SARA model construction and translation.

An eighteen-step procedure has been developed that serves as a guideline on how to construct a model of the CIPS using RDL and graphics, perform analysis checks, and decompose a specification.

USAGE EXPERIENCE

The ISDOS Project's META System[20] was used to implement an RDL interpreter and editor (including database management system) and query system. A collection of SARA tools (fully operational and accessible on the ARPANET) exist and include the GMB Simulator and Control Flow Analyzer, plus structural modelling tools and a sophisticated help system. Using this continually expanding support environment, RDL has been applied in two practical CIPS specification activities since early 1981. The experience of these applications revealed that the modelling scheme was well liked, particularly

for the separation of environment and CIPS, and the provision of integrated multiview aspects. However, the following improvements are considered essential:

1. A graphics interface is needed to reduce the tedium of CIPS view construction and translation into RDL.
2. A better user interface to the support environment is needed.
3. Automated utilities to support test construction are needed to lessen the difficulty of creating appropriate requirements tests.
4. The ease with which RDL can express specifications of existing building blocks (e.g., manufacturers chip specifications) must be tested.

REFERENCES

1. Ross, D. T., and K. E. Schoman, Jr. "Structured Analysis for Requirements Definition." *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, January 1977.
2. Alford, M. W. "Requirements for Distributed Data Processing." *Proceedings of First International Conference on Distributed Data Processing,* IEEE, 1979.
3. Alford, M. W. "Software Requirements Engineering Methodology (SREM) at the Age of Two." *COMPSAC 78 Proceedings,* November, 1978.
4. Willis, R. R. "DAS: An Automated System to Support Design Analysis." *Proceedings of the Third International Conference on Software Engineering,* Atlanta, Georgia, May 1978.
5. Alford, M. W., and I. F. Burns. "R-nets: A Graph Model for Real-time Software Requirements." In *Proc. Symp. On Comput. Software Eng., MRI Symp. Ser.,* Vol. XXIV, Polytechnic Press, Brooklyn, NY.
6. Riddle, W. E., J. C. Wileden, J. H. Sayler, A. R. Segal, and A. M. Stavely. "Behavior Modeling During Software Design" *IEEE Transactions on Software Engineering,* Vol. SE-4, No. 4, July 1978, pp. 283–292.
7. Zave, P., and R. T. Yeh. "Executable Requirements For Embedded Systems." *Proceedings of the Fifth International Conference on Software Engineering,* San Diego, California, 1981.
8. Heninger, K. "Specifying Software Requirements For Complex Systems: New Techniques and Their Application." *IEEE Transactions on Software Engineering,* Vol SE-6, No. 1, January 1980.
9. Teichroew, D., and E. A. Hershy III. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, January 1977, pp. 41–48.
10. Gardner, R. I. "A Methodology for Digital System Design Based on Structural and Functional Modeling." Ph.D. dissertation in Computer Science, University of California, Los Angeles, January 1975.
11. Estrin, G. "Modeling for Synthesis—The Gap Between Intent and Behavior." *Proceedings of the Symposium on Design Automation and Microprocessors,* Palo Alto, California, February 24–25, 1977, IEEE, Piscataway, New Jersey, 1977, pp. 54–59.
12. Estrin, G. "A Methodology for Design of Digital Systems—Supported by SARA at the Age of One." *Proceedings of the National Computer Conference,* Anaheim, California, June 1978.
13. Winchester, J. W. "Requirements Definition and Its Interface to the SARA Design Methodology for Computer-Based Systems." UCLA Technical Report, UCLA-ENG-8092, January, 1981.
14. Razouk, R., M. Vernon and G. Estrin. "Evaluation Methods in SARA—The Graph Model Simulator." *1979 Conference on Simulation, Measurement and Modeling of Computer Systems,* Boulder, Colorado, August 1979.
15. Razouk, R. R. "Computer-Aided Design and Evaluation of Digital Computer Systems." UCLA Technical Report, UCLA-ENG-8055, February 1981.
16. Penedo, M. H. "The Use of a Module Interface Description in the Synthesis of Reliable Software Systems." UCLA Technical Report, UCLA-ENG-8091, January 1981.
17. Vernon, M., D. Patel, and G. Estrin. "A SARA Building Block Model: Am2909 Microprogram Sequencer." UCLA Internal Memorandum #210, October 1981.
18. Campos, I. M., and G. Estrin. "SARA Aided Design of Software for Concurrent Systems." *Proceedings of the National Computer Conference,* Anaheim, California, June 1978.
19. Razouk, R., and G. Estrin. "Modeling and Verification of Communication Protocols in SARA: The X.21 Interface." *IEEE Transactions on Computers,* Vol. C-29, No. 12, December 1980, pp. 1038–1052.
20. Teichroew, D. "Overview of the META System." ISDOS Research Project, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor, Michigan, ISDOS Project META-1 Memorandum, May 1977.

The role of requirements analysis in the system life cycle

by YUZO YAMAMOTO, RICHARD V. MORRIS, CHRISTOPHER HARTSOUGH, and E. DAVID CALLENDER
California Institute of Technology
Pasadena, California

ABSTRACT

One of the problems that personnel from the computer industry face today is to find the proper role of requirements analysis in the design and implementation of information-intensive systems so that the results of that activity may be effectively transferred to the rest of the life cycle. This paper addresses the problem by examining the life cycle process in terms of the various viewpoints that human beings use. The interplay between human capabilities and limitations for dealing with the problems of design representation and the increasing complexity of modern information-intensive systems is discussed. The concept of viewpoints around a life cycle wheel that are used throughout the entire life of the information-intensive system is introduced and used to define the functions performed during requirements analysis. Finally, the concept of a system-engineered set of techniques and tools to support the life cycle activities is proposed.

INTRODUCTION

The scope of this paper is the design and implementation of information-intensive systems. Such a system is one where the use and production of information is either a major function or a major component of the control of the process. Such a system usually has as its components hardware and human beings, using software and procedures, respectively.

Historical Background

The development of information-intensive systems is still a difficult task despite progress in relevant methods and tools. One of the early developments in attempts to better manage the development process was the recognition of a system life cycle. This allowed a phased approach[1,2] in managing the development of information-intensive systems from the definition of initial requirements through operation.

Historically, the phases receiving the most attention have been implementation, integration, and test. The importance of the early phases has been recognized,[3,4] partly because of the increased attention to the entire system life cycle (as opposed to preoccupation with a particular part of it). These early phases are called by various names, such as requirements analysis, logical design, or systems analysis. A few tools are being used during the early phases. For example, successful applications of PSL/PSA[5] have been reported.[6,7]

One problem encountered by users of requirements analysis techniques is to determine how to transfer the results of the requirements analysis to the later phases of the life cycle. In order to accomplish this successfully, the execution of the system life cycle itself must also be viewed as an information-intensive system and the role of requirements analysis identified in that respect.

Objectives of This Paper

The main objective of this paper is to identify the proper role of requirements analysis in the system life cycle. The second objective is to present an improved life cycle model based on the concept of viewpoints. The third objective is to present the concept of a "techni-kit" to support the life cycle activities.

Structure of the Paper

First, the fundamental causes of the problem, i.e., the complexity of modern information-intensive systems, and the limitations of human beings in handling a complex situation directly are discussed. Next, the viewpoint dimension of characterizing the distinct activities of the life cycle is introduced. Based on the viewpoint dimension, the life cycle wheel model is introduced, and the role of requirements analysis is defined in that framework. Finally, a life cycle support facility, called a *techni-kit,* is proposed.

HUMAN CAPABILITIES AND LIMITATIONS

We believe that a major source of problems faced during the design and implementation of an information-intensive system is the conflict between the complexity of such systems and human limitations in concurrently handling large amounts of complex information.

Today's information-intensive systems are complex. Instances of complexity can be seen in a number of ways. Typically, systems are made up of many parts, related to one another in many ways. The same is true of many of their components. These systems are often required to meet concurrently a number of distinct but interacting needs. Moreover, a system can often be operated in any of several different ways to meet a particular need.

On the other hand, there are characteristic limitations to human capability in dealing simultaneously with large numbers of concepts and relations. Over a period of time a person can deal with a great number of intricately related concepts, but with only a few at any particular time.[8] There is considerable evidence that indicates that this limit is a small number.[9] This human limitation directly affects the ways in which complex system developments and applications can be conducted.

Methods and tools that are developed to support designers and implementers of information-intensive systems must take explicitly into account two conflicting properties: complexity and human intellectual characteristics. One way to do this is by using (1) conceptual models that provide structure and (2) computer-based tools that augment human logical and clerical capabilities. In this manner the designer can focus individually on a series of small parts of the system with the expectation that the pieces of the system can be combined into an integrated whole. To aid the designers and implementers we have extended the life cycle model and provided a structure for the methods and tools they use.

DIMENSIONS FOR DESCRIBING DEVELOPMENT PROCESS

Any model of the life cycle process must include the following distinct dimensions:

1. Phases: When in time an activity occurs.
2. System level: Where in the hierarchy of the system structure attention is being focused.

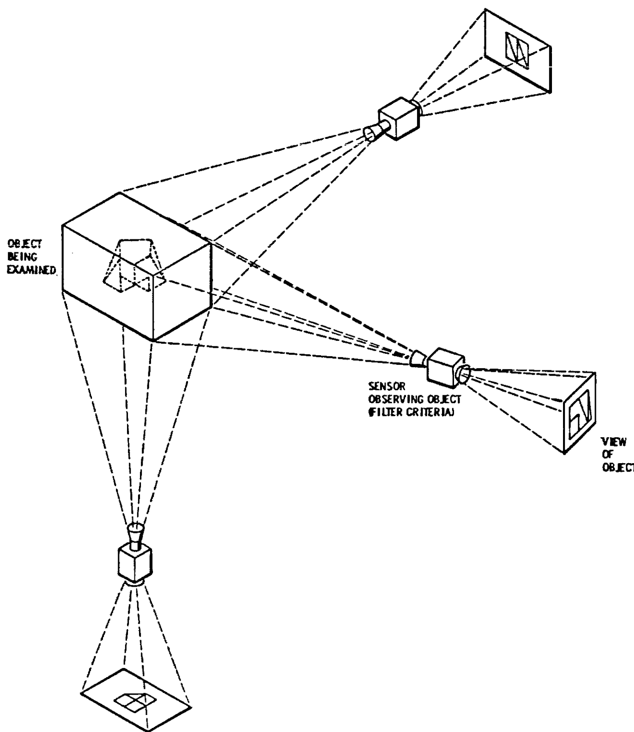


Figure 1—Taking a view

3. Viewpoint: The context in which one is currently working.

In this paper one part of a complete model of the life cycle process, the formal structure of viewpoints, is examined. The formalization of this concept is necessary because an important way that human beings deal with complexity is to work with views. Formalization of viewpoints forces the articulation of fundamental, hidden assumptions that different individuals use during the life cycle process. Making these hidden assumptions public facilitates the exchange of information (results of work) between individuals during the life cycle process.

A viewpoint is the position of an individual taking a view. A view is the result of filtering the available information about a situation and selecting a subset that is useful to an individual in doing a particular task. The information in the view may be formatted in a particular way to increase its usefulness. Figure 1 illustrates this concept.

Filtering may be considered a process operating on a stream of inputs. Operations of the following kinds are performed on each item in the stream:

1. Comparing the input items with a set of filtering criteria
2. Tagging it with additional information that states which (of several possible) output stream(s) or output set(s) the item will be inserted into
3. Separating the filtered items or copies of them into the individual output streams or sets that will subsequently receive different kinds of processing

As Figure 1 indicates, when a physical object is being examined, the observer's particular location relative to the object is one of the factors that determines the subset of information that the observer can see. The location of the observer is thus a part of the filtering criteria.

Important aspects of the process of taking a view include

1. The location of the observer (vantage point)
2. The other filtering criteria (process)
3. The data to be filtered (input)
4. The resultant view (output from the viewing process)

The concept of viewpoint will be used as a basis for a model of system development process in the next section.

LIFE CYCLE WHEEL MODEL OF SYSTEM DEVELOPMENT

In conventional life cycle models, the role and context of the individual has been largely ignored. To correct this omission, a structure of viewpoints is associated with the development process. The six top-level viewpoints, which form the life cycle wheel model of system development, are illustrated in Figure 2 and defined below. Such viewpoints can most easily be introduced in terms of the activities associated with them. It is important, however, to note that the essence of the viewpoints is not the activities but the continued, vested interest that a group of people develop as they undertake their normal activities. To facilitate understanding, an analogy of the process of building a manufacturing facility for a particular product is given for each viewpoint.

The *user needs* viewpoint is taken when unanalyzed user needs are sought, captured, and recorded. This may also be called the *buyer's viewpoint.* This viewpoint is interested in results—solutions to the specific problem presented. The viewpoint also sees and levies constraints under which the system must operate. Individual users often express their needs in terms of scenarios. A scenario describes how the system will operate to achieve a particular purpose and the situation and environment in which the system will be applied. It is also a temporal exposition of the interdependent activities

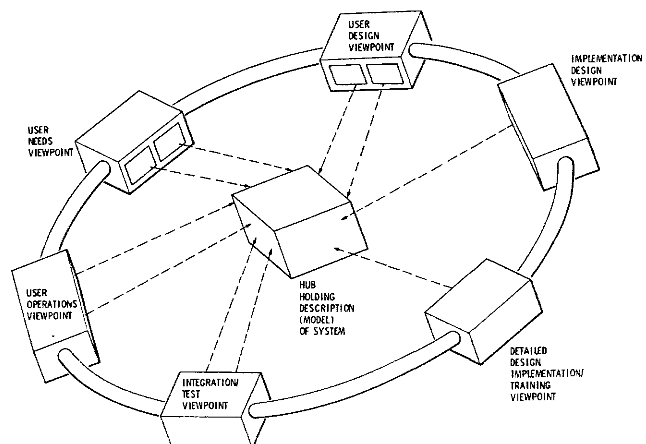


Figure 2—The life cycle wheel model

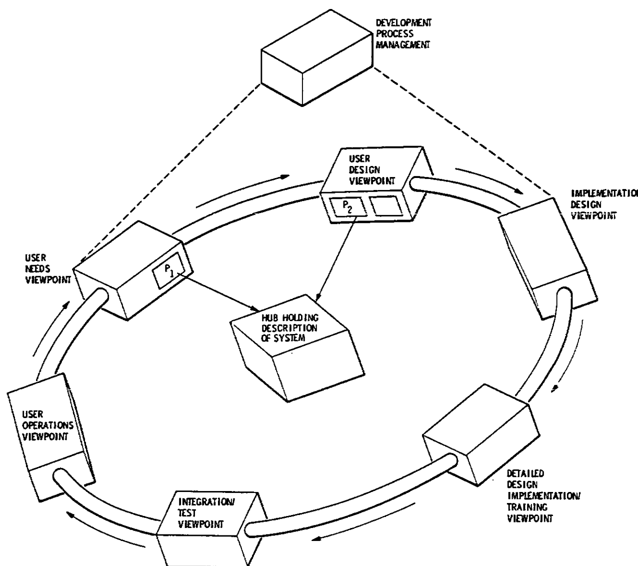


Figure 3—The project management viewpoint

of the environment, the system, and its operators in the accomplishment of a particular purpose. In addition to scenarios, individual users may have needs or desires for certain other characteristics of the system, such as throughput capability, mean time between failures, and adaptability. In a manufacturing environment, the analogous activity is market research.

The *user design* viewpoint is principally concerned with extracting a common, logically consistent set of requirements from those expressed by individual users. From this viewpoint, *system requirements* are defined. These requirements provide an envelope of services that encompass all reasonable/acceptable demands by individual users. From this viewpoint, assessments are also made of the stability of requirements and of the effect that changes in requirements will have on the system design. The manufacturing analogy for this viewpoint is product design.

The *implementation design* viewpoint is principally concerned with laying out an overall structure, or architecture, of components to meet user design requirements. Sometimes work done from the implementation design viewpoint directs the attention of the user design viewpoint to an incomplete area in the design. This work leads to *derived requirements* that must be incorporated into the user design. Although frequently perceived as different, derived requirements differ from other user requirements only in their date of discovery. The same type of analysis needs to be performed on all user requirements. The manufacturing analogy is the engineering of the manufacturing plant.

The *detailed design, implementation, and training* viewpoint is concerned with the specifics of constructing the system according to the architecture developed from the implementation design viewpoint. Training is included here because it provides the human components of the system with the guidelines and procedures they will use during system operation. The manufacturing analogy is the construction of the plant and training of the workers.

The *integration and test* viewpoint is concerned with the fitting together of implemented (or trained) parts of the system to produce a correctly-operating, verified, and validated whole that is ready for use in achieving the users' objectives. In manufacturing, pilot production and product test are the closest analogies.

The *user operations* viewpoint is concerned with applying the system, as constructed, to achieve the users' objectives. The experience gained in this viewpoint is a rich source of user needs for the next (or modified) system. Use of the plant to produce the product in volume is the manufacturing analogy.

One common viewpoint has been omitted from the present picture of the model—the *maintenance viewpoint.* The model includes maintenance as a part of operations and restricts its scope to

1. Identifying and evaluating problems within the system
2. Adjusting previously designed controls to keep the system in tune
3. Replacing failed parts with identical spares

If changes in the design of the system or its component are needed, the design is recycled as necessary through the life cycle to design and implement the changes.

There is another important viewpoint associated with the development process, the viewpoint of *project management.* This viewpoint is illustrated in Figure 3. Project management is able to plan and control the development process by viewing and monitoring the efforts of the major inline development activities performed from the six viewpoints.

The viewpoints have been described in terms of the activities performed. However, as pointed out earlier, the essence of the viewpoints is not the activity, but the continued, vested interest of a group of people involved in the life cycle activities. Two important concepts that arise out of the life cycle wheel model are identified and explained in the following subsections: information transfer and locus of principal activity.

Information Transfer Between Viewpoints

The first concept derived from the life cycle wheel model is that of information transfer between viewpoints. Since the context in which an individual works has been formalized, it is necessary to consider how information is transferred between different viewpoints. When transfers of information are made between viewpoints that are not adjacent, our experience indicates that misinterpretations and misuses of the information are more likely. For this reason, only information transfer between two adjacent viewpoints is considered.

Information exchange and information mappings are the two important kinds of information transfer. An information exchange is a formal process of handing information from one viewpoint to another. It can be further subdivided into feed-forward and feedback. Feed-forward is the transformation of a particular element of information from a viewpoint to the next viewpoint (along the general direction of the locus of principal activity, described in the next subsection), and feedback is the reverse. Mappings are relationships established

between the contents of information items of two viewpoints. A consistency check or requirement traceability between views at different viewpoints is an important use of such mappings.

Locus of Principal Activity

The second concept is that of locus of principal activity. The locus of principal activity is the time-ordered sequence of major development activities. It is the replacement for the idealized concept of life cycle phases used in the conventional life cycle model. If the development process were to proceed in the idealized manner, then the development phases would correspond directly to the six major viewpoints. Rarely does the idealized occur. Iteration and concurrent activities make the idealized model invalid.

The separation of viewpoints from life cycle phases is one of the major differences between the life cycle wheel model and the classical development model. It has important implications. First, the viewpoint structure exists throughout the complete life cycle of the information-intensive system. Second, the separation allows the model to explain and show the place for iteration between development activities and concurrent development activities.

The locus of principal activity is the bridge between viewpoints and development phases. Treating phases in this manner focuses project management's attention on several factors, all of which require management:

1. The existence of iteration in the development process
2. The oscillation of the focus of activity back and forth between viewpoints
3. The existence of concurrent development activities (at least two major efforts existing simultaneously in time)
4. Ensuring consistency between the work done from different viewpoints (requirements traceability)

REQUIREMENTS ANALYSIS IN THE LIFE CYCLE WHEEL

Based on the life cycle wheel model, requirements analysis is viewed as a design activity from a user viewpoint. This design is synthesized from various (incomplete, inconsistent) user scenarios and other expressions of needs. The emphasis is on what functions the system is to perform and how the system interacts with the users. It is helpful in distinguishing between *user* design and *implementation* design to say that, from a user design viewpoint, the functions are performed "as if by magic." A major pitfall to be avoided is the temptation to specify implementation details in the statement of the user design. Use of implementation information should be allowed only as a communication aid or for enunciation of constraints that the users impose. An example of such a constraint would be a de facto selection of computer hardware. This type of implementation design information should be considered as implementation information that appears during the user design phase. To further illustrate the importance of being able to distinguish between views and phases, we point out that it is impossible to finish the user design (in the form of user how-to manuals) until the final stages of implementation.

Requirements analysis (user design) is an important activity at both the information systems level and the software engineering level. As an increasing number of system levels are added to the description of the product system, requirements analysis will first (going along the locus of principal activity) be performed at the information system engineering level for the top system levels of the product system. Later, requirement analysis will again be performed at software subsystem levels. The user design at the system level is a major part of the user needs at the software system level.

A classic problem of requirements analysis is ensuring that an internally consistent and complete user design has been prepared. The model delineated in the previous section provides two major features to assist with this problem. The first is the application of consistency mappings to verify that all the scenarios can be supported by the functional features of the user design. In this manner the user design may be tested for completeness using the user scenarios. Internal consistency is a more difficult test to apply. The application of the life cycle wheel model will assist the designer in the check for internal consistency. However, additional design rules that are dependent on the product system need to be applied. The ultimate test for internal consistency is an application of the consistency mapping—from user design around the life cycle wheel to the successful use of the product system by the users against all of the required scenarios.

TECHNI-KIT

During the life cycle activities, methods and tools are used to support the human being in the design or implementation of an information-intensive system. There are two parts to such support: methodology used in thinking; and tools to capture, store, and manipulate the products of thought. Both methodology and tools are used to amplify the effectiveness and productivity of the human being doing design and implementation. The methodology may be divided into

1. Theories
2. Methods (how theories are applied)
3. Criteria (for judging the development and its product)

Supporting tools may be manual (such as paper and pencils) or computer-based. We are principally interested in the computer-based tools. Such tools can provide the enhanced speed, capacity, and accuracy needed to augment the human thought process in the development of large, complex systems.

A number of methodologies and supporting tools are usually pertinent to the design and implementation of any particular information-intensive system. The application of such a collection of methodologies and tools can be made more effective if its components are selected and shaped to form an integrated, system-engineered set of methodologies and supporting tools. We call such a set a *techni-kit.* We believe that many of the components of a particular techni-kit can be drawn from a reservoir of techni-kit resources that are generally applicable to a class of information-intensive systems. The

generation of any particular techni-kit involves two important, but not as yet well understood, tasks: (1) the particularization of the methodologies and supporting tools for the task at hand (modify the candidate components of the techni-kit to be most effective for the particular task) and (2) the design and implementation of that particular techni-kit in an efficient and rapid manner.

Techni-kits have been proposed, and some have been built and used.[10,11] To date, most of the techni-kits that the authors are aware of (including the ones that they use) are not well engineered, are incomplete, and place large clerical demands upon their users. Formalizing the concept of a techni-kit can achieve a better balance between the efforts spent on methodology and the efforts spent on tools.

CONCLUSION

The life cycle wheel model of the development of an information-intensive system has been introduced; it is based on the concept of viewpoint. In conventional life cycle models for information-intensive systems, the role of requirements analysis has been unclear. Further, such models have had difficulty in dealing with concurrent activities and iterations. By introducing the concept of a viewpoint and mappings between the results of work done from different viewpoints, these issues can be clarified. In addition, the context in which an individual works is recognized as having a major impact on the development of an information-intensive system. By adding the dimension of viewpoint to the traditional dimension of phases and system levels, a richer life cycle model has been created.

Requirements analysis is recognized as a design activity from the users' viewpoint in the extended life cycle model. Because of the increasing complexity of function and use of information-intensive systems, the insertion of a user design between user needs and implementation design allows the engineers, designers, and implementers to better handle the ever increasing levels of complexity of information-intensive systems.

The last concept introduced in this paper is that of a techni-kit. It provides a structure for considering the techniques and aids that exist or have been proposed for use in the creation of complex information-intensive systems. Methodologies and computer-based tools are two main components of any good techni-kit. They must complement one another if the techni-kit is to be an effective aid to engineers, designers, implementers, and managers.

ACKNOWLEDGMENTS

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

REFERENCES

1. Metzger, P. W. *Managing a Programming Project.* Englewood Cliffs, New Jersey: Prentice-Hall, 1973.
2. Biggs, C. L., E. G. Birks, and W. Atkins. *Managing System Development Process.* Touche Ross Management Series. Englewood Cliffs, New Jersey: Prentice-Hall, 1980.
3. Teichroew, D. "A Survey of Languages for Stating Requirements for Computer-based Information Systems." *AFIPS, Proceedings of the Fall Joint Computer Conference* (Vol. 41), 1972, pp. 1203–1224.
4. Couger, J. D. "Evolution of Business Systems Analysis Techniques." *ACM Computing Surveys,* 5, (1973), pp. 167–198.
5. Teichroew, D., and E. A. Hershey. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering,* SE-3, (1977), pp. 41–48.
6. Winters, E. W. "Experience with Problem Statement Language: A Language for System Requirements and Specification." *Proceedings of IEEE Compsac,* Chicago, November 5–8, 1979, pp. 283–288.
7. Farny, A. M., et al. "An Application of Computer Aided Requirement Analysis of Deep Space System." *AIAA Computers in Aerospace III Conference,* October 26–28, 1981, San Diego, California, pp. 531–536.
8. Wickelgren, W. A. *Cognitive Psychology.* Englewood Cliffs, New Jersey: Prentice-Hall, 1979, p. 384.
9. Miller, G. A. "The Magical Number Seven Plus Or Minus Two: Some Limits on Our Capacity for Processing Information." *Psychological Review,* 63 (1956), pp. 81–87.
10. Teichroew, D., E. A. Hershey, and Y. Yamamoto. "Computer-Aided Software Development." *Software Reliability, Part 2.* Maidenhead, Berkshire, England: Infotech International Limited, pp. 299–368.
11. Ramamoorthy, C. V., and H. H. So. "Software Requirements and Specification: Status and Perspective." In V. Ramamoorthy and R. Yeh (eds), *Tutorial: Software Methodology.* New York: IEEE Computer Society, 1978.

Application generators: an introduction

by JERROLD M. GROCHOW
American Management Systems
Arlington, Virginia

ABSTRACT

Application generators represent a new class of software development tools, which may yield the next order-of-magnitude productivity improvement in systems design, programming, and maintenance. This paper provides an introduction and bibliography for the topic. Current references to more than 50 articles and publications, as well as to some two dozen products, indicate the extent of recent interest in this topic.

INTRODUCTION

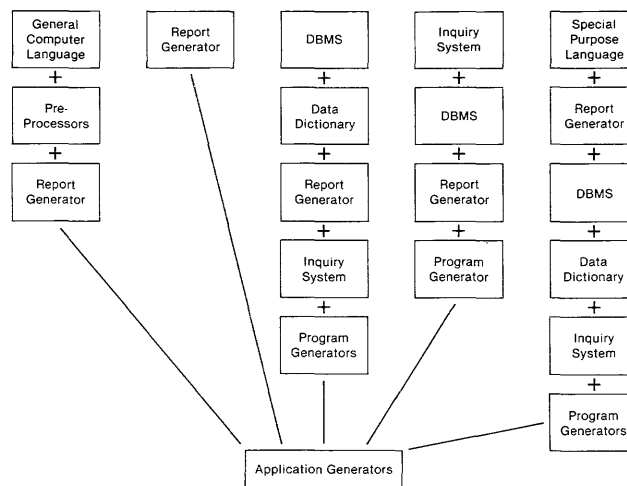
As recently as 18 months ago, an article in a major computer publication stated that application generation was a technology that was unlikely to have any significant impact on the development of computer systems.[14] Today, almost 50 companies have products which they call application generators[46] (see product description bibliography). They operate on a wide variety of computers ranging from microprocessors to large-scale mainframes. Almost every major manufacturer of computers has or is working on application generator products. IBM, with perhaps the largest research and development budget of all, is devoting substantial resources to expanding its offerings in this area.[15] In short, we have seen a revolution within the past two years in the types of software-development tools that are being offered to the industry. Rather than producing just new computer languages, software suppliers are aiming their sights much higher at a new generation of systems software that transcends the concept of both procedural and nonprocedural languages and takes us into the new realm of complete application specification and application generators.[6]

Application generators are a natural outgrowth of our search for better ways to develop not just single programs but complete application systems. While the term "application generator" is currently applied to a wide range of products, it is generally used to connote a development tool or tools whose input is a specification not just of a single program but of an entire system, including the database, transaction formats, reports, programs, and job-control logic. In the ideal case, the output of an application generator would be a complete application system in executable form. However, the current reality is somewhat less than the ideal. Application generators today typically produce only pieces of the application system, often ignoring one or more of the inputs listed above and the consequent outputs.

Claims have been made that application generators result in 10, 100, or even 1,000 to 1 productivity improvements over traditional system-development techniques. While these claims are largely unsubstantiated and are certainly not directly comparable, they are indicative of the goals of the developers of these system-development tools and of the order-of-magnitude changes in the system-development process that can be achieved, at least in certain well-defined cases. Application generators do indeed represent a generational step in the software evolution process.

The following bibliography indicates the range of recent publications on application generators and related topics. A separate section provides a representative listing of products advertised as members of the application generator family.

The Genealogy of Application Generators



BIBLIOGRAPHY

1. American Management Systems, Inc. "Generation Five: An Application Generator for Financial Systems." Arlington, Va. July 15, 1980.
2. Cardenas, A. F., and W. P. Grafton. "Challenges and Requirements for New Application Generators." *AFIPS, Proceedings of the 1982 National Computer Conference,* June 7–10, 1982, Houston, Texas.
3. Cardenas, A. F. "Technology for Automatic Generation of Application Programs—a Pragmatic View." *MIS Quarterly,* 1 (1977), 9, pp. 49–72.
4. Chamberlin, D. D., et al. "A History and Evaluation of System R." *Communications of the ACM,* 24 (1981), 10, pp. 632–646.
5. Cobb, R. H. "COBOL-80 Controversy: Exercise in Futility." *Computerworld,* 15 (1981), 12, p. 45.
6. *Computerworld Extra.* "Dawn of the Software Decade." 14 (1980), 38.
7. de Peyster, D. "Application Generator Runs on TI 990 Series." *Computer Business News,* 3 (1980), 6, p. 16.
8. de Peyster, D. "High-Level Languages Lauded as Software Tools." *Computer Business News,* 3 (1980), 26, p. 3.
9. *EDP Analyzer.* "Application System Design Aids." 19 (1981), 10.
10. *EDP Analyzer.* "Developing Systems by Prototyping." 19 (1981), 9, pp. 1–12.
11. *EDP Analyzer.* "Easing the Software Maintenance Burden." 19 (1981), 8.
12. *EDP Analyzer.* " 'Programming' by End Users." 19 (1981), 5.
13. Enos, J. C., and R. L. Van Tilburg. "Software Design." *Computer,* 14 (1981), 2, pp. 61–83.
14. Frank, R. A. "Let the Users Program." *Datamation,* 28 (1982), 1, p. 88.
15. Goodman, A. "Application Generators at IBM." *AFIPS, Proceedings of the 1982 National Computer Conference,* June 7–10, 1982, Houston, Texas.
16. Goodman, A. M. "IMSADF: A Tool for Programmer Productivity." *Telesystems Journal,* 7 (1980), 4, pp. 11–21.
17. Grochow, J. M. "Application Generators Anticipate Requirements." *Computerworld,* 15 (1981), 13, p. 30.
18. Grochow, J. M. "Financial Information Systems: The New Generation." *Proceedings of the Conference on Financial Information Systems: The New Generation,* October 5, 1981. Chicago, Illinois: National Institute for Management Research.

19. Hammer, M., W. G. Howe, V. J. Kruskal, and I. Wladawsky. "A Very High Level Programming Language for Data Processing Applications." *Communications of the ACM*, 20 (1977), 11, pp. 832–840.
20. Hancock, J. L. "Chipping Away at Productivity." *Computerworld,* 13 (1979), 32, pp. 7–18.
21. Holtz, D. H. "A Nonprocedural Language for On-Line Applications." *Datamation,* 25 (1979), 4, pp. 167–176.
22. Houghton, R. C. *National Bureau of Standards Software Tools Database.* Washington, D. C.: National Bureau of Standards, 1980.
23. *Information Systems News.* "The Productivity Dilemma." February 9, 1981, pp. 21–40.
24. Kaufman, S., and M. J. Elmore. "The Package Software Industry—An Investor Perspective." San Francisco, Ca.: Hambrecht & Quist, October 8, 1981.
25. Leavitt, D. "Software Generators are Keys to Efficiency." *Computer Business News,* 3 (1980), 7, pp. 1, 6.
26. Leverett, B. W., R. G. G. Cattell, et al. "An Overview of the Production-Quality Compiler—Compiler Project." *Computer,* 13 (1980), 8, pp. 38–49.
27. Loomis, J. E. "Reader Puts Spotlight on Applications Generators." *Computer Business News,* 4 (1981), 8, p. 10.
28. Loomis, J. E. "Using Generation 5 to Produce Product Labor Rate Sheets." AMS memorandum (available from author). August 1979.
29. Mandell, M. "Gloom and Glad Tidings for James Martin." *Computer Decisions,* 13 (1981), 1, pp. 10, 12.
30. Mishelevich, D. J., and D. Van Slyke. "Application Development System: The Software Architecture of the IBM Health Care Support/DL/1-Patient Care System." *IBM Systems Journal,* 19 (1980), 4, pp. 478–504.
31. Musgrave, B. "Relating to Data Bases." *Datamation. Special Edition, vol. 29* (1980), p. 30.
32. O'Connor, R. J. "Applications Generator to Target 32-Bit Systems." *Computer Business News,* 4 (1981), 8, p. 2.
33. Pettinger, J. "Are Programmers Obsolete?" *Interactive Computing,* 7 (1981), 2, pp. 2–11.
34. Rosen, E. C. "Vulnerabilities of Network Control Protocols: An Example." ACM SIGSOFT, *Software Engineering Notes,* 6 (1981), 1, pp. 6–8.
35. Roth, R. L. "Program Generators and Their Effect on Programmer Productivity." *AFIPS, Proceedings of the 1982 National Computer Conference,* June 7–10, 1982, Houston, Tx.
36. Runyan, L. "Applications Development Software Still a Sore Spot." *Datamation,* 27 (1981), 3, pp. 165–167.
37. Seaman, J. "Make or Buy . . . Software, Processing, Maintenance, and Facilities Management, Too." *Computer Decisions,* 12 (1980), 12, pp. 43–56.
38. Schofield, D., A. L. Hillman, and J. L. Rogers. "MM/1, A Man-Machine Interface." *Software Practice and Experience,* 10 (1980), 9, pp. 751–763.
39. Shoor, Rita. "'Mark V' Hikes Productivity 50%." *Computerworld,* 15 (1981), 39, pp. 1, 7.
40. *Small Systems World.* "Generators vs. Compilers." 7 (1979), 4, p. 23.
41. Snyders, J. "Generators Overcome Programmer Shortage." *Computer Decisions,* 13 (1981), 3, pp. 34–42.
42. Snyders, J. "Snyders on the Software Highlights of 1980 and What They Mean for You." *Computer Decisions,* 12 (1980), 12, pp. 74–87.
43. Sperling, T. B., and R. Egli. "The Future Application Generators in Corporate Systems." *Infosystems,* 25 (1979), 6, pp. 88–92.
44. Stiefel, M. L. "Surveying Data Base Management Systems." *Mini-Micro Systems,* 12 (1979), 11, pp. 94–104.
45. Stotland, V. G., "Shortcut Systems." *Datamation,* 27 (1981), 4, p. 163.
46. "Systems and Software." *Electronic Design.* Special Editorial Series. November 12, 1981.
47. Teichroew, D., and H. Sayani. "Automation of System Building." *Datamation,* 17 (1971), 8, pp. 25–30.
48. VanErp, S. D. "Software Flexibility." *ICP INTERFACE Insurance Industry,* 6 (Spring 1981), pp. 14–16.
49. Waldrop, J. H. "Application Generators: A Case Study." *AFIPS, Proceedings of the 1982 National Computer Conference,* June 7–10, 1982, Houston, Tx.
50. Weiss, H. M. "The ORACLE Data Base Management System." *Mini-Micro Systems,* 13 (1980), 8, pp. 111–114.
51. Wulf, W. A. "Trends in the Design and Implementation of Programming Languages." *Computer,* 13 (1980), 1, pp. 14–22.
52. Zloof, M. M. "QBE/OBE: A Language for Office and Business Automation." *Computer,* 14 (1981), 5, pp. 13–22.
53. Zornes, A. "Then My Computer Said to Me . . ." *ICP INTERFACE Data Processing Management,* 6 (Autumn 1981), pp. 27–29.

REPRESENTATIVE PRODUCT DESCRIPTIONS

1. Admins, Inc. *Admins/II Summary.* Cambridge, Ma.
2. American Computer Group, Inc. *The Proven DBMS Application Development System for DEC II & VAX.* Boston, Ma.
3. American Management Systems, Inc. *Generation Five.* Arlington, Va.
4. Artificial Intelligence Corporation. *Intellect User's Guide.* Newton Center, Ma.
5. Cincom Systems, Inc. *MANTIS.* Cincinnati, Oh.
6. Cullinane Corporation. *IDMS. The Data Dictionary Driven Data Management System.*
7. Distribution Management Systems. *ORACLE.* Bedford, Ma.
8. DJ 'AI' Systems Ltd. *The Last One.* Los Angeles, Ca.
9. Evaluation and Planning Systems. *FCS-EPS: The Management Decision Support System.* New York.
10. General Automation. *NoCode.* Anaheim, Ca.
11. Henco, Inc. *INFO.* Wellesley, Ma.
12. IBM. *Application Development Systems Catalog.* G320-6562. White Plains, N.Y., 1981.
13. IBM. "Application/Development Aids." *Information Processing* (January 1982), pp. 22–23.
14. IBM. *A Departmental Reporting System.* G320-6097. White Plains, N. Y., 1979.
15. IBM. *Development Management System /VS.* GH20-1863. White Plains, N. Y.
16. IBM. *Storage and Information Retrieval System/Virtual Storage. General Information Manual.* GH12-5114. White Plains, N. Y., 1981.
17. Information Builders, Inc. FOCUS. New York.
18. Information and Systems Research, Inc. *RIMS/MPG.* Coraopolis, Pa.
19. Mathematica Products Group. *RAMIS II.* Wilton, Ct.
20. National CSS. *NOMAD 2.* Wilton, Ct.
21. Prime Computer, Inc. *Prime/TAPS—Terminal Application Processing System.* Natick, Ma.
22. Science Management Corporation. *IDOL.* Product Description of Database Management System. Riverdale, Md.
23. Planning Research Corporation. *Central Software.* McLean, Virginia.

Software product quality assurance

by JOHN R. RYAN
Texas Instruments, Inc.
Austin, Texas

ABSTRACT

Providing clear objectives, guidelines, and requirements in an environment conducive to high productivity is absolutely essential to designing and producing high-quality software. The Software Quality Branch of the Computer Systems Division of Texas Instruments is tasked with providing support functions that are vital to producing high-quality software.

This paper explains the role of the Software Quality Branch in administering the development methodology of the Computer Systems Division. The paper also describes our participation in a corporate effort to define and monitor quality indices and our use of a software quality circle to encourage commitment to quality goals and to develop solutions to quality problems.

INTRODUCTION

For a computer system to be competitive in today's marketplace, the manufacturer must commit significant resources to its support. For the product to remain viable, timely maintenance releases must be made to correct functional deficiencies, to provide the additional functionality needed to maintain a leadership position, or simply to meet competition. When functional enhancements are made, it is essential that upward compatibility with earlier releases be maintained in order to protect the users' investment in application software. The product must be scrutinized to assure that manufacturability, configurability, and ease of installation have been accounted for in the packaging.

Good software engineering practices must be employed in the analysis, design, and implementation of the product for it to be testable, maintainable, and easy to use. It is economically essential that the development and maintenance of a viable computer system product be carefully controlled in order to meet customer requirements while minimizing development, maintenance, and support costs.

It is the responsibility of the Software Quality Branch to provide all levels of management with the information necessary to orchestrate the development and maintenance of a computer system product without neglecting any of the critical quality attributes,[1] including the following:

1. Correctness—The extent to which a program meets its specifications and fulfills the user's objectives.
2. Flexibility—The effort required to modify an operational program.
3. Interoperability—The effort required to couple one system to another.
4. Reliability—The extent to which a program can be expected to perform its intended function with the required precision.
5. Maintainability—The effort required to locate and fix an error in an operational program.
6. Usability—The effort required to learn, operate, input data and interpret output of a program.
7. Testability—The effort required to test a program to insure that it performs its intended function.

THE SOFTWARE QUALITY ORGANIZATION

Each section of the Software Quality Branch is responsible for insuring that one or more of the quality attributes are present in the product under development. The sections are as follows:

1. Software Quality Assurance
2. Software Audit
3. Software Engineering Support
4. Software Configuration Management

Software Quality Assurance

The Software Quality Assurance (SQA) Section verifies that the product meets all specifications (requirements, functional, and design). When authorization has been granted to develop a new product or enhance an existing one, staff begin immediately to develop an overall test plan.

The test plan describes both the strategy for testing the product and the hardware and software configurations in which it will be tested. A list of features added since the last version is complemented by a description of new tools and programs that will be developed to test them. All previously developed test tools and regression test software are described. A schedule is included, indicating when specifications, user documentation, unit test results, hardware, and software are to be delivered to SQA and when status reports will be made to the project manager.

An SQA representative attends the weekly project reviews to report on the status of the test effort. SQA verifies the technical content and clarity of the user documentation produced by the Technical Publications Branch.

Documentation and software defects are noted by submitting Software Trouble Reports (STRs). When SQA has received a new version and verified that the deficiency has been corrected, the STR is marked "fixed." Both the SQA and the project managers receive reports showing the status of all STRs against the product. Using these reports to track outstanding STRs reduces the chance that a problem will be accidentally overlooked.

When all tests have been passed satisfactorily and the product and documentation are deemed customer-ready, i.e., meet all specified requirements, they are turned over to the Software Audit Section for final verification.

Software Audit

The Software Audit (SWA) Section is responsible for verifying the installation and operation of the computer system product on actual production hardware, using *only* the documentation that will be shipped to the customer. A software auditor must simulate as closely as possible a "real" customer. He uses the system in a simulated production environment.

A minor defect that is encountered after SWA has begun final verification must be documented in the product Release Information document. If the problem is sufficiently serious that the product is no longer judged to be customer-ready, it is returned to the project and must undergo regression testing by the SQA Section before it can be resubmitted to SWA.

Some examples of serious problems that can cause a product to fail final verification are these:

1. It causes the operating system to crash or hang.
2. It causes data to be lost or destroyed.
3. It interferes with the operation of other programs.
4. It does not satisfy its specifications.

By insuring that all known problems with the final version of the product are either fixed or documented, the SWA Section protects the customer from unexpected difficulties when installing and using the product.

Software Engineering Support

The Software Engineering Support (SES) Section reviews specifications, evaluates software prototypes from a human-factor point of view, and coordinates the development and distribution of appropriate software tools that are used internally for development activities.

SES representatives attend all design reviews and comment on the consistency and compatibility of the proposed design with respect to other related products. In addition, they coordinate the proposed changes in development standards or methodology.

Because of the involvement of the SES Section, many compatibility and usability problems are identified and corrected early in the development cycle. Their efforts in promoting standards, methodology improvements, and the use of software tools also contribute to the efficiency of the development effort.

Software Configuration Management

The Software Configuration Management (SCM) Section is responsible for controlling source changes for products under development. They verify that product installation kits (the collection of programs that the customer purchases to install on a system) can be manufactured from the corresponding source code by using standard manufacturing procedures. SCM integrates new modules to be tested into the master library and produces each intermediate test version for the project and SQA.

One of the key elements of configuration management is the control of reports of software failures and requests for design changes.[2] For this reason the SCM Section manages the Software Trouble Report (STR) system. This system tracks functional deficiencies and requests for enhancements from customers, field analysts, and factory personnel.

THE DEVELOPMENT CYCLE

The Software Quality Branch participates actively in each phase of software development, assisting management in verifying that all milestones have been met and all quality requirements are being satisfied. The major phases of product development are as follows:

1. Initiation
2. Definition
3. Design
4. Programming
5. System test
6. Acceptance

Initiation Phase

During the initiation phase of product development, all market requirements are analyzed by the Product Planning Branch. The product planners work with systems analysts from one or more of the software development branches to produce the marketability requirements specification. At this time, the SQA Section becomes familiar with the system requirements in order to later evaluate the functional specification, prepare the product test plan, and develop any test tools or environments needed.

The SES Section uses the marketability specification to establish a user profile for the proposed product. This user profile, which assesses the background, capabilities, and preferences of the projected user, forms a basis for evaluating the applicability of the user interface and documentation.

Definition Phase

The definition phase of product development includes the analysis of functional requirements necessary to satisfy market demands. It culminates in the review and approval of the functional specification. SQA verifies that the proposed product meets all of the marketability requirements and develops a preliminary test plan. The SES group projects whether the system will be easy to use and whether it will be operationally consistent with related software products.

Design Phase

The SQA Section reviews the system design during this phase to insure that it is complete, consistent with the approved functional requirements, and compatible with related system products. SQA also completes the product test plan and puts it into final form. The test plan is reviewed and approved by the project manager. As draft copies of the user's guide and other manuals become available, they are reviewed by SQA for accuracy of technical content and conformity to system specifications.

Programming Phase

During the programming phase of product development, the Software Configuration Management (SCM) Section assists the development programmers in controlling source code changes; archiving copies of source, object, and listings as required; and maintaining the unit test library.

The software prototype of the user interface is evaluated by the SES Section for usability, user friendliness, and simplicity.

SQA participates in project code-reading sessions to see that the code is being developed according to standards and to help identify problems early. They evaluate the unit test results to determine when the product is ready to proceed to the system test phase.

System Test Phase

Most of the system test phase is executed in the factory by the SQA Section. An internal *alpha test* is conducted as soon as the product is sufficiently functionally complete and operationally stable to be used in a limited production environment. When the product has satisfactorally completed the alpha test, it is distributed to selected external customers for *beta test.* The beta test period is concurrent with the final weeks of system test.

Internal alpha test

As integration is completed on major subsystems, they are turned over to the SQA Section for system testing. Problems encountered are reported to the project manager so that the modules affected may be identified, corrected, and resubmitted to SCM. SCM integrates the changes into a new test version for SQA. The test history and current problem status are used by the project manager and the SQA manager to determine when the product is ready to proceed to beta test and finally to the acceptance phase.

External beta test

The SQA Section coordinates the planning and execution of the beta test. With the assistance of Product Marketing, customers are identified to use the new software for its application in a controlled, quasi-production environment. Each beta test site is contacted weekly for a report of confirmed or suspected bugs. This information is used to identify areas within the system where intensive testing may be needed.

The beta test arrangement gives the selected customer the advantage of advance information about the new product and a head start in developing applications to use the new functions. It also enables SQA to identify problems in environments that are extremely difficult to stage or even simulate in the factory. The beta test sites provide an important evaluation of the usability of the software and user documentation.

Acceptance Phase

The SWA Section performs the final verification for the product during the acceptance phase. SWA receives the product on the same media as will be shipped to customers, including the released documentation. It is installed and executed according to the instructions in the manuals and verified to be functionally complete prior to shipment.

THE SOFTWARE QUALITY CIRCLE

The Software Quality Circle has been established within the Computer Systems Division to provide representatives of each development group with a forum where they can express their concerns about quality and propose solutions to any perceived problems. This not only promotes cooperation but also gives the Software Quality Branch access to the combined experience, wisdom, and innovative creativity of the development branches through their representatives on the circle.

Typical Issues

One of the first issues raised before the circle was the critical dependence of all other development groups on the basic functions provided by the operating systems. In order to better coordinate incremental changes to the operating systems, an OS control board was established. Composed of representatives from the various development areas, the OS control board evaluates the potential effect of each proposed change on other products.

After synthesizing, prioritizing, and carefully studying a list of problems that were judged to undermine the quality effort, the circle focused on several key areas of development methodology that were not being consistently applied. Various subcommittees were formed to formulate standards and procedures. They operate under the guidance of the Software Quality Branch and periodically report on their progress to the Software Quality Circle.

Assessment of Results

The Software Quality Circle has made several important contributions to the overall software quality effort during its first year of operation. The major benefits have been promoting better understanding of the root causes of quality problems and promoting community commitment to the proposed solutions.

The members come to the monthly meeting with symptomatic observations; group discussion and analysis normally lead to the identification of the underlying causes. Finally, the Software Quality Circle members have been of enormous help in promoting awareness of quality issues in the development community.

Future Plans

It is our intention to continue using the circle as a source of both new ideas and feedback with respect to quality standards and procedures. If a problem is identified that will require management attention to resolve, appropriate action will be taken by the Software Quality Branch to send the problem to the level of management necessary to solve the problem. The circle will also play an important role in coordinating our training effort to maximize the quality awareness of the development community through the use of structured methods.

QUALITY MEASURES

The project managers are required to forecast the quality of all computer system products. The Software Quality Branch

manager must report on the actual results to the corporate quality manager. Any variance from the forecast must be explained and an action plan described to remedy the problem.

Leading, concurrent, and lagging quality indices have been established to indicate the measurable quality of the product during the programming phase, during the system test phase, and after the release of the product.

Leading Index

We have found that program complexity as measured by the Halstead Effort metric[3] correlates well to the number of times that a module must be reworked to correct errors. We are currently establishing complexity guidelines for our products. Any module that exceeds the guideline will be reviewed and, if possible, reworked or decomposed into multiple smaller, simpler modules.

Concurrent and Lagging Indices

The concurrent index of quality is based on the number of problems documented by STRs while the software product is in the system test phase. At the time development is begun, a maximum number of acceptable outstanding problems is established for the product. If the number of STRs ever exceeds the maximum, corrective action is mandatory.

The lagging index is established in an analogous manner, but it is computed by using STRs that are submitted after the product is released. If the number of STRs for a product exceeds the established maximum, a new version of the product will be released to correct the reported problems.

CONCLUSIONS

Throughout the development cycle, Software Quality Branch personnel review specifications, plan and conduct various tests, and verify completion of each of the development phases. The project manager is regularly provided with an objective assessment of the status of the product relative to approved specifications. When conflicts arise, they are sent to higher management.

One can conclude from the specification, coordination, assessment, and verification activities described that software quality is fundamentally a management problem. This fact sometimes becomes lost in the myriad of very real technical issues and business decisions that project and quality management are faced with. The fact that computer products must be suited to a variety of applications can provide further complications for the computer system vendor.

To deal in an objective way with the complexity of modern systems having diverse requirements, it is essential to agree formally on the system requirements and a quality plan that insures that they will be met. Objective quality measures are essential to the avoidance of conflicting assessments of the true state of project completion.

A software quality circle is an invaluable forum for gaining community acceptance and support for methodology changes.

Although the responsibility for product quality must rest squarely with the project manager, the successful execution of both development and quality plans depends on the level of cooperation that the project manager and the quality manager are able to achieve.

REFERENCES

1. McCall, James A. "An Introduction to Software Quality Metrics." In J. D. Cooper and M. J. Fisher (eds.), *Software Quality Management.* New York, Petrocelli, 1979.
2. Bersoff, Edward H., V. D. Henderson, and S. G. Siegel. "Software Configuration Management, An Investment in Product Integrity." Englewood Cliffs, New Jersey: Prentice-Hall, 1980.
3. Halstead, Maurice H. *Elements of Software Science.* New York: Elsevier North-Holland, 1977.

A quality assurance program for software maintenance

by JOHN W. CENTER
Medtronic Incorporated
Minneapolis, Minnesota

ABSTRACT

This paper presents the description of a quality assurance (QA) program applied to software maintenance projects. The QA program is a set of checks or inspections overlaid on the steps of the maintenance project. The relationship between the QA program and project management is shown. The paper includes a brief discussion of waivers and deviations to standards and control documents. The QA checks are delegated to three levels. The rationale, scope, and authority for each level are discussed. A list of sample criteria used for each QA check or inspection is given.

INTRODUCTION

Recently a great deal of interest has developed relative to quality assurance (QA) techniques and their application in the environment of management information systems (MIS). Problems associated with controlling software maintenance have also been recognized. The MIS department of a manufacturing company is usually responsible for applications with diverse characteristics. The applications cover the range of finance, planning, inventory control, personnel, process control, and automatic test equipment. With such differences in applications, the requirements and risks of maintenance are different. A means to control the maintenance process and reduce the risks would be of distinct benefit. If the MIS department has or is starting a QA function, a QA program applicable to software maintenance would be most useful.

Fischer[1] and Mendis[2] give excellent descriptions of general QA programs applied to software environments. Though the articles describe programs intended for new software development, they establish excellent background useful in understanding how a QA program works and what the objectives are.

Lientz and Swanson[3] have a new book on the topic of software maintenance. Roberts[4] presents the software maintenance problem in a very practical light. A set of plans is a major aspect of any QA program. Included are test plans, assurance plans, configuration control plans, etc. Buckley[5] and Dunn and Ullman[6] recently gave excellent presentations of QA plans for software projects.

The QA program presented here should be treated as an example. Readers who attempt to implement an identical one in their departments may find certain problems and inefficiencies. However, readers should be able to establish a software maintenance QA program by using the major features and concepts presented in this paper as guidelines.

THE PROGRAM

There are several components to the software maintenance QA program. The program is based on a set of checks or inspections. The placement or execution of inspections is determined when the software maintenance project is broken into steps. The checks or inspections are placed according to a set of rules used by traditional quality assurance management. The rules have evolved over time. The checks are further classified by type, in-line or off-line, and by three levels. There is often difficulty in applying current standards to older software being maintained. Waivers and deviations become an important component of the QA program as mechanisms to document difficulties in the universal application of standards.

Each of the following sections discusses the various components of the QA program in more detail. The description of the rationale, authority, etc., of each level of QA check is discussed. A list of the criteria used for each inspection or QA check is also included.

Project steps

There are dozens of ways to break a software maintenance project into separate steps. One might treat maintenance of sufficient size or magnitude the same as traditional new system development. It would then make sense to use steps given in one of the current books or tutorials on system development methodologies. The QA program for very large maintenance projects would be the same as the QA program used for new development.

One could also consider the maintenance of a single program. Perhaps the number of lines on a report page is being changed. A project of this size is so small, and would be completed so quickly, that the benefit of a formal QA program is questionable. Projects in the emergency maintenance class could also be too small to require the formal QA program suggested in this paper. One must remember that emergency maintenance relates to the occurrence of a failure condition. There should be a separate formal QA program for failure investigation and corrective action.

Thus, the QA program discussed in this paper is oriented toward the middle-size maintenance project. The vast majority of maintenance resources are expended on projects in the middle group. Books and articles have been published recently on the topic of software maintenance. However, there appear to be few articles directly related to breaking the maintenance project into separate steps. There is one interesting potential source of information on this aspect of software maintenance projects. A major computer hardware conversion has strong similarity to the maintenance process. A publication provided by Sperry Univac[7] to assist in the conversion process gives an excellent breakdown of steps in a conversion project. Some of the quantitative results of using a QA program during a conversion were presented and published at the 1981 National Computer Conference.[8] The QA program for software maintenance is almost the same as one used during that conversion.

The box labeled "Establish Need for Maintenance" in Figure 1 represents the first identification of a potential or real software maintenance project. This need might be determined by the user department when a new law is passed, a new format for ZIP codes is used, etc. The need might be determined by the MIS department when it becomes apparent that much higher processing efficiencies might be obtained, a constraint on the physical hardware has been reached, etc. Usually a formal request or form is generated. The request is

processed through the MIS channels to place the maintenance project in the queue.

The box labeled "Schedule Maintenance" is the first task of the newly assigned project leader. This task includes the when, who, and how of the maintenance project. The project leader determines which subsystems are involved, which people should do the specific tasks, whether there are any automated tools available, etc. It is assumed that the project is broken down primarily by information subsystem. The subsequent steps for each subsystem are concurrent. The project leader establishes the detailed schedule of subsystems on the basis of contingencies and requirements. The steps for a single subsystem are shown as a representation of many that might be in parallel.

The box labeled "Maintenance Preparation" represents the tasks performed prior to the actual maintenance work. This step includes the following tasks: (1) establishing the documentation package, detailed requirements documents, status checklists, forms, etc.; (2) gathering test data, sample files, or data specifically designed to test the (sub)system; (3) gathering file information or file descriptions, record layouts, and look-up tables for both the old and new versions; and (4) gathering program listings, cross-reference data, special or library routine descriptions, etc.

The box labeled "Modify Code" is the activity on which the entire maintenance project depends. Closely associated with the code modification is compilation. The box that directly follows code modification is error free or "Clean compile." It is usually assumed there were no program coding errors prior to maintenance. A significant portion of software maintenance relates to data or file descriptions. When changes are made in the data descriptions, the data and files must be modified to match the new ones. The box labeled "Convert Data/File" represents this process.

After making the appropriate modifications to the program code and the data representations, the "new" programs have to be tested or checked out. In order to isolate problems quickly, each program should be tested separately. The process of testing the programs or routines is represented by the box labeled "Program Test and Debug."

After all the programs have worked successfully, the subsystem is put back together and checked out. The box labeled "Subsystem Test and Debug" represents this process. Test failures mean looping back to modify code and convert data/files.

The "new" subsystem now has to be integrated into the system. The processing related to the integration is represented by the box "Preparation for System Integration." This preparation includes a large number of administrative tasks. The following is a sample: (1) Generate new control language explanations or higher-level flow charts. (2) Generate new program documentation updates, new program listings, cross-references, etc. (3) Generate new data/file documentation updates, new data element, record or file descriptions. (4) Generate transactions to update the data catalog or configuration management mechanism. (5) Generate updates for operations documentation.

The new system is now ready to be put into operation. This process is represented by the box labeled "System Integration." The entire information system is tested; this step is represented by the box labeled "System Test and Debug." Problems and errors may be discovered during this process. This means that the maintenance for the subsystems with problems will have to loop back to modify code and convert data. With tight project control, the maintenance for the problem subsystems may have to start over completely.

The remaining process is represented by the box labeled "Documentation Update." The preparations done at subsystem level should make it easy to actually perform this task. The final task is the box labeled "System Acceptance." The user now gets involved in using the newly maintained system. The user will perform the acceptance test procedures that were generated early in the maintenance project.

The steps shown in Figure 1 are representative of the tasks performed during software maintenance. The remainder of this paper assumes that this set of steps is used to actually perform the maintenance and control the project. The QA program and associated QA checks grow around on the skeleton of these steps, or the project management methodology, used for software maintenance projects.

Placement of QA checks

One of the more critical aspects of developing a QA program is the placement of the QA checks or inspections. Experience from the long history of manufacturing QA gives some of the basic ground rules. Juran and Gryna[9] give an excellent list of potential locations for manufacturing QA inspections. The following is a paraphrased version of that list, emphasizing software terms or analogies:

1. At receipt of software from vendor, called incoming inspection or vendor inspection.
2. Following the setup of the production process, setup approval, to provide added assurance against producing defective software. Sometimes the setup check is used to give prior acceptance of the software that goes through the subsequent process.
3. During the execution of critical or costly operations, usually called process inspection.
4. Prior to delivery of software from one processing group to another, called toll-gate inspection.
5. Prior to shipping completed software to the customer/user, called finished-goods inspection for hardware.
6. Before performing a costly irreversible operation.
7. At natural peepholes in the project flow.

The QA checks are represented by triangles on Figure 1. The names for the numbered triangles are (1) preparation, (2) compilation, (3) data/file conversion, (4) subsystem test and debug, (5) preparation for system integration, (6) documentation update, and (7) system acceptance.

The triangles were placed on the project step flow on the basis of the list of potential locations given above. The preparation check is called a set-up approval. The checks done for compilation and data conversion are called process inspections. In some cases an additional set-up approval QA check should be established prior to data conversion, especially when the conversion itself is irreversible. The check

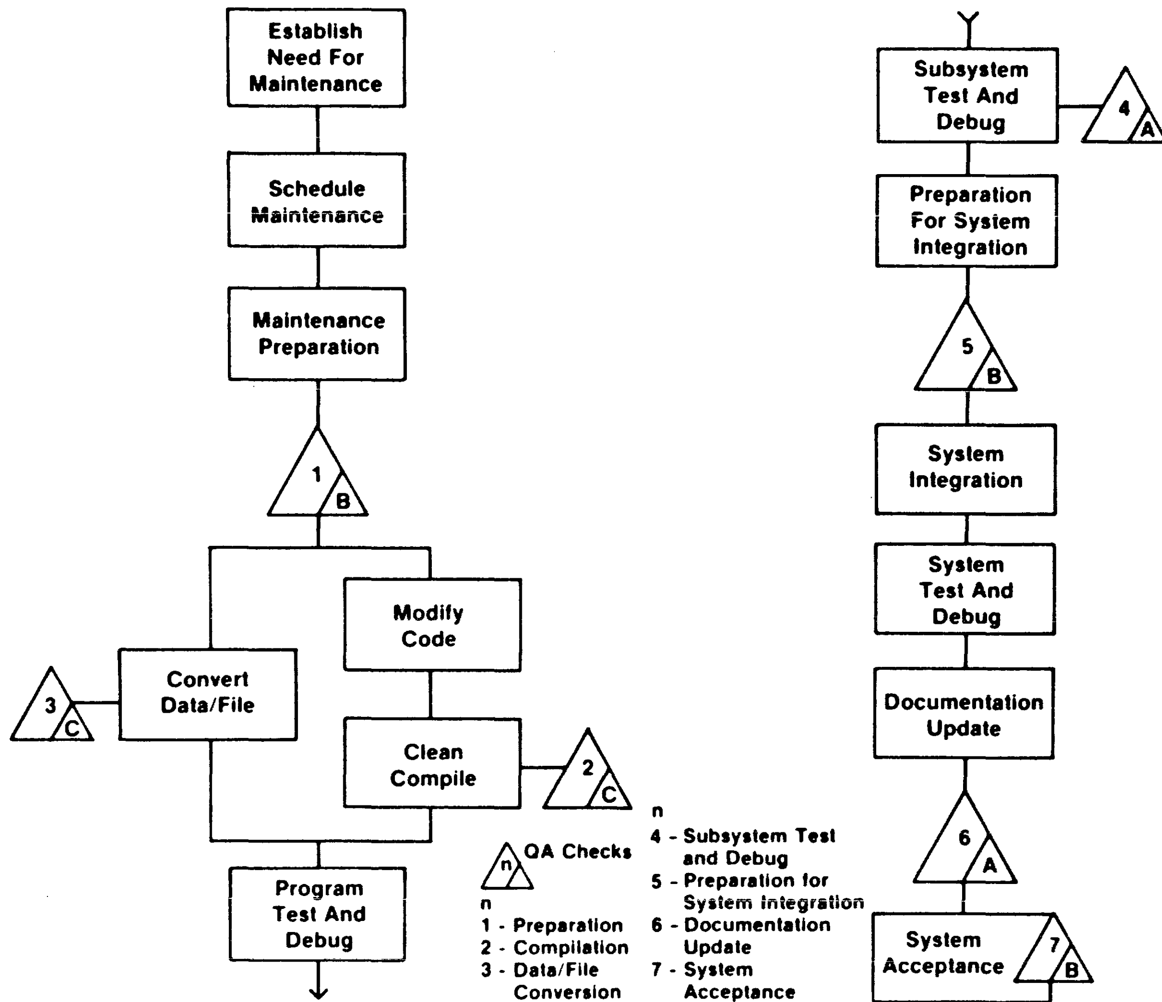


Figure 1—Software maintenance project steps overlaid with QA checks

done for System Test and Debug is a toll-gate inspection. Additional formal checks could be done for Program Test and Debug as process inspections. The check done at Preparation for System Integration is a toll-gate inspection. Documentation Update is a natural peephole. The check called System Acceptance is a finished-goods inspection.

The QA checks were placed assuming a normal project, based on the scales of size, level of criticality, magnitude of costs, etc. Additional checks are certainly not precluded. When any additional checks are desired, consider the list of potential locations. Extreme care must be given regarding the removal or consolidation of the seven locations listed above.

Project management milestones

Placing QA checks in the stated locations gives an additional project management benefit. QA checks are associated with each of the major tasks of the maintenance project. Passing a QA check provides an absolute milestone. The project leader can truthfully say that the documentation has been updated only when the associated QA check has been completed. Additional QA checks provide the project leader with more detailed project milestones when needed.

The relationship between project management and QA checks has been recognized before. The NBS Special Publication by Fife[10] gives a brief but excellent presentation of this relationship.

In-line QA checks

Some of the project steps have distinct boundaries, are very critical, have a serial relationship, etc. The QA checks associated with these distinct steps are placed in the line of the project flow. These checks are known as in-line checks. They may also be known as gates. These checks are called mandatory hold points in the draft by the Canadian Standards Association[11] on software quality assurance programs.

When the project encounters an in-line QA check, progress is technically stopped. No further work can be done on the maintenance project until the inspection or check is considered accepted. Generally, these are the steps where further progress with undetected errors or problem conditions would

be costly in terms of time, staff, etc. Corrections can be made when it is still cheap. Since these are placed in the more critical positions, they make excellent project milestones.

Off-line QA checks

Some of the steps of the software maintenance project are of a detail, recursive, or parallel nature. The QA checks associated with these steps are often a quick inspection or observation for a particular condition. One example involves errors detected during compilation. The code for other routines can be modified or compiled while the cause for a detected compile error is being researched. The QA checks for steps with these characteristics can be done off the line of project flow. This class is therefore known as off-line checks. They might also be called monitoring checks.

The off-line checks usually relate to conditions that can be quickly repaired or changed. The off-line checks are good for determining the effectiveness of the task being executed. The successful completion of the off-line checks is a good means to determine the project or task status, schedule impact, etc. They are good milestones internal to the major project steps.

Waivers and deviations to standards

Most MIS organizations have standards, test plans, procedures, and other control documents. QA programs require adherence to the appropriate control documents. However, maintenance takes place on systems that were probably developed under few or no control documents, under a different set of control documents, or in a different environment. Standards and procedures now used may not be applicable. The old standards may even conflict with those now used for development. Maintenance projects should be treated in ways similar to those employed for new development. The system should be brought up to current standards. The desire for bringing the systems into adherence is not practical in some cases.

Waivers are conditions where a standard, plan, procedure, etc., is completely dropped. Deviations are cases where temporary or very isolated changes are made relative to control documents. With documented waivers and deviations, the test plans, standards, etc., can be referenced by exception. It would be assumed that the normal or standard procedures were executed unless a waiver or deviation was requested.

There is risk associated with each deviation. Documentation of deviations is necessary to properly assess the risk, determine need for document changes, and make later maintenance more effective. The method of quantifying software quality discussed by Mendis[12] is based on careful documentation of deviation and failure conditions. Only the QA function can approve the maintenance waivers and deviations.

LEVEL OF QA CHECKS

In Figure 1 there are seven triangles symbolic of QA checks or inspections. In the lower right corner of each triangle is a letter, A, B, or C. These letters designate the level of QA check.

There are several spectra to which the term *level* refers. These include detail, impact and scope, experience of personnel, degree of personnel responsibility, depth of personnel knowledge, breadth of personnel knowledge, and significance of results.

The QA function has the ultimate accountability for all QA checks. The actual execution of the check and the associated responsibility can be delegated to other people or organizations. The delegation is an attempt to place the responsibility for detecting and correcting problems, deficiencies, and errors at the functional level closest to the source and most directly affected by the discovered problems. The use of levels allows the QA checks to be performed more cost-effectively.

Level C

The Level C checks are the lowest level of the QA checks. This level of check is delegated by the QA function to the people or organizations actually performing the maintenance. These checks are usually performed by the people with the least general experience, the least management responsibility for the maintenance project, and the narrowest direct scope of responsibility. The checks at this level are inspections of the tasks or functions that have the most constrained impact on the project. Referring to Figure 1, all the Level C checks are off-line or monitoring steps.

The Level C check could be done by the programmer, by a supervisor or senior programmer, or by a QA inspector assigned to and closely associated with the task. The NBS Special Publication by Brandstad[13] is a good tutorial on checks at this level. The Level C checks are normally performed in or near the same physical area where the task is executed.

The programmers physically performing the maintenance should not check or inspect their own work. Independence can be obtained by having the programmers rotate or exchange work. On a periodic basis, the pattern of exchange needs to be changed, because the effectiveness of the "independent" check may deteriorate if it is not changed.

Level B

The QA checks performed at Level B are delegated by the QA function to the project leader. The results of the checks done at this level usually have significant impact on the schedules and budgets of the task or project. There are three Level B checks in Figure 1. Two of the Level B checks are gates or in-line checks. The other is a monitor or off-line check.

The QA checks done at Level B are usually not performed in the same physical area where the work or task is done. This type of check would be done in a project work area or in the office of the project leader. The Level B check could be a consolidation of checks done at Level C. It might be a check of consolidated work or a consolidation task itself. The Level B check is a project milestone in most cases.

Level A

The QA checks at Level A are kept within the QA function. The checks done at this level are the most crucial, have the widest scope, and have the biggest impact on overall schedules and budgets. The checks done at Level A are also very informative in a project management sense. They are used to determine the adequacy and completeness of the checks done at Level C and Level B.

The Level A checks are similar to those done at Level B. A great deal of the effort of the Level A check is consolidation of previous checks. However, the final system acceptance evaluation is an extensive system execution. The system test would probably be executed against a standard or special set of test data.

The QA check done at Level A would seldom, if ever, be performed in the same physical area where the maintenance work was done. It would probably be performed in a work area dedicated to the QA function or in the office of the assigned QA personnel. Separation is used to maintain the independence and integrity of the Level A inspection or evaluation.

QA CRITERIA

The primary purpose of QA checks is to determine whether maintenance was performed properly. A secondary purpose is to determine the magnitude and associated risk of any deviation. The determination is made against a set of objective standards. Each step of the maintenance project process is different and has different goals. The various QA checks need different criteria oriented toward the object of the task being inspected.

The following sections give a representative set of criteria for each of the QA checks identified in Figure 1. When the QA program for software maintenance is implemented, those responsible for QA, system maintenance, etc., will have to determine whether the proposed set of criteria is correct for their department, systems, and people.

Preparation inspection criteria

These criteria apply to the QA check shown as Triangle 1 on Figure 1. This check is in-line or a gate, and is at Level B. The objective is to make sure that information needed in the later steps of the project is available. It is expected that the maintenance project leader will take a few extra minutes to check and double-check the status of the required documentation. Any oversights or omissions could cause significant delays during the execution of the maintenance.

The following criteria are the bare minimum. The project leader is given the authority to make additional checks and add criteria based on the case, people, system, and situation. The criteria are as follows: (1) Complete, or sample, data files must be present or storage location must be referenced. When sample or test data are to be used, the source and general content of the data file are to be documented. (2) File and record descriptions are to be present, referenced, or otherwise accounted for. (3) Program descriptions, lists, specifications, and other appropriate information are to be present, referenced, or otherwise accounted for. (4) Status, background, and detail information necessary to execute the maintenance properly and to insure that it was done, must be available. (5) Test plans, system acceptance criteria, etc., must be present or available.

Compilation inspection criteria

A check is performed to insure that the changes made as part of the maintenance project did not contaminate the original source program. This check is shown as Triangle 2 on Figure 1. The QA check, Level C, is performed by the programming personnel in an off-line or monitoring mode.

If no error messages are generated by the compiler, the program or routine is considered to have passed this Clean Compile check. Error messages are to be cleared or disposed on the basis of criteria given below. The classification of error messages is based on the ANS COBOL compiler used on the Univac 1100 series of computers.[14]

1. Leveling: Violations of ANS or Federal Standard COBOL.[15,16] This error message is conditionally acceptable. A request for deviation is needed. The request should discuss the reasons for using the extension. Include plans or suggestions to remove the condition where applicable.
2. Remarks: Actions taken by the compiler, but not necessarily an error in the source program. This class of error message is generally acceptable. Where possible, and reasonable, the condition should be removed for the sake of clarity.
3. Minor: The compiler generates code based on assumptions. Attempt to remove all error messages in the minor class. When it is considered necessary and reasonable to keep the condition, a request for deviation is to be filed. The request should include a plan to remove the condition where appropriate.
4. Serious: The compiler is unable to make reasonable assumptions, and no code or incomplete code is generated for statement. Errors in the serious class are not acceptable. Corrections will be made until all errors of this class are removed. When the serious error cannot be removed after reasonable effort, a compiler specialist will investigate the problem and provide a solution. If the specialist feels the risk is warranted, a deviation can be issued until solution is found.
5. Fatal: No code for the program or routine is generated. Errors in the fatal class are not acceptable. Corrections will be made until all errors of this class are removed. When the fatal error cannot be removed after a reasonable effort, a system or technical specialist will investigate the problem. The subsystem, and possibly the project, will not be allowed to proceed until a solution is found.
6. Compiler Errors: Internal conditions that generate errors during the compilation. This class is obviously not

acceptable. The compiler specialist will work with the experts at the vendor to remove the condition.

Data conversion criteria

The QA check associated with the data conversion step is shown as Triangle 3 on Figure 1. This is a Level C check performed in an off-line mode.

Typically, the data are converted by using a system utility, a local utility, or a special program. The utility routines are executed by using directives that give parameters, selected options, etc. The following criteria associated with the use of utilities and directives must be met: (1) There must be no syntax or format errors in the directives. (2) There must be no obvious execution errors associated with the directives.

In most cases, a test is made for input and output record counts and control totals within both utilities and special routines. The following criteria apply to the record counts and control totals: (1) There are to be no unexpected mismatches in the record counts or control totals. (2) There are to be no unexpected changes in the data size parameters like record size, blocking factor, and file size.

When the specified criteria are not met, an investigation is to be made. Disagreements involving the acceptance or rejection of a data conversion QA check will be resolved by the project leader. It is assumed the project leader will consider technical advice from a data or system specialist.

Subsystem Test and Debug Criteria

The QA check for Subsystem Test and Debug is indicated by Triangle 4 on Figure 1. This check is a Level A check done in an off-line mode. Though the QA function does not actually perform the tests, the status of success or failure is of prime interest and concern. The QA function is to see that all the tests required by the appropriate test plan are carried out. The presence and adequacy of the test plan was one of the criteria points in the QA check for preparation. The NBS Special Publication by Adrion[17] is a brief tutorial on testing procedures. Smith[18,19,20] is an author of some other books and articles published recently on the topic of testing and validation techniques. The QA function acts as an overseer and expeditor. Any disagreements that arise about the acceptance or rejection of tests will be settled by the QA function.

The test and debug process for the subsystem is executed after all the routines and programs have been successfully tested. There is no formal QA check during the debug phase of programs and routines. It is assumed that all the programs and routines have passed their tests, so any problems found at the subsystem level are considered very serious. If the fault cannot be quickly cleared and resolved, one of the previous steps will become the point for starting maintenance over.

One of the techniques often used to determine the success of a subsystem test is a file and report comparison. This approach is very effective when maintenance does not result in major changes in the data file structures or report contents. It is often easy to use a system utility or a local utility to do the file comparison. The following criteria are to be met when comparisons are used to evaluate success: (1) Files must have no unexpected mismatches, and (2) reports must have no unexpected mismatches. Expected differences can be cleared by documenting them with a request for deviation. Unexpected differences that cannot be easily removed are to be documented for further investigation.

Preparation for system integration criteria

The QA check for this step is Triangle 5 on Figure 1. It is a Level B check performed in a gate or in-line mode. The objective of this QA check is to make sure that all the documentation required for system integration is present. A second objective is to insure that all previous QA checks have been performed to a satisfactory degree.

The check is to make sure that the system can be reintegrated from the maintained subsystem. The check is also used to make sure that the new documentation can be easily generated. The project leader is expected to take sufficient time to check, double-check, and even triple-check the status of documentation. Any omission or oversight would cause major setbacks. The project leader should take enough time to insure that there will be no major problems found at system test.

The project leader is given the authority to make additional checks and add criteria based on case, people, system, and situation. The following criteria are the bare minimum: (1) Flow charts are to have adequate explanations. (2) Updates to the file and records descriptions must be present or referenced. (3) Updates to program and routine descriptions, lists, and other appropriate information must be present or referenced. (4) Updates for generating operations documentation must be present or referenced. (5) All deviations and waivers are to be approved. (6) All documented problems are to be cleared.

Documentation update criteria

This QA check is the last step performed prior to final operational testing. The check appears as Triangle 6 on Figure 1. It is a Level A check performed as an in-line gate.

The QA check is to insure the presence of documents that will define the system at a later time. The following criteria are to be used for the check: (1) All internal MIS documents for the system must have been replaced or updated. (2) All entries in the data catalog, or configuration management mechanism, must have been updated. (3) User documents must have been updated, retrieved and replaced, etc. (4) The system history must have been brought up to date.

System acceptance criteria

The QA check for System Acceptance is shown as Triangle 7 on Figure 1. The system acceptance is the last step in the maintenance project. This check is at Level B and is really integrated into the acceptance procedure itself.

The users of the system play a major role in system acceptance. The criteria for the acceptance of a system will vary with

the features of that system. One of the criteria for the preparation QA check is the presence of system acceptance criteria. The responsibility for controlling the execution of the acceptance procedure is left to the project leader. With well-defined and adequate QA checks executed at each of the major steps of the maintenance project, the system acceptance check becomes a simple procedure.

CONCLUSIONS

This paper has presented the structure and some details of a QA program for software maintenance. The program is still in its infancy. A QA program almost identical to the one presented in this paper was used when our company converted its systems from the computer of one hardware vendor to another. Once the personnel have become accustomed to the QA program, they should find it beneficial.

The QA program discussed in this paper can serve as a model or prototype for other organizations. Those who intend to implement a similar QA program should study the structure or features of the program rather than the details. The most significant features of the QA program are the use of three levels of QA checks, two types of checks, and the waiver or deviation. These features make the program work, make it effective, and make it acceptable to the personnel and management.

REFERENCES

1. Fischer, K. F. "A Program for Software Quality Assurance." *ASQC Technical Conference Transactions,* 32 (1978), pp. 333–340.
2. Mendis, K. S. "A Software Quality Assurance Program for the 80s." *ASQC Technical Conference Transactions,* 34 (1980), pp. 379–388.
3. Lientz, B. P., and E. Swanson. *Software Maintenance Management.* Reading, Massachusetts: Addison-Wesley, 1980.
4. Roberts, T. J. "Maintaining Quality after the Software is Released." *ASQC Technical Conference Transactions,* 31 (1977), pp. 157–166.
5. Buckley, F. "A Standard for Software Quality Assurance Plans." *Computer,* August 1979, pp. 43–50.
6. Dunn, R. H., and R. S. Ullman. "A Workable Software Quality and Reliability Plan." *Proceedings of 1978 Annual Reliability and Maintainability Symposium,* Los Angeles, Ca, Jan. 17–19, IEEE, pp. 210–217.
7. *Conversion Standards and Procedures.* Document FS9009. St. Paul, Minnesota: Sperry Univac Customer Marketing Services, 1979.
8. Center, J. W. "Quantitative Measures of MIS Quality Assurance during Hardware Conversion." *AFIPS, Proceedings of National Computer Conference,* 1981 (Vol. 50), pp. 323–327.
9. Juran, J. M., and F. M. Gryna, Jr. *Quality Planning and Analysis* (2nd ed.) New York: McGraw-Hill, 1980, p. 362.
10. Fife, D. W. *Computer Software Management: a Primer for Project Management and Quality Control.* NBS Special Publication 500–11, Washington, DC: US Government Printing Office, 1977.
11. *Software Quality Assurance Plans*. Document CSA Q396 (Draft). Rexdale, Ontario: Canadian Standards Association, April 1981.
12. Mendis, K. S. "Quantifying Software Quality—A Practical Approach." *ASQC Quality Congress Transactions.* 35 (1981), pp. 11–18.
13. Branstad, M. A., J. C. Cherniavsky, and W. R. Adrion. *Validation, Verification, and Testing for the Individual Programmer.* NBS Special Publication 500–56. Washington, DC: U.S. Government Printing Office, 1977.
14. *American National Standard COBOL (ASCII) Programmer Reference Manual.* Document UP8582. Blue Bell, Pennsylvania: Sperry Univac Corp., 1978.
15. *American Standard COBOL.* ANSI X3.23–1974. New York: American National Standards Institute, 1974.
16. *Federal Standard COBOL.* FIPS PUB 21–1. Washington, D.C.: U.S. Government Printing Office, 1975.
17. Adrion, W. R., M. A. Branstad, and J. C. Cherniavsky. *Validation, Verification, and Testing of Computer Software.* NBS Special Publication 500–75. Washington, D.C.: U.S. Government Printing Office, 1981.
18. Merilatt, R. L., M. K. Smith, and L. L. Tripp. *Computer Software Validation and Verification: A General Guideline.* BCS-40342. Seattle, Washington: Boeing Computer Services Co., June 1981.
19. Smith, M. K., and A. R. Bennett, *A Guideline for Business-oriented Software: Validation and Verification.* Document BCS-40343. Seattle, Washington: Boeing Computer Services Co., June 1981.
20. Smith, M. K., L. L. Tripp, L. J. Osterweil, R. N. Taylor, and W. E. Howden. "An Approach to Transfer Verification and Validation Technology." *AFIPS, Proceedings of National Computer Conference,* 1981 (Vol. 50), pp. 367–373.

The independent role: verification and validation, and compliance testing

by BARBARA J. TAUTE
*Time, Inc.**
New York, New York

ABSTRACT

The independent role of quality assurance can be most beneficial in the final testing phase of the software development life cycle. At this point various efforts and groups merge, often for the first time. The period of time devoted to the discovery, understanding, prioritization, and correction of software problems is critical to the timely delivery of the product. Quality assurance can provide and foster a healthy working environment for this interaction, as well as serve as an intermediary during points of disagreement.

This paper discusses an actual compliance testing effort of customized, complex vendor-supplied software. The value of having an independent group involved during the testing effort was clearly made visible during the arbitration process. The critical timing was monitored, points of disagreement over the contract were resolved, and the testing effort was shortened through the involvement of an independent group. In effect, both vendor and user/purchaser benefited from the quality assurance techniques brought by an independent role involvement.

*This paper was written while the author was employed by Peat, Marwick, Mitchell & Co., New York, New York.

INTRODUCTION

A close interaction between the developer and user is required to guarantee the delivery of data processing software which complies with a given set of requirements. This interaction has classically been a difficult process because of both groups' differing points of view and varying interpretations of the software's requirements. The final phases of the development process, particularly that of testing, can become strained due to the addition of time criticality for system delivery and the stress of interaction between these two groups. This intense testing interaction can be even further complicated in a vendor-contracted software environment in which there is no common ground to resolve the problems. An independent Quality Assurance (QA) Group, whether internal or external to the company, can greatly ease this transition period by laying early plans for the compliance test process, monitoring or assisting in the testing, and acting as arbitrator if difficulties arise.

This paper discusses the specifics of one such testing effort of customized, complex vendor-supplied software and the benefits of an independent QA Group during the verification and validation and compliance (acceptance) testing phases. The independent QA Group was a different company than the vendor or the intended user. The intended user was the purchaser of the package and as such was concerned with its timely test and installation in their environment. Specifics of the environment, perceived and actual QA needs, techniques of implementation, and effects of the independent QA approach are described.

PERCEIVED NEED FOR QA

The intended users of the software package felt they had a need for and initiated the request for the involvement of an independent QA Group because

1. They were unsure of the compliance test process
2. The timeliness of software delivery was critical
3. Communication with the vendor was becoming difficult
4. Staff numbers were insufficient
5. Forms for the new process were not developed

The user/purchaser felt it was important to select a QA Group that was proficient in the independent testing methodology and would be able to assist during the arbitration process, since they lacked this discipline and knowledge in their own shop. From the user's past experiences with this vendor, they had learned that many areas which appeared to be clearly defined in the specifications were confused or misinterpreted by vendor and user alike. During this particular installation effort, the user hoped to avoid these costly misunderstandings by employing a group which would interpret and mediate during the discussion periods. A high level of independence was important to the users because they desired to involve a QA Group which would neither offend the vendor development group nor impede progress.

Whether or not this perceived need justified the involvement of an independent QA Group depended upon the specifics of the environment.

USER ENVIRONMENT DEFINITION

The contracted system was comprised of minicomputer hardware, associated systems software, hardware and software maintenance agreements, and the application software package. The package was intended to be turnkey software, but more than 30% of the original code had been changed at the user's request. Further, additional new modules were specifically created by the vendor for the user and added to the system; thus the system approached custom-developed software. The total cost of the system (hardware and software modifications) to date of involvement was $850,000, and the development and initial data conversion effort had thus far spanned a period of two years.

The vendor was implementing the system in a phased approach as shown in Figure 1.

The independent QA Group was requested to give assistance primarily during the major implementation phase, especially regarding specification review and testing assistance.

The application package was industry-specific and was to provide an online automatic method of merchandise control. The processes of merchandise ordering, credit verification, delivery, billing, inventory, returns, terminations, accounts payable, and accounts receivable comprised the various subprograms of the package. Direct general ledger entries were

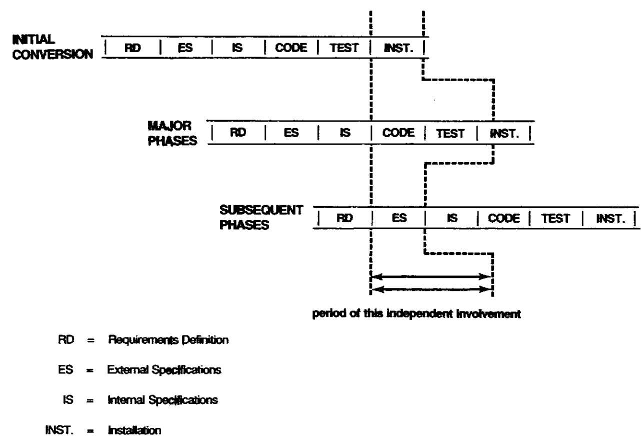


Figure 1—Period of involvement

to be implemented at a later date. This system (hardware and software) was intended to replace an existent batch minicomputer system; therefore, established databases, expectations of data processing output, and forms of input to the system were already present.

These conditions indicate a fairly complex system. However, the need and level of involvement of an independent QA Group is directed by implementation factors.

ACTUAL NEED FOR QA

The level of involvement of an independent QA Group is dependent upon the program complexity, the user, and the developer. In this type of environment, the involvement of an independent QA Group is essential for the following reasons:

1. Program is extremely large ($850,000 in 2 years)
2. Internal QA group does not exist
3. Software is developed by vendor
4. Time is extremely critical
5. User lacks data processing compliance testing expertise

Any development effort of this size or implementation time is necessarily complex and should have rigorous control mechanisms. A QA Group can assist the effort by early involvement in helping to define deliverables, monitoring the process, and providing testing assistance or acceptance sign off. Ideally, even in internal development, the QA Group should be independent in order to ensure that equally high standards of quality apply to all development groups. For externally developed, vendor-contracted software, the involvement of an independent QA Group is essential. The independent group can help resolve problems and points of disagreement in either party. This can help avoid litigation proceedings. The independent QA Group can help ensure the timely delivery of the software by evaluating the delivery schedule and monitoring the process. Finally, if the user lacks data processing expertise, the independent QA Group can help educate and train the users to enable a smooth installation. Also, an independent QA Group can help improve the developer's testing process through the installation of procedures that more thoroughly control the test process and the software release process. This yields more information, more control, and fewer chances for error in the installation process.

The involvement of an independent QA Group will yield certain deliverables, or tangible evidence of the QA process.

QA PRODUCT DELIVERABLES

Quality assurance must use a disciplined approach and as such it is important that documentation be maintained to reflect its involvement. The involvement of the independent QA Group, specifically during this testing effort, yielded various deliverables; among them were the following:

1. Project "pert" chart
2. Project test plan
3. Problem report forms

Project "Pert" Chart

A scheduled daily chart based upon key items and integration points was established for the testing time involvement. This allowed the progress of the time critical project to be measured and reported to management. Responsibilities, dates, dependencies, and a brief description of the task were included in the chart.

Project Test Plan

A test plan which described in detail the responsibilities and activities of all three parties (user, vendor and independent QA Group) was constructed. In the development of the test plan, the absence of certain testing efforts was discovered, and these were subsequently assigned to individuals.

Problem Report Forms

Problem report forms and summary logs were installed. The problem reports permitted a priority, description, dates, version numbers, and error category to be specified and formally recorded for every "query" about the system. This helped ensure that trivial problems were not lost and provided an identification element for all items.

These three deliverables allowed the QA Group to monitor the development effort.

QA TECHNIQUES DEFINED

The deliverables were used to record results of the activities and techniques of the QA Group. The specific methodologies that an independent QA Group can bring to the final testing process are as follows:

1. Progress measurement
2. Test development
3. Problem report control
4. Configuration management

All of these techniques are key to the successful completion of the project and critical during the final testing phase.

Progress Measurement

The ability to measure the test/development effort can provide invaluable information about the project and its expected delivery date. If no control interaction chart is in effect (or if it is no longer being used) QA should formalize this process. For this specific effort, a daily manual "pert" chart was drawn, showing responsibilities and activities of all participants.

Test Development

The tests that an independent QA Group develops will often test various aspects of the system that would be missed by both the users and vendors. For this specific effort, tests of

"an accounting nature," naive operator understanding, ease of use, error conditions, and uniformity were tested by the involvement of the independent QA Group.

Problem Report Controls

QA can act as an effective mediator between user and/or vendor during the testing process. For this specific effort, the independent QA Group insisted that formal trouble report sheets be used, priorities be established, enhancements be recognized and as such postponed, approval from the user be given before changes were made by the vendor, and an appropriate retesting cycle established for RETEST.

Configuration Management

The knowledge of what is being used or tested is essential to the testing process. During this specific effort, the independent QA Group required that version numbers be placed on every report and screen, the file interaction be made clear (levels of complexity), all subprograms be identified, and the system be tested in a stand alone environment, with no changes during the test process.

These four aspects of test control greatly enhanced the control of the project and were accepted well by the user and vendor.

The actual applications of these techniques will be described next.

METHOD OF QA APPLICATION

There were five steps through which the QA Group added value to the effort. These were comprised of the following:

1. Quality assurance review
2. Specification sign off
3. Test plan development
4. Test case assistance
5. Compliance test participation

A quality assurance review was first conducted to establish a base point for the project. The level of involvement of the independent QA Group needs to be modified by the level of testing or assumed responsibilities of the other participant groups. In this effort, the quality assurance review exposed the fact that no project schedule, no test plan, and no contingency plans existed. The development of these was recommended to the user, who developed the documents with QA's assistance.

Specifications for the major phases were close to sign-off when the independent QA Group became involved. Therefore, QA briefly assisted with this effort. The QA Group sat in on review meetings and requested clarification on several points. Because of QA's attention to detail, its direct input clarified numerous vague statements and formulae. Basically, the independent QA Group acquired industry expertise while reviewing the specifications. Thus by initially bringing a naive point of view to the review process, they helped ensure a more product-representative specification. This eventually led to satisfactory specifications sign-off between the user and the vendor.

It was discovered by the independent QA Group that no test plan had been developed for either the vendor or the user. The independent QA Group recommended that a test plan be developed and reviewed by both groups. This clarified the responsibilities of both parties and identified all efforts.

Since the user had not yet developed test cases, the independent QA Group acted in an advisory position and also actually helped develop the test cases. Five increasingly complex iterations were selected as sufficient to prove the acceptability of the software. Test data and expected results were created by the QA Group and user for these five iterations. The independent QA Group also wrote specific tests to detect if invalid entries were being trapped by the system.

During the actual testing, the independent QA Group ran tests by inputting data at the terminal, recording results, and organizing the cycle process. This methodology was used in subsequent RETESTS.

Sufficient errors were discovered to warrant a retest. These errors were prioritized, discussed, and agreed to correction by both parties, with the independent QA Group acting as mediator. The QA Group provided a very useful function as mediator, because it allowed both user and vendor to vent their feelings. After the discussions, the QA Group would help find a "best-fit" answer.

From the comments by both vendor and user, it was obvious that the QA Group role was effective and perceived to be beneficial by both parties.

EFFECT OF APPLICATION

Any interface has both positive and negative effects, and so did the involvement of the independent group in this effort. Overall, the benefits of the QA Group's involvement far exceeded any negative aspects, and the entire effort was seen as productive.

Some of the benefits included the following:

1. Security
2. Role definition
3. Information transfer

The development of plans and schedules assisted the effort by giving guidelines and the ability to measure progress. The user group thus felt there was direction to the installation process, and they had a tangible means of relating to the effort. Also the user had a basis to report progress to upper management. This measurement gave them a good feeling of security.

Role definition was made possible by assigning the identified tasks to specific individuals. Thus, on an individual personal basis, people knew their activities to achieve the desired results.

Most important, information was transferred among all parties. This yielded a better understanding and increased appreciation for each other's jobs. More than anything else, the transfer of information yielded greater system satisfaction by

the user. Also, the vendor gained knowledge of the user's environment that would prove useful in future modifications to the system.

However, the involvement of an independent QA Group was not entirely positive. Some of the negative aspects of the QA Group involvement are the following:

1. Cost in dollars
2. User dependence
3. Group qualifications

The role of the independent QA Group can be expensive, depending upon their level of involvement. In this specific effort, multiple tasks were actually done by the QA Group instead of the user. Some of these tasks (test data preparation) could have been completed by the user had there been sufficient staff. The QA Group could then have done a management or monitoring of the process rather than actual participation. The degree of actual involvement by the QA Group will vary, depending upon the tasks which can be done by the user group. Therefore the user should be made aware early of the QA Group's responsibilities and activities in order to plan for possibly less expensive alternatives or to allocate budget for the effort.

Another problem in the involvement of an independent QA Group is the eventual user dependence on the group. In this specific effort, because of the depth of QA Group involvement, the user relied upon the QA Group to act as an intermediary and in many cases desired that it authorize changes to the system. The QA Group should *never* do this, because it will possibly lose its independence. QA should act as an advisor or mediator only in this role; and if they feel the user is relying too much on them for direction, the QA Group should seek to correct the problem.

The final problem of group qualifications is one which will affect many efforts. When one searches for the best group to function in the independent QA role, a select set of characteristics emerge. In this specific effort, the independent QA Group lacked industry expertise (this lack, however, was turned positive through naive testing). Although not all characteristics will probably be found in a candidate group, the following (in decreasing order of priority) should be weighed:

1. State-of-the-art quality assurance views
2. Multiple-level testing experience
3. Maintenance program responsibility experience
4. Applications systems development experience
5. Specific industry expertise

These should help determine the group with the "best fit" for the job.

SUMMARY

An independent QA Group is invaluable in the verification and validation, and acceptance test process. QA can fulfill the functions of mediator, director, advisor, and participant. These functions are especially needed in a vendor-supplied, complex software environment in which the user/purchaser and the vendor may clash during the final testing phases. The user and the vendor represent different disciplines and necessarily differing points of view. Most important, through the use of an independent QA Group, communication between these two groups can be greatly improved, and the delivery of a more functional system can be aided.

At its best, the QA Group can assure a trouble-free, well controlled acceptance of a jointly agreed upon system. At its worst it is a costly, dependency-inducing process. In any case, both vendor and user have much to gain by the involvement of an independent QA Group.

Quality assurance in a large commercial data processing installation

by C. W. LYBROOK
Chemical Bank
New York, New York

ABSTRACT

Quality assurance (QA) is one of the most misunderstood, highly desired, yet inconsistently defined, organizational entities in the data processing industry today. There is an increasing number of articles and literature available on the subject. Senior management finds it very attractive. In spite of this, however, few commercial data processing organizations have successfully assimilated effective QA programs into their organizations. This paper describes such a program by the Information Services Group (ISG) of Chemical Bank. Particular emphasis will be placed on the organizational defintion of QA and the philosophy of its operation.

INTRODUCTION

Quality assurance suffers from a lack of standard definition. When discussing QA with different companies, it is very important to begin the discussion with a carefully worded description of what you mean when you refer to QA. What are the component parts? Where does the function report? What is the management philosophy behind the organization and how does it know that it is satisfying the objectives as viewed by senior management, middle management, and technical personnel? Anyone faced with the challenge of establishing a QA organization has been faced with these questions. They are not easy questions to answer. In light of the lack of standardization and of the general lack of understanding by management and technical personnel regarding the function itself, QA becomes a most difficult organizational unit to create and manage. In this paper the Chemical Bank QA organization and the organizational structure, goals, and philosophies are discussed.

THE CHEMICAL ENVIRONMENT

Chemical Bank is the sixth largest bank in the United States. It has the second largest retail branch system in New York City. There are many offices overseas and throughout the United States, with over 18,000 employees. Chemical's assets are in excess of $40 billion.

Most data processing within Chemical falls under the management of the Information Services Group (ISG). ISG has responsibility for all of the general purpose computing for the bank, as well as most of the special purpose computing. Special purpose computing outside the management concern of ISG includes check processing, the processing requirements of several subsidiaries throughout the country, and data processing for our international branches. ISG has responsibility for the operation of three computer sites: one in New York City, a second in New Jersey, and a third on Long Island. ISG operates multiple IBM 3033s and 3032s as well as a significant number of smaller mini systems. An extensive, standard communications facility is available, providing the Bank with a worldwide communications capability.

The quality assurance organization is one of the units reporting to the senior vice-president in charge of ISG (see Figure 1). There are two Systems Development Groups with a total staff of over 400 programmers and analysts. The 1982 ISG staff includes more than 1,000 people, with a budget in the range of 100 million dollars. Systems Development (SDD) develops and maintains business application systems in a matrix-like fashion, working very closely with both user groups as well as database and communications technical support groups (represented in the organization chart as Information Systems Integration [ISI]). In most cases, the user groups are surrogate users (not the end user) and are formed into what we call Automation Groups. This relationship of SDD to users and technical staff becomes very important when understanding the role that QA plays in the development process.

QUALITY ASSURANCE ORIGINS

The quality assurance function was established in late 1978/early 1979. It started as a staff of four. There were four start-up activities: systems assurance, change control, project accounting, and standards and procedures.

It was created by consolidating existing functions which were placed in other parts of the organization. For example, at that time, ISG had a standards and procedures group as well as a systems assurance function reporting to the database group. Change control was in the computer operations area.

The ISG budget was about $24 million. At that time, ISG was experiencing the problems that many people consider typical of commercial data processing: all too frequent delays and cost overruns in Systems Development; general instability in the computer operations environment; and an apparent lack of acceptable administrative, developmental, and operational standards and guidelines that were used by the personnel in the organization. This is not to suggest that the introduction of QA solved all the problems nor is it to suggest that it was introduced in response to a specific crisis. Our budgets were growing rapidly, and change was being introduced into ISG at an accelerated rate. This accelerated rate of change, coupled with dramatic personnel growth and increasing complexity (from both a technological and organizational perspective), caused severe strains on management. It became evident that we needed to standardize and institutionalize many management and technical functions if we were to be successful. QA was but one of several management improvements that were introduced to help us manage better the data processing and communications enterprise. In establishing QA, however, several specific objectives were identified.

We wanted to establish a quality environment and to reduce the tendency for crisis management through problem avoidance. We wanted to be able to provide rules and consistency in terms of the way we worked and the products that are produced.

We also wanted to provide consistency in terms of the way we conducted business with our clients. Our final objective was to raise the level of productivity and productivity awareness. For ISG to be successful we had to have a disciplined development process, provide for continuing integrity of our production systems, increase our organizational productivity, and raise our level of organizational effectiveness. These overall objectives, as expressed above, were translated into a QA

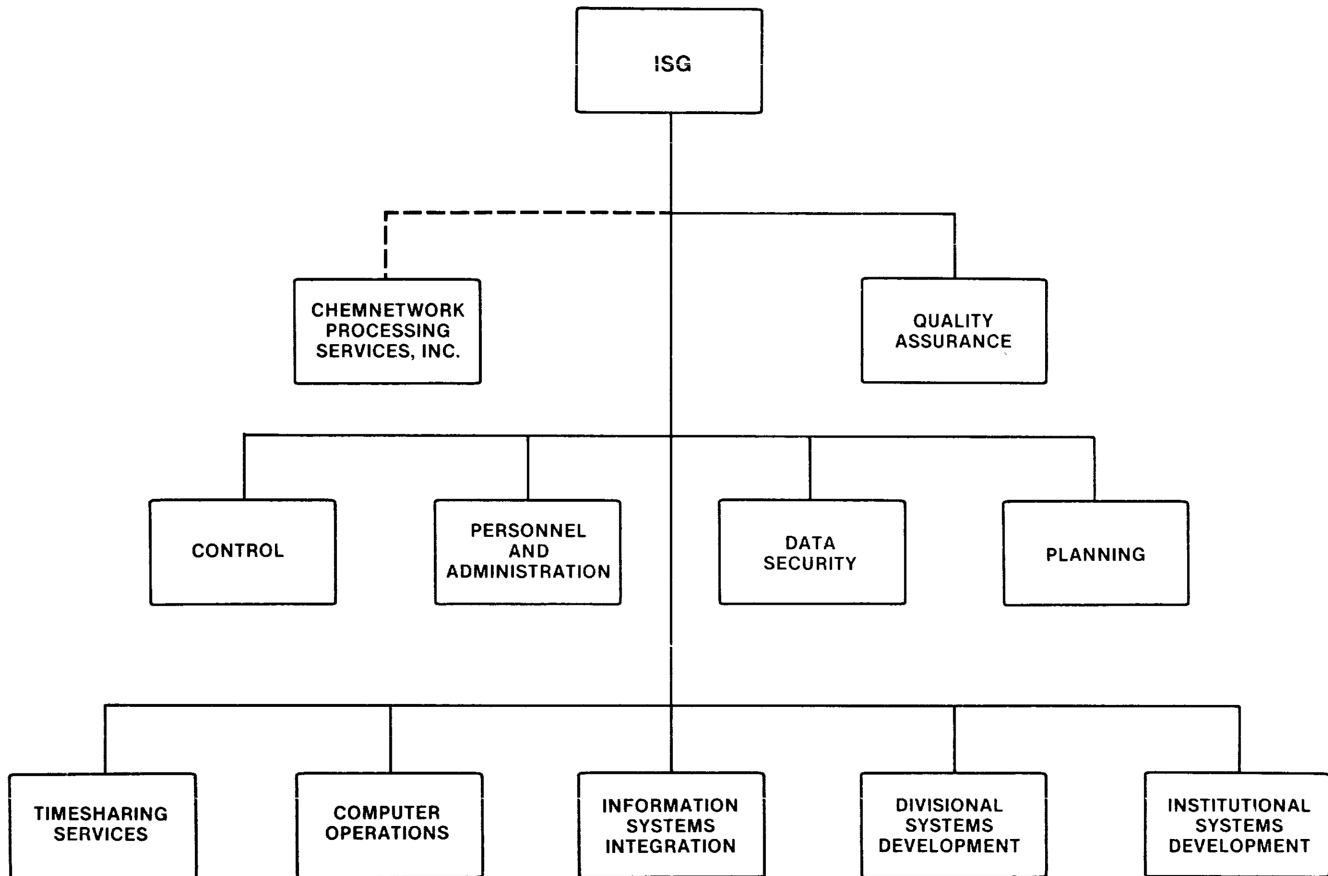


Figure 1—ISG organization

charter and set of responsibilities as outlined in Figure 2.

An important part of our approach in creating QA was to view the establishment of QA as a management process, not as a series of technical challenges. We believed that QA must have a management orientation and that the function had to become an integral part of the management process.

QA MANAGEMENT PHILOSOPHY

There are essentially two ways to approach QA. One is for QA to view itself as the watchdog, or policeman, of the organization. There is a significant risk to QA in assuming this posture. When taking that role, QA tends to establish an adversary relationship with other organizational units. Informal communication generally tends to break down and this, in turn, can give the QA organization additional problems.

The group tends to begin relying solely on formal communication channels. This can evolve to the point that QA acquires a focus on "after the fact standards checking." When such occurs, QA not only loses its credibility, but also its effectiveness. Soon senior management starts to question the payback of QA. There are many examples of unsuccessful attempts at starting QA, and I suggest that scenarios such as the one above have accounted for many of these failures.

A DIFFERENT APPROACH

I would like to suggest a different approach to QA, one of active and positive management participation. QA must play a leading role in the management of the organization and must have processes and check points in place that facilitate the identification of problems before they occur. QA should be able to identify and understand the technical issues, while applying management judgments in terms of recommendations for resolution. Managing a large data processing organization is indeed a challenging and demanding job. Data processing management is constantly dealing with large, complex issues, and there exists a need for third party objectivity to current issues and problems. This would suggest that QA must be in a position to address a technical issue and help interpret the issue in management terms in such a way that management can take timely, corrective action. In my view, one of the worst things that a QA organization can do is to determine that a situation is out of control when it is too late to take corrective action. The anticipation of problems, being in a position of knowing and anticipating a problem, is a key to having senior management support for the QA activity.

Assuming a positive management stance provides many benefits. One favorable result of functioning this way is that

QUALITY ASSURANCE
CHARTER AND RESPONSIBILITIES

The primary purpose of Quality Assurance (QA) is to foster a uniformly high level of quality in EDP Systems developed and installed on behalf of the Information Services Group. This purpose is achieved by means of provisions which have been set up:

1) for assuming active and coordinated participation in considerations leading to the establishment, revision, evaluation and dissemination of standards, management guidelines, and procedures;

2) for research into definition, establishment, enhancement, and maintenance of a systems development methodology(ies);

3) for consultation, review and evaluation of computer projects at significant milestones in their development;

4) for establishment, enhancement, and maintainance of a methodology to facilitate the orderly introduction of change into the operational environment; to manage the process of introducing such change into the environment;

5) for research into definition, establishment, and maintenance of a standard, consistent, and well-defined testing methodology;

6) for implementation and maintenance of an automated project control and project planning system that facilitates planning and project accounting;

7) for providing the impetus and focal point for the successful and timely introduction of new technology to ISG's computing environment.

Figure 2—QA organization charter

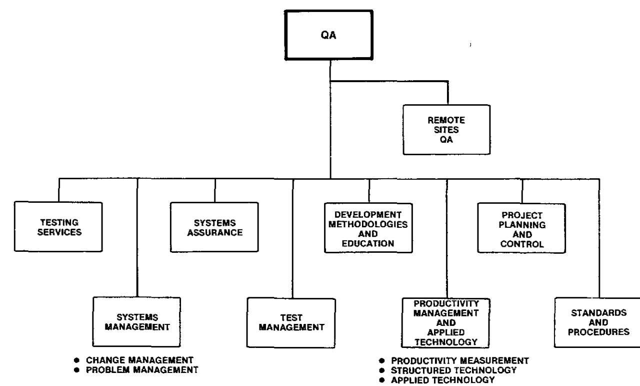


Figure 3—QA organization

information becomes more readily available to the QA staff. Managers, project managers, and programmers are less reluctant to discuss their problems. Certainly there is a lot to learn from looking at a document, and a lot can be gained by working in isolation. However, the key to QA effectiveness, the key to producing a positive effect on the company and the data processing organization, is being able to assimilate all that information, including what you learn in the hallways; exercising good judgment; and making timely recommendations to management in such a way that no one is embarrassed. A lot of work must be accomplished behind closed doors, and a lot of persuasion takes place. QA should become part of the management process, have a positive influence on events, and become involved in organizational decision-making. Much of this involves the availability of information. If QA is considered by the organization to be a help rather than a hindrance (and a bureaucratic hindrance at that), it starts to flourish. Like a snowball rolling down a hill, it gains momentum. In this kind of a situation, people will look for QA's help as a third party. This is fundamental to the way we have approached the introduction of QA at Chemical Bank.

Philosophically there are essentially two ways to approach QA. One is to stop people from doing things incorrectly. The other is to help people do things correctly. There is a great deal to be gained by choosing the latter approach.

ORGANIZATIONAL COMPONENTS

The QA organization at Chemical Bank is composed of approximately 35 people and is organized into eight (8) units (see Figure 3). There is a high ratio of experienced people in the organization. This is not only a result of the need for quality analytical work but also a reflection of the need for management oriented interpretation of technical issues and information. The remainder of this paper discusses the component parts of QA and provides some insights into how the organizational units interrelate and reinforce each other.

DEVELOPMENT METHODOLOGY AND EDUCATION

One of the key requirements for effective QA is to provide the organization (ISG) with well-defined methods of work as it relates to systems development. This means having a well understood systems development methodology. I will put everything from breadboarding (prototyping) to structured techniques under the umbrella of development methodologies. To be effective, systems development methodologies must be simple to use, simple to understand, and flexible. They must also be adaptable to changing technologies, changing organizational environments, and changing needs. This unit of QA is concerned that the methodologies in use for systems development and maintenance are effective for ISG and the Bank. In a sense, this unit provides the cornerstone for work accomplished by the other units of QA and ISG. Chemical Bank uses a *project life cycle* (PLC) as defined in Figure 4. This PLC provides the overall framework for most other QA activity. The PLC is straightforward and very easy to understand. It provides standardization in terms of the way we develop and maintain our computer systems. The PLC is occasionally modified, as necessary, to meet the needs of the organization. It is the responsibility of this group to define and properly coordinate PLC changes and to provide education

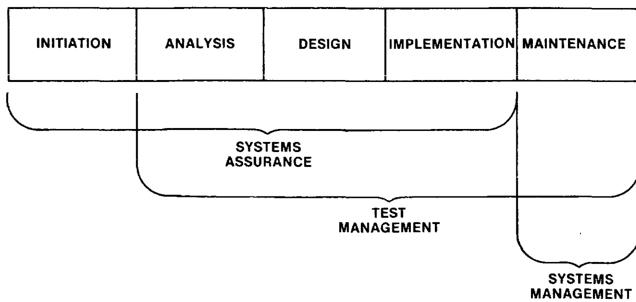


Figure 4—ISG project life cycle

regarding the nature of the change to affected groups. There are four major users of the PLC from a QA perspective:

1. Systems Assurance
2. Project Planning and Control
3. Systems Management
4. Test Management

SYSTEMS ASSURANCE

The cornerstone of our development process concerns project reviews (Table I). Systems Assurance is responsible for reviewing and monitoring all PLC projects while they are under development. It holds reviews on these projects at defined milestone points in the development process. We have found formal project reviews to be very effective. These meetings are always chaired by Systems Assurance and are held at predetermined points within the life cycle (see Figure 5). For example, once requirements are established, there is a formal

TABLE I—ISG project life cycle reviews

PROJECT LIFE CYCLE/REVIEWS

REVIEW	PHASE
PROJECT INITIATION MEETING	INITIATION AND SURVEY
REQUIREMENTS REVIEW	ANALYSIS
SECURITY AND AUDIT REVIEW	ANALYSIS
TECHNICAL ALTERNATIVES REVIEW	ANALYSIS
PROPOSAL REVIEW	ANALYSIS
ARCHITECTURE REVIEW	DESIGN
SECURITY/AUDIT DESIGN REVIEW	DESIGN
CRITICAL DESIGN REVIEW	DESIGN
PROJECT COMPLETION MEETING	IMPLEMENTATION
PRODUCTION ACCEPTANCE MEETING	IMPLEMENTATION
WALK-THRU	any phase
IN-PROCESS REVIEW	any phase

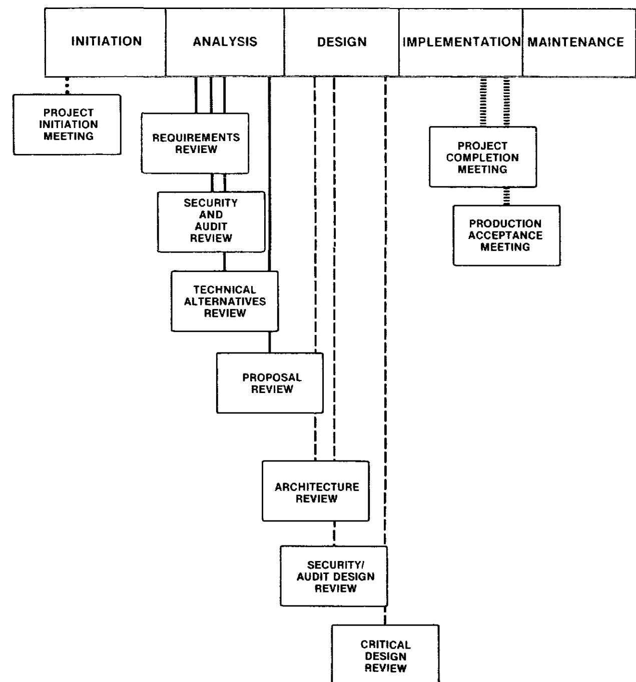


Figure 5—ISG project life cycle/review

requirements review. It is at this point that we ensure that all parties are brought into the process. All groups must either sign off on the requirements definition or express their reservations. A management report of review is drafted by Systems Assurance and delivered to the director of data processing for comment and/or signature. The signed report is then sent to all project and management personnel, including senior user management. Most reviews result in the creation of action items (things to be done by the project team). These action items are listed in the management report. Systems Assurance will track action items to ensure their completion. Other reviews in the life cycle follow the same basic process.

People from EDP Auditing, Data Security, Information Systems Integration, Computer Operations, Systems Development, and the user areas are present at every review (Table II). Other participants at the reviews vary, depending upon the nature of the project and the PLC phase in question. Information Systems Integration is responsible for all architectural questions and, as such, they are with the project from the very inception. This is also true of computer operations people. They attend all reviews.

The Systems Assurance Group is composed of project managers drawn from other ISG groups (usually Systems Development). It is considered a positive move in terms of their career development. There is every attempt to ensure that they have organizational credibility and exceptional communication skills. They must exercise good judgment and tact in dealing with people. They are asked to be helpful in their approach. Most development project managers, when approached that way, are receptive and will take advice.

TABLE II—Project life cycle review attendees

REVIEW PARTICIPANTS

ALWAYS	WHEN REQUIRED
USER	CONTROLLER
DEVELOPMENT	PLANNING
OPERATIONS	BANK ACCOUNTING
COMMUNICATIONS	TRAINING
SECURITY	
EDP AUDITING	

Competence, good judgment, and a positive attitude of Systems Assurance personnel are fundamental to the success of the project review process. Systems Assurance people must exercise good judgment and understand when a problem needs escalation. Tom West of Data General once said, "Not everything worth doing is worth doing well."[1] These are very appropriate words for Systems Assurance Personnel to heed when reviewing projects. Our objective is to help get systems out the door, systems that can be operated and maintained *cost-effectively* by ISG. Knowing what is important and understanding when to take a firm stand is critical. It is sometimes counterproductive to require compliance to the letter of the law in striving for the last 10% of perfection.

The role of Systems Assurance at Chemical Bank can perhaps best be described by the following excerpts of a letter from the head of ISG to a senior user manager at the Bank:

"Attached is the Requirements Document and supporting Approval Package for the . . . System. As a part of the project management procedures specified in the Project Life Cycle, ISG, through its Quality Assurance organization, conducts a series of reviews at critical points throughout the life of a project. One such review is conducted immediately prior to the submission of a proposal. It is called the Proposal Review and was conducted for this system on April 17th. A follow-up was conducted on May 6th to resolve the action items generated at the reviews.

Quality Assurance representatives concluded that although the requirements are well documented and the project is sound from a technical viewpoint, the completed system may be cumbersome and expensive to operate. (More detailed comments by Quality Assurance are included in the last Tab of the proposal package.)

I want to emphasize that Quality Assurance exists to protect the Corporation and the user as well as to insure proper ISG performance. I am very well aware of the fact that any time a Quality Assurance organization reaches a conclusion other than "all is well," it is often viewed as being a roadblock to progress or supporting a vested interest of one of the players. I am also aware that the Quality Assurance finding in the extant instance can be viewed as "covering ISG"—if the system turns out to be a failure (however defined), ISG is vindicated; if on the other hand the system is successful, ISG can say that it was only through dint of ISG effort that success was achieved. Finally, I don't doubt but that at some levels in both your office and mine, this project is being viewed as a series of tedious confrontations on both sides—a "we vs. they." Only reasonable and strong management is going to overcome or at least negate such feelings.

I have been involved in many projects such as this and a reasonable share of them have turned out to be unsatisfactory. I think that the Quality Assurance finding on this project deserves your special attention and I highly recommend an independent assessment of the Requirements by a third party. This should not reflect on your staff but should help to assure that you are going to get the system you need and one which the Corporation can afford."

It is also perhaps obvious that such senior DP management support is also critical to the success of Systems Assurance.

PROJECT PLANNING AND CONTROL

Another QA organizational unit at Chemical Bank is Project Planning and Control. This unit is responsible for providing support and automated capabilities in the following areas:

1. Project planning
2. Project accounting
3. Project monitoring

We currently use PAC II® for these purposes. Once again, the project life cycle provides the overall framework for the products and activities of this area. In terms of project planning, standard PLC activities and deliverables are defined in the planning model. This proves to be a great aid in terms of the standardization of development work. It is also an aid to Systems Assurance in monitoring development activity.

In 1979, ISG established a budgeting and project accounting methodology that categorizes ISG activities into three major groups for management purposes:

1. Minimum maintenance
2. Discretionary enhancements
3. Development work

Minimum maintenance is defined as that expense level necessary to continue to operate and maintain the current portfolio of computer application systems. In addition to programming staff, it includes the systems support infrastructure necessary to operate the computer complex. In this sense, technical support personnel, equipment, and supplies needed to keep the computer applications running are included in the minimum maintenance category. This is a baseline component.

Unlike baseline, expense levels for discretionary enhancements and development are largely controlled by the user community. The identification and justification of new work and the decisions to add functions to existing systems are user driven. However, once the bank commits itself to the development of a new system and after implementation of that system, the ISG baseline will be driven upward to accommodate the continuing maintenance requirements. There is,

therefore, a direct correlation between ISG baseline and the impact of past decisions. The identification of ISG work in such a manner offers management an opportunity to assess ISG workload characteristics in terms of spending patterns and work type trends. The objective of ISG management is, of course, to reduce the amount of resources (in a relative sense) required to maintain its portfolio of business application systems. This, in turn, provides the Bank greater opportunity to devote more resources to new automation. While accounting for the fact that past decisions cause baseline increase, ISG strives to control the rate of increase in maintenance work, thereby maintaining a favorable ratio of development to maintenance activity.

SYSTEMS MANAGEMENT

Another aspect of quality assurance concerns the operational environment. This is the concept of Systems Management. Systems Management is composed of two functions, Change Management and Problem Management.

The objective of Change Management is to minimize the risk of making applications and systems changes in our operational environment. This is much easier to do in a large, centralized computer operations area operating mainframes than in a distributed systems environment where the hardware and software are operated within the user area. Change Management is responsible for the establishment and maintenance of a process to accommodate the planned, orderly introduction of all application and software changes. As such, all work being requested of ISG is sent to a *work request desk,* managed by Change Management. When the work in question is completed, the request to make the change to the computer system must also be sent to the work request desk. This process affords ISG an opportunity to closely monitor and analyze all changes taking place in the operational environment.

One method to help reduce the risk of change is to reduce the frequency of such change. All users are requested to adhere to a planned change cycle for their business applications. The cycle frequency is determined by them, in concert with ISG Systems Development. Once-a-month change cycles are not uncommon. The checks and balances implied in this process provide ISG an opportunity to reduce the risk of introducing software change.

Another way of providing for more operational stability is through the process of problem management. Problem Management provides for a systematic way of identifying, categorizing, assigning and tracking problems that occur in Computer Operations. We currently use an IBM product "Information Systems" for automated support. There are terminals strategically located in the data centers. As problems are encountered, information concerning the problem is keyed into the system (we are in the process of building automated interfaces to SMF, RMF, etc. to reduce the manual intervention).

The group in Problem Management monitors all this activity and ensures that the problems are being properly identified and assigned. They also monitor the process to ensure that follow-up action is being taken and that the problems are being resolved.

TEST MANAGEMENT

Software testing is one of the most critical tasks performed by a large data processing organization. Testing is important in the development of new systems, but it may have an even greater effect on the maintenance of production systems. In spite of this, testing is rarely approached in the same disciplined manner as other software production activities. This neglect of software testing is not, however, a result of the lack of available technology. Over the last several years, software testing has been the subject of intense activity in the research community, and many books and articles can be found on the subject.[2, 3] An organization can often achieve significant improvements in both software testing effectiveness and efficiency through a relatively low-cost investment in testing methodologies, tools, and techniques.

The Test Management staff of QA is responsible for providing ISG with a standard methodology for testing. This standard testing methodology is closely integrated with the PLC framework (see Figure 6). As a project moves through the various phases of the development or maintenance life cycles, the program guides and identifies testing activities to be performed and possibly documented. The review points established in the PLC provide the opportunity for QA to evaluate testing plans and progress at critical life cycle milestones.

Test Management has defined three progressive levels of increasing involvement with the testing of individual systems in development or maintenance:

1. Testing program review
2. Testing coverage audit
3. Independent testing

Referred to as certification levels, these procedures provide increasing organizational assurance of system reliability through third-party review.[4]

Testing Program Review

This is the certification level for most systems at Chemical. It uses the project life cycle review points to verify adherence to the standard testing program. These reviews are handled by Systems Assurance in the development cycle and by Change Management in the maintenance cycle. Test Management provides technical assistance to both project teams and the QA review teams in preparing for these reviews.

Testing Coverage Audit

At this certification Level, QA Test Management uses the TRAILBLAZER tool (including its change analysis feature for systems in maintenance) to assess independently the thoroughness of test data provided by the systems developers. Standards of thoroughness ranging from 75% to 95% coverage of (changed) program logic are established as part of a system's testing strategy or regression test manual and agreed to by QA, the developers, and the user(s). If these standards are met, QA certifies the system; otherwise, the detailed reports showing unexecuted logic are returned to the project team for additional testing.

STANDARD TESTING METHODOLOGY

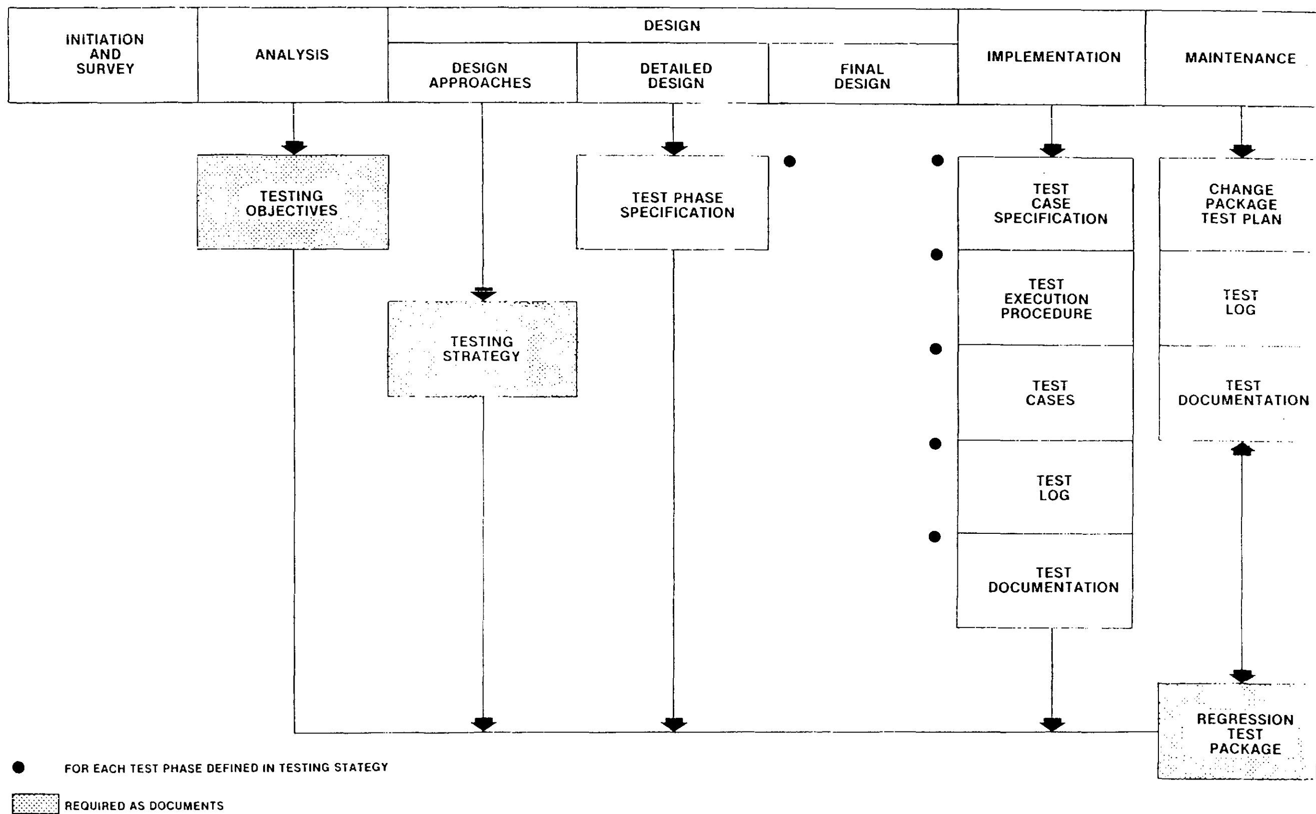


Figure 6—ISG project life cycle/testing

Independent Testing

This highest certification level transfers system testing responsibility from the development organization to QA. It is relatively costly, since QA analysts must understand the applications area of the system under test in order to do an effective job. In an organization with a high volume and diversity of applications being developed and maintained simultaneously, this approach can be justified for only a few, extremely critical systems.

TEST SERVICES

The success of the Test Management program is, to a large extent, dependent upon a reliable test environment. Chemical Bank has significant resources devoted to testing and, as such, is concerned that testing services are consistently available and that performance is reliable. A Test Services Group was established in QA to monitor the ISG test environment, seek new tools and methods to improve the service, and to follow up on performance problems.

The group is responsible for:

1. Establishing and maintaining all management reporting functions as they relate to the performance and availability of the testing environment.
2. Performing liaison with all appropriate areas of ISG in providing a consistent and cost-effective level of testing service.
3. Performing liaison with the users of the testing environment to ensure proper education and training in the utilization of the test environment.
4. Establishing and communicating appropriate policies and procedures in the use of TSO and other test systems (CICS, IMS, etc.).

FUNCTIONAL INTERACTION

Testing, Systems Assurance, Systems Management (Change and Problem) and other facets of quality assurance bring individual benefit to the company. However, it is important to remember that the real payback is when these functions begin to interact and support each other. Accept for a moment that there are two ways a system can be changed: (1) through problem correction and (2) through a user-generated change (see Figure 7). Also accept that there are two outputs of that process. One is a product of some nature (a report, CRT screen, etc.); and the other is another problem. This could be conceptualized as the normal processing cycle.

One of our objectives is to institutionalize procedures whereby the various groups reinforce each other. Organiza-

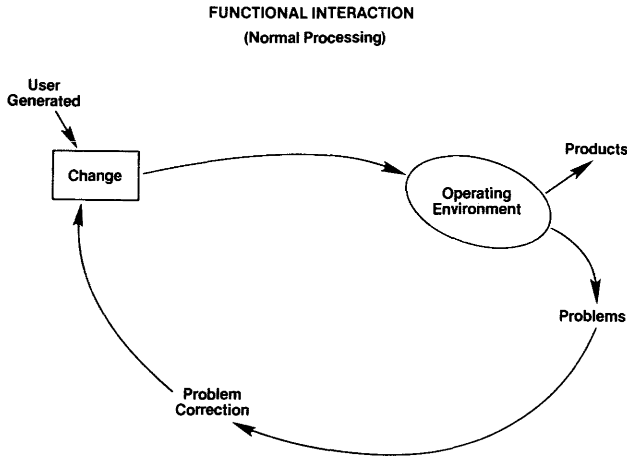


Figure 7—Normal processing

tional payback is greatly increased when the units begin working together as seen in Figure 8 and described below.

In the Problem Management organization, we establish a threshold of problems for each production system and keep track of the problems encountered with that system. Once the threshold is reached, the Problem Management group notifies Change Management. The Change Management Group places it on a key problem list. As long as the problem threshold on the application in question is being exceeded, any new changes for that application must be rerouted to Test Management. The Test Group has two options. They can start doing third-party independent testing on those changes or they can start reviewing, through the use of their test tools, the program coverage of those changes until the problem level falls below the threshold. Through such interaction, the organization begins to get significant payback from QA.

STANDARDS AND PROCEDURES

I will briefly cover Standards and Procedures. It has been said that data processing people do not believe in standards. We all

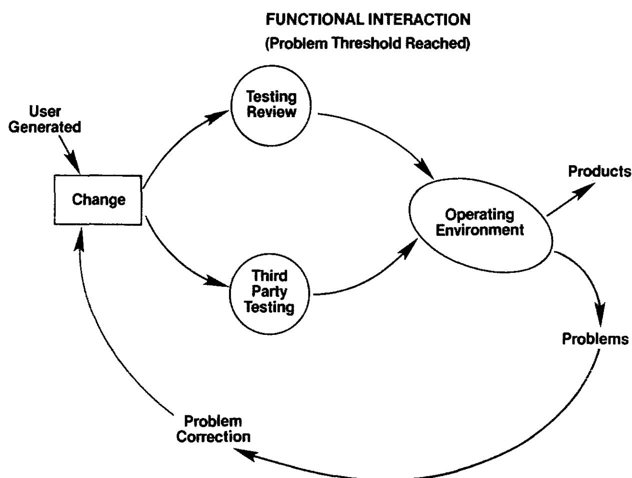


Figure 8—Problem processing

are aware of manuals gathering dust on the tops of desks. Standards is traditionally a paper-driven process.

When the creation of standards is a unilateral process accomplished by a Standards and Procedures Group, the users of the material have a tendency to ignore them. Often Standards and Procedures Groups are not responsive, taking months and months to get new material out the door. Oftentimes the standards are partially outdated by the time they are distributed. We developed a program to overcome these problems by focusing on three things:

1. Introduce the idea of ownership. Foster the thought that Standards and Guidelines belong to the area most directly affected by the Standards and Guidelines in question. Give the users of Standards and Guidelines a piece of the action.
2. Move from a paper-driven process to utilization of office technology. Create, update, and deliver material electronically. Cut down the lead time required to deliver material.
3. Create standards, guidelines, and requirements that reflect minimal needs (a standard is not a standard unless it is machine enforceable). One area that needs most attention in this regard is Systems Maintenance documentation requirements. Requiring extensive narratives that describe systems applications functions are usually counterproductive (the use of structured analysis and design is helping to overcome this problematic area). Require only that which is necessary.

In our view, Standards and Procedures is an internal service organization. It is staffed with professional technical writers.

PRODUCTIVITY

Data processing (DP) and communications are playing an ever-increasing role at Chemical in helping the Bank meet the competitive challenges of the marketplace. To meet the demands being placed on EDP, ISG must be constantly seeking new tools, techniques and methods to assist in keeping cost and service at cost-effective levels.

ISG established a management unit with the responsibility to identify, assess, and, where appropriate, help implement new productivity-oriented products and procedures within ISG. Another aspect of this productivity program is to provide concise, meaningful, and easily understood measurements of ISG productivity. Quantifying data processing organizational productivity has been an elusive target of the industry for years. There is a great deal of theory written in journals and publications; but few, if any, organizations have arrived at satisfactory measurement techniques.

One key to understanding or discussing a productivity measurement program lies in obtaining a working definition of productivity that most people in the organization can accept. We define productivity as the ratio of an output produced by an activity to an input used by the activity. Our reference to productivity and productivity measures refers to this rather simple, working definition. ISG has taken steps to better understand the DP productivity issues, and we have begun to apply measurement techniques to our activities.

There are two fundamental objectives that were established in the measurement of ISG productivity:

1. To better understand and measure the effectiveness of the products and services that we provide to the Corporation. This "external" view is briefly discussed below and is undoubtedly the greater of the challenges in the area of productivity measurement.
2. To measure the efficiency with which we provide these products and services. This could be thought of as the internal view of ISG productivity.

Effectiveness

Measuring ISG effectiveness is undoubtedly the more difficult. The activity of an organization like ISG, as seen by senior bank management, is to provide appropriate services to the user divisions and, where appropriate, directly to bank customers (i.e., ATMs).

The measure of our success is how well the users are served in relation to the cost of the service. Thus, increasing the productivity of ISG is defined as: (a) increasing the service levels provided to the divisions and customers while consuming no more resources or (b) providing equal services while consuming fewer resources. Assuming the resources can be measured and the services provided are measureable, then either (a) or (b) results in an increased productivity ratio.

Efficiency

We view the measurement of ISG efficiency as having two dimensions: (1) a measure of the operations or productions set of activities and (2) a measure of the factors relating to the systems development process. Productivity, and the measurement of our productivity, is viewed as one of the key issues for ISG in the 1980's.

QUALITY ASSURANCE IMPERATIVES

Quality Assurance is like motherhood and apple pie. Everyone believes in it, but most are not sure how to define it in terms of its role in the organization. This paper has presented QA at Chemical Bank. In the process of implementing the function, certain needs and requirements for successful implementation have been identified.

Functionally Complete

To get maximum payback out of a QA function, the QA organization should evolve to the point where it has sufficient function whereby units can mutually support each other. In a sense, this creates the potential for making the whole stronger than the sum of the component parts.

Technical Competence

It is probably obvious that QA must be staffed with technically competent personnel. Historically, QA has been viewed by DP professionals as a dead-end road. Some DP organizations have, in some instances, moved incompetent people aside (to QA) to minimize their visibility and lessen the potential impact of their "mistakes." This view of QA is changing, and must change if an organization is serious about the function. There is a school of thought that DP will see the emergence of QA professionals and that QA will soon be considered a productive and meaningful career in and of itself. The DP industry (unlike manufacturing) has not yet matured to this point. I believe such will be the case, but in the meantime, we must attract (by providing visible career paths) well qualified and respected developers and other DP specialists to the QA area.

Management Orientation

A good technical decision is sometimes not the best management decision. Quality assurance must never lose sight of organizational objectives and not consider their work as a series of technical challenges.

Third-Party Objectivity

In the course of their work, QA can and must assume the role of an objective third party. This is particularly important to senior DP management. QA *should not* become a part of the problem. It is possible to work closely with a problem solver and be so closely associated with a proposed solution that, as viewed by senior management, QA becomes part of the problem. In that sense, QA can offer no alternatives or proposed remedies to senior management.

Information Source

To effectively anticipate problems, QA must establish itself as a fertile source of information for the organization and for senior management. The degree of success in this area is due, in large part, to the perception of QA by the rest of the organization. Formal reporting structures are important. The informal communications process is equally important.

Senior Management Peer Level

Quality Assurance must actively participate in the decision-making process of the DP organization and should be in a position to influence direction and strategy. There is great variation in data processing organizations in terms of the reporting level of QA. Groups producing papers on QA (GUIDE et al.) seem to begin with a discussion of reporting levels. We believe it is important for QA to report at the highest organizational level if management wants to realize maximum benefit.

Positive Contributor Towards Organizational Goals

This is the bottom line. Success in this regard is a reflection of the attitudes, philosophies, and posture of the QA organization. It is a function of the technical competence and orientation of QA. If QA is considered a hindrance to progress, i.e., a group that gets in the way and fails to provide added value, then the QA organization will not be able to make a positive contribution and will eventually fail. Quality assurance must thoroughly understand the goals, objectives, and problems of the corporation and the data processing organization. These concerns must be considered in daily activities.

REFERENCES

1. Kidder, T. *The Soul of a New Machine.* Boston: Little, Brown and Company, 1981.
2. Hetzer, W. C. *Program Test Methods.* Old Tappan, New Jersey: Prentice Hall, 1972.
3. Myers, G. I. *The Art of Software Testing.* New York: Wiley–Interscience, 1979.
4. Holthouse, M. A. and C. W. Lybrook. "Improving Software Testing in Large Data Processing Organizations." *AFIPS,* (Volume 50), 1981, pp. 353–359.

PERSONAL COMPUTING

Data-server design issues

by FRED MARYANSKI
Digital Equipment Corp.
Hudson, MA

ABSTRACT

The expected proliferation of local-area networks has created a need for network database servers. It is reasonable to view local-area network data servers as extensions of backend database systems. This paper addresses several critical data-server design issues: distribution of functionality, high availability, security, and performance. Particular consideration is given to applying experience with backend databases to the problems of data servers. Several design alternatives are proposed and evaluated in terms of their impact on reliability, security, and performance in a gross sense. The concluding section emphasizes the need for greater practical experience with local-area networks in order to more accurately weigh the tradeoffs of different data-server configurations.

INTRODUCTION

In the 70's the concept of a backend database system was introduced as a potentially cost-effective method of increasing the data-processing capability of third-generation mainframes.[6] Although some activity continues in this area, the backend concept has not been realized in a large number of commercially successful systems owing in a large part to the high-performance costs of communications.[17] With the expected advent of local-area networks in the commercial marketplace in the 80's, it appears that backend database technology may find a more suitable basis for its application. Strictly speaking, the local-area networks of the future will probably not contain backend machines in the traditional sense. Figure 1 pictures a backend database system in which

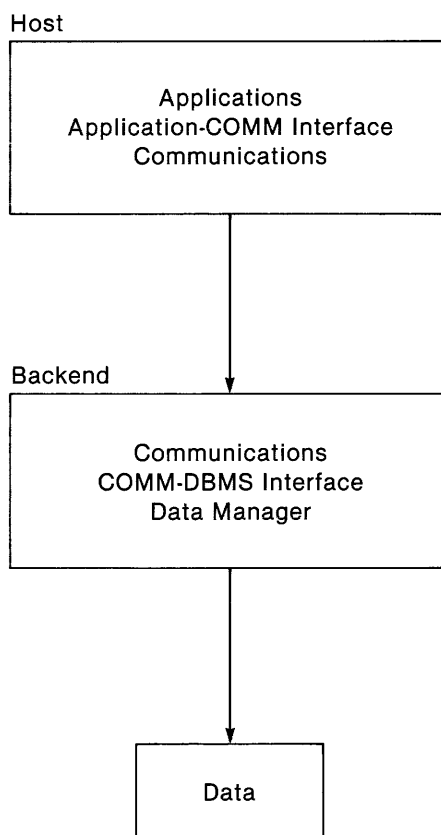


Figure 1—Backend database system

the database and the data manager are offloaded onto a dedicated processor that is tied directly to a mainframe host. The local area network equivalent of the backend database is the *data server,* which provides data-management facilities for the network. This paper considers the design of data servers for local-area networks, with an emphasis on applying the lessons learned with backend database systems to this new environment.

The next section provides references on backend database systems and mentions some related work on network data servers. The type of local network environment for which a data-server design is specified in this paper is defined in the third section. Critical design issues of the data server are covered in the following section. The paper concludes with a discussion of the key problems to be faced in this area and some suggestions for further work.

BACKGROUND

The groundbreaking paper on a dedicated database processor was written by Canaday and his associates in 1974.[6] The functionality of a backend computer has been investigated by several researchers who have detailed the progression from general purpose computers to special-purpose database machines.[4,13,22] A survey of backend database systems is available.[17]

From the point of view of data-resource management, there is a strong argument for dedicating processors to the data-management function.[20] Until recently, the high cost of wide-bandwidth communication imposed severe performance limitations upon such configurations. With the advent of local-area networks, it now appears feasible to designate one or more processors as network data servers. Several microprocessor data servers have been prototyped:

- The PHLOX project's network data server[8] runs on a specially tailored microprocessor in which the data manager and the operating system are integrated.
- The MINIMET[14] and MICRONET[25] data servers are composed of several microprocessors each containing a portion of the database. Each database request is broadcast to all of the microprocessors, which respond to the host independently. It is the responsibility of the host software to assemble the results.
- The UNITY system[26] consists of a microprocessor "data server" that has its data downline loaded from a minicomputer. The microprocessor server then operates in a standalone fashion. The minicomputer database is periodically refreshed from the microprocessor server. A basic design decision of UNITY is not to provide continuous synchronous access to the main database.

The local-area network environment for which the problem of data-server design is considered in this paper differs slightly

from the environments of prior studies. The structure of the network is presented in the next section.

LOCAL-AREA NETWORK STRUCTURE

Before we consider data server design issues, the environment in which such a system will operate must be defined. Figure 2 pictures the generic local-area network for which the data server will be defined. In all subsequent discussion, communication is assumed to take place using a high-bandwidth broadcast mechanism with perhaps some distance limitations. The issues of contention, nets versus rings, will not be addressed.[19,29]

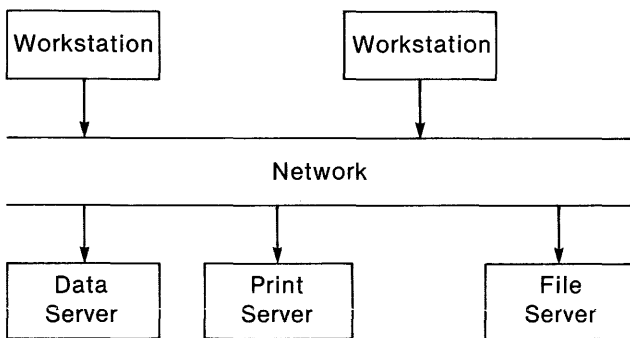


Figure 2—Local-area network

There are two basic types of processors in the network, workstations and servers. Workstations directly support user terminals. A workstation contains sufficient functionality to support the professional tasks of the user. The servers provide all users with access to shared facilities such as high-speed, high-quality printers, a file repository, and a common database. The data server is distinguished from a file server in the same manner that a database-management system is functionally distinguished from a file system in a single-machine environment. Descriptions of networks of this nature that have been developed for special application environments have been given by Kaisler and Lind.[11,12]

A basic principle of local-area networks is that the high-communication bandwidth and the declining cost of processors dictate the functional specification of processors. Such dedicated processors can be tuned, or perhaps specially constructed, to provide their allocated functionality in an optimum fashion. The specific concentration of this work is the design of a processor, or group of processors, dedicated to the data management function.

DATA SERVER DESIGN CONSIDERATIONS

In order for an efficient, reliable data server to be built, several important issues must be carefully considered. An attempt is made to address in this paper the most pressing of these issues, as indicated in the following list.

- Distribution of data management functionality
- Availability
- Security
- Performance

Distribution of Functionality

As in the case of backend systems, a data server cannot be created by simply placing the data-management software in a dedicated processor. The data-management software must be partitioned between the workstation and the data server. As with a traditional backend, additional communication software must be added to a data-server configuration in order to tie the pieces of the data-management system together. One of the lessons of the backend experience is that the record-at-a-time DML statement, a la CODASYL, is not the proper unit of interprocessor communication.[16,17,21] Even in a local-area network with high communication bandwidth, the frequency of interprocessor communication must be minimized. Thus a higher-level data-access language must be employed. A relational data-manipulation language such as SEQUEL or QUEL is well-suited for the data-server environment. The attractive feature of this class of languages is that a single request can result in the transfer of a large amount of data. The underlying data manager need not be relational, provided that a high-level data-manipulation language can be employed. Germano gives an example of a high-level data-manipulation language for a CODASYL data manager.[9] However, for purposes of this discussion, the data server will be designed to support the relation model.

A local-area network data server must support spontaneous queries as well as the execution of canned transactions. The discussion of the distribution of data-management functions will use the structure of System R, which is described in several articles.[1,5,7] As explained by Chamberlain,[7] PL/I or COBOL application programs may access a System R database by including $LET, $OPEN, $FETCH, and $CLOSE statements in the program. These statements perform the following functions:

- $LET—Defines the SEQUEL query as a transaction and names the program variables utilized in the query. This statement is processed by the System R precompiler to create an access module for this transaction.
- $OPEN—Binds the variables.
- $FETCH—For a retrieval operation, this function causes data from one selected tuple to be written into program variables.
- $CLOSE—Frees the variables.

The precompilation of System R transactions in a single-processor environment is depicted in Figure 3.[7] In the local-area network described in the preceding section, the compilation function resides on the workstation. Thus a copy of the data management precompiler must be available to every workstation. The question of the placement of the access module generated by the precompiler must be deferred pending consideration of the structure of the transaction run-time environment. The System R transaction run-time environ-

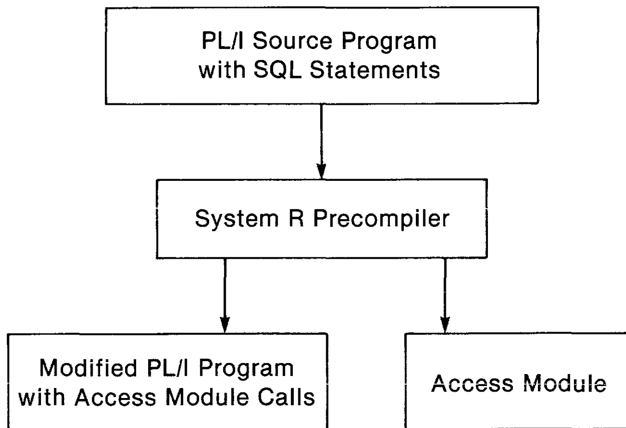


Figure 3—System R precompilation step

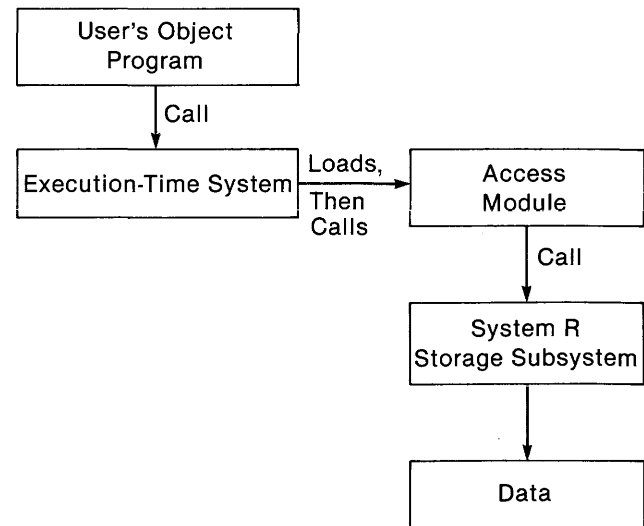


Figure 4—System R transaction execution step

ment is protrayed in Figure 4.[7] The most straightforward approach to partitioning the run-time environment is to assign the user object program to the workstation and place the remaining modules on the data server. However, if the interaction between the access module and the object program is examined closely, certain problems with the aforementioned partitioning strategy can be detected. In the precompilation phase, each of the System R operators just described is replaced by a call to the appropriate access module. If the program issues a request that results in the retrieval of several tuples, the $FETCH statement will be executed to retrieve the data for each tuple. Consider the sample transaction shown below.

```
$LET C1 BE
    SELECT NAME, SALARY INTO $X, $Y
    FROM EMP WHERE JOB = $Z;

DO JOB INDEX = 1 TO NUM JOBS;
    $Z = JOB TABLE ( JOB INDEX );
    $OPEN C1;

    DO WHILE ( SYR CODE ^ = DONE );
      , $FETCH C1;
        . . .

    END;

    $CLOSE C1;
END;
```

In a local-area network, if the entire execution-time system resides on the data server, then an exchange of messages between the workstation and the data server will be required each time the $FETCH statement is executed. In effect, the data server will be provided information on the same basis as if a CODASYL DML were used. Our experience with back-end databases has taught us to avoid this type of access.

The solution is to partition the execution-time system and the access module in a manner that permits all of the tuples selected in a retrieval operation to be moved from the data server to workstation local memory simultaneously. Individual tuples are then extracted from the workstation memory in response to $FETCH statements. The actions associated with each of the System R statements and their location in the network are summarized below.

- $LET—Processed in the precompilation phase. Pieces of the resulting load module are stored at both the workstation and the data server.
- $OPEN—The binding of the variables of the selection expressions occurs at the workstation using the load module. Then the database request is transmitted to the data server in the form of a call to the load module. At the data server, the request is executed. The result, in the form of status information and perhaps data tuples, is then transferred back to the workstation. The tuples must be buffered in the workstation memory.
- $FETCH—A tuple is retrieved from the workstation memory and the data associated with program variables. This entire operation is local to the workstation.
- $CLOSE—Frees the buffers that held the selected tuples. This operation is also local to the workstation.

The above discussion indicates that a substantial portion of the processing of database transactions is the responsibility of the workstation. Figure 5 shows the partitioning of the data-management modules for transaction handling between the workstation and the data server.

The processing of queries in System R resembles transaction processing with the exception of the precompilation step. The System R query-processing facility is illustrated in Figure 6.[7] The two specialized query operations are

- PREPARE—The operands of this statement are the text of a query and a set parameter, which are identified by a "?" token. The text is processed by the precompiler to produce an access module.
- EXECUTE—This statement is called with the name of a query that has been processed by a PREPARE statement

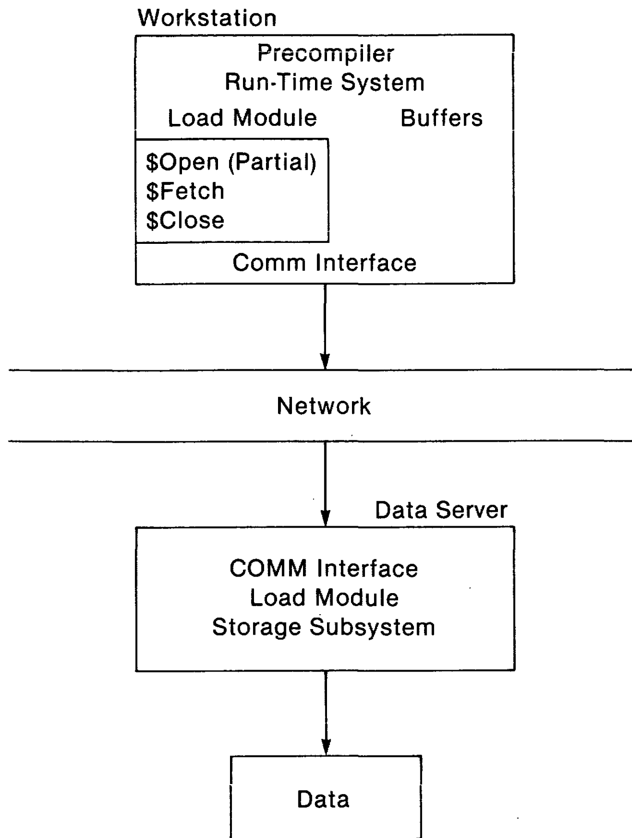


Figure 5—Distribution of data-manager functionality

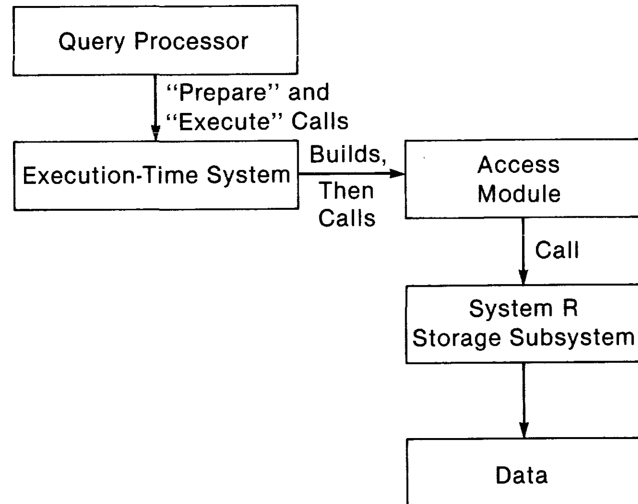


Figure 6—Availability

and the name of the formal parameters. The formal parameters are bound to the access module produced by the PREPARE statement.

These statements are used by the SQL query processor to build load modules for spontaneous database requests. These modules are called by the query processor using the $OPEN, $FETCH, and $CLOSE statements in the same manner as in the case of canned transactions. Once a query has been compiled, the SQL query processor accesses the database in the same manner as an application program. Since the processing of queries and transactions differ only in the precompilation step, there is no need to alter the data-server architecture described above to support a query facility.

Availability

In a discussion of data servers and local-area networks, it is quite common to hear a remark such as, "If I am going to have to put all my data on a data server, it better be reliable." One might be tempted to enquire about the reliability of the general-purpose machine on which the speaker's data are likely to be residing at present. However, it is easy to accept the argument that data servers must provide a high degree of availability and reliability. The first design decision with respect to high-availability data servers relates to redundancy. Clearly, additional reliability can be obtained through the use of multiple processors. In any reasonable database facility, backup copies of the data are maintained. In terms of a data server, the issue to be addressed is whether the backup copies should reside on the same processor as the current data or be located on another processor. The basic philosophy of local-area networking argues for multiple processors. There is, of course, an added cost associated with additional processors, but this is offset by the increased reliability, the declining cost of processors, and the potential for high-bandwidth communication.

If the decision to employ multiple processors is made, the next problem is organizing the data and data-management system on a multiprocessor data server so that high availability can be provided with reasonable efficiency. There exist two basic alternatives for distributing the control of data across a group of data servers. In a high availability environment, multiple copies of the data must be present. Thus it is the control of the data that is distributed.

1. One server contains the primary copy of the data while the other nodes maintain backup versions. All requests are initially directed to the master data server.
2. The control of data is partitioned among the data servers. Each server has primary responsibility for some portion of the data and backup responsibility for other pieces of the database.

A primary difference between these two general approaches is the requirement of the second philosophy for a distributed concurrency-control algorithm. Distributed concurrency control has proven to be among computer science's knottier problems. While several theoretical solutions have been proposed, the performance effects of the overhead introduced by these algorithms is still unknown.[3] Therefore, for reasons of simplicity and perhaps performance, a master data-server scheme will be presented here. In a local-area network with a large demand for database service requiring several data-server nodes, some combination of the strategies may be employed.

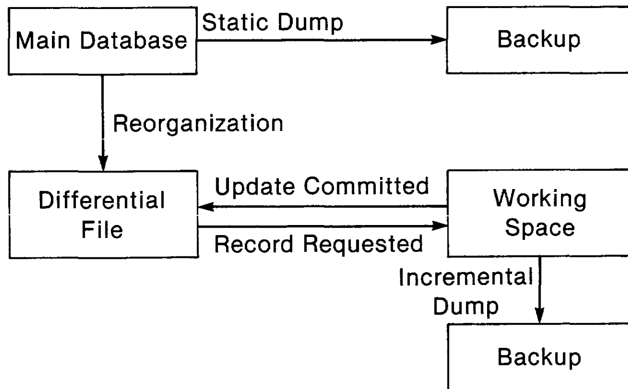


Figure 7—Fault-tolerant database architecture

The master data-server scheme proposed here is an extension of an architecture developed for a single-machine environment. Figure 7 presents the fault-tolerant database architecture defined by Maryanski.[18] This approach to reliability is based upon the concepts of differential files[23] and careful replacement.[28] Briefly, the differential-file approach involves maintaining all modifications to the database in a separate file, the differential file, and periodically merging the main database with the differential file. A distinct backup copy of the main database is also maintained. A filter is utilized to determine whether a record requested by a read operation is present in the differential file. Read requests for data that have not been changed since the last reorganization are directed to the main databases. The proportion of requests processed against the main database with respect to the differential file declines with time until a reorganization takes place. An algorithm for online reorganization is given by Maryanski.[18]

The technique of careful replacement avoids "updating in place." When an update occurs, the current value is copied into the working space, with the change taking place in that space. Pointers are adjusted after the new value is written. The careful replacement strategy is only employed when a record already in the differential file is updated. The use of a differential file effectively provides careful replacement for the first time a record is updated.

Initially assume that two processors are utilized to provide the database service. The goal of a strategy for partitioning the components of Figure 7 between the two processors is to maximize performance while maintaining fault tolerance. One partitioning formula (see Fig. 8) is to place the differential file on the master processor with the main database residing at the backup or slave processor. In this arrangement, all update requests are handled by the master processor. A read operation is executed by the master if it involves data in the differential file. Read requests for unaltered data are passed to the backup processor. Since the master node handles all updates, it also assumes responsibility for concurrency control. The distribution of work between the two processors is a function of the update frequency. Initially, all read requests are routed to the slave processor with the master handling only updates. In fact, the execution of an update of data not in the differential file in this configuration requires the

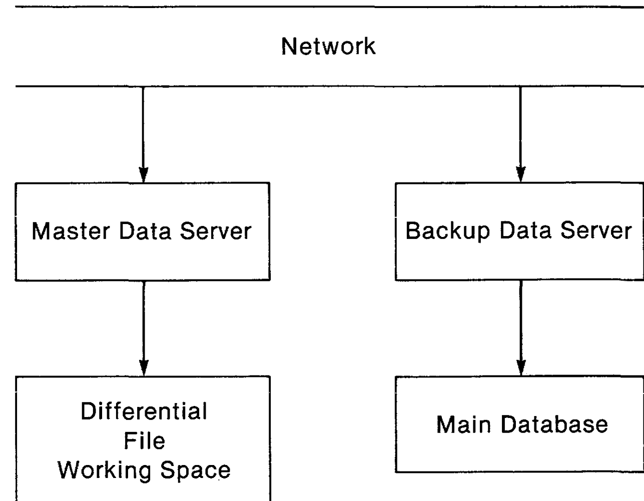


Figure 8—Redundant data servers, configuration 1

master to obtain data from the backup processor.

Another problem with this configuration is that utilization of the processors is completely data driven. The allocation of work between the processors may be unbalanced, depending on the frequency at which read requests access data in the differential file. Initially the backup processor will handle all read requests. As the differential file increases in size, the master will assume an increasingly large portion of the activity.

The configuration portrayed in Figure 9 alleviates the first problem of the previous configuration by eliminating the need for interprocessor communication when data not in the differential file are updated. Since a copy of the main database is maintained on the master processor as well as the backup, the transfer to the differential file from the main database can occur more quickly. The maintenance of a copy of the main database at both nodes presents the opportunity for balancing

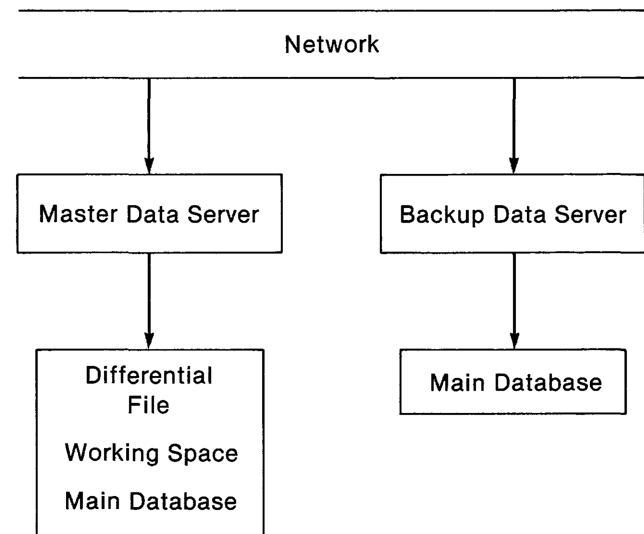


Figure 9—Redundant data servers, configuration 2

the utilization of the database processors. This modification to the strategy will only have an effect when the differential file is small. According to the 80–20 rule,[15] most of the database activity will be concentrated on a relatively small portion of the data. As the database reaches a stable size, the differential file will contain the most frequently accessed portion of the database. Thus, the master will handle a larger percentage of the database requests. However, since the differential file is a small subset of the entire database, searching the differential file is faster than searching the main database.

The final possibility for the partitioning of the components of the high-reliability data-server configurations involves replicating the differential file and the main database on both processors. This configuration, which is depicted in Figure 10, eliminates both of the deficiencies noted in the data-server organization of Figure 8. In addition, it provides the potential for more consistent balancing of processor utilization than the configuration given in Figure 9. Unfortunately, this latest architecture introduces the problem of maintaining consistent copies of the differential file. The possible mechanisms for keeping the differential files consistent are

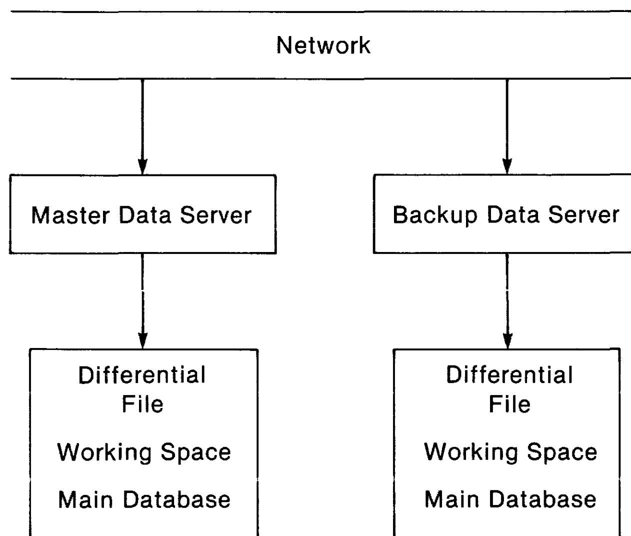


Figure 10—Redundant data servers, configuration 3

- Execute all updates at the master, then broadcast the changes to the backup processor. This approach limits load balancing since only retrievals can be executed by either processor. However, concurrency control remains the function of the master processor only.
- Use the load-balancing mechanism to distribute the updates between the data-server processors. In effect, the master-slave relationship no longer exists. While this approach has the potential for an equitable distribution of the workload among the processors, the problem of distributed concurrency control is introduced. The resulting configuration bears a strong resemblance to a fully redundant distributed data-management system.[2]

The first of these partitioning alternatives is simpler to implement and consequently requires less memory and time for the controlling software. Since the data server is dedicated to a single function, efficiency may not appear to be critical. However, the demand for this resource may be high; therefore the data server must execute as effectively as possible. The multiprocessor data-server configurations presented in this subsection, along with the algorithms given by Maryanski,[18] Severance and Lohman,[23] and Verhofstad,[27] will ensure a high degree of data availability. As explained by Maryanski[18] and Severance and Lohman,[23] the performance of systems based upon the differential-file concept is strongly dependent on update frequency and the presence of locality. The architectures described here make every effort to provide rapid execution of both retrieval and update commands.

Security

The issues of security in a local-area network's data server are akin to those in a backend database system. The basic question is "Does a network data server provide more security than a data manager on single, multipurpose processor?" The key factors involved in the evaluation of this question are

- Isolation
- Authorization

Isolation

The primary reason behind the claim that a backend database system provides enhanced security is that the backend processor is dedicated to the data-management function.[7] Therefore, no corrupt applications can circumvent the data manager and access the database files surreptitiously. In many data managers that are constructed upon a standard file system, it is possible to access the data files directly through the operating system without knowledge of the data manager. Both the backend database and data-server configurations eliminate this problem by isolating the data from all application programs. Thus the bogus user must be camouflaged in order to access the database in an unauthorized manner. This observation leads to the conclusion that the data isolation provided by the data server can only be useful for security if the problem of authorization is resolved.

Authorization

Database systems have always relied on an authorization mechanism to provide the most basic level of security. The dependence heightens in a local-area network in which the data manager and the application are not coresident on a single processor. In System R, authorization information is maintained as relations in the schema of Griffiths and Wade.[10] Designated users may grant and revoke access to portions of the database by issuing special commands that effect these relations. INGRES employs a similiar authorization strategy.[24] An authorization mechanism of this type can be adapted to the data-server environment with little alteration.

The relations containing authorization information reside at the data server. Commands to grant and revoke privileges emanate from the workstations to the data server.

The above scenario assumes that the underlying network control structure provides a foolproof method for the identification of users. The question whether a local-area network enhances or degrades the effectiveness of the authorization mechanism is an open one. Clearly, the data server is heavily dependent on the network's authorization mechanism. If the network's security mechanism does prevent penetrators from disguising themselves as valid users, the standard relational approach to authorization will be adequate for a data server.

Performance

Database performance is of course dependent upon such difficult-to-characterize variables as database structure and usage patterns. The two primary questions that arise in the evaluation of data-server performance are

- How does the performance of a data server on a local-area network compare with a single mainframe running both the data manager and applications?
- Which data server configuration gives the best performance for a particular environment?

Answering the first question is particularly difficult, since the performance of a data manager on a general-purpose mainframe is influenced by the nondatabase workload of the mainframe as well as by database requests. Overall disk utilization has a strong effect on the behavior of the data manager. In a local-area network with dedicated servers, the data manager will not have to contend with other subsystems for disk resources. However, the interprocessor communication of a local-area network will exact a performance penalty in terms of transmission delay and the overhead of communication software. The proper determination of the performance tradeoffs of a single mainframe system versus a local-area network of dedicated processors requires a detailed analysis. Simulation is the best approach to judging these tradeoffs for a particular application environment.

Queueing-model analysis could be supplied to the problem of comparing the relative performance of the data-server configurations pictured in Figures 8–10. These models must be parameterized to describe a particular environment, with the key factors being distribution of requests over the database, frequency of updates, and the communication delays, both line and software, between the data servers. Simulation techniques could also be applied here to project the performance of the various configurations in a range of environments. Another interesting variable in the analysis of data-server performance is the optimum number of processors in a data-server configuration. Here the tradeoff between concurrent operation and communication overhead must be balanced. Cost becomes an important factor in an analysis in which the number of processors is permitted to vary.

Experience with backend database systems leads us to expect a difference between performance projections obtained by simulations or analytical studies and the performance of actual prototypes.[17] The difference between theoretical and actual performance can be attributed to inaccurate modeling of the workload and the underestimation of the time spent executing communication software. Experimenters have had difficulty in creating large workloads with several applications concurrently accessing the database. Unless the database is heavily used, the benefits from concurrent operation of the host and backend will not be realized. The higher speed of the local-area network links certainly will yield better performance than a backend system using standard point-to-point communication. However, the utilization of a high-bandwidth communication medium does not automatically imply more efficient communication software in the workstations or the data server. The overabundance of communication software that was present in many backend database system prototypes must be reduced in local-area network data servers if they are to meet performance expectations.

CONCLUSION

The issues of data-server design raised here are the product of initial investigations into the problem. As mentioned earlier, experience with backend database systems has strongly influenced the definition of the problems as well as many of the proposed solutions. Since experience with local-area networks is limited, it is difficult to model accurately user workload characteristics in order to establish reasonable performance models for data servers. The key problems facing data server designers in the immediate future are

- The definition of reasonable high-level communication protocols that allow a high rate of actual data transfer between the application and the data server
- The synthesis of accurate performance models to permit the evaluation of architectural alternatives
- The understanding of the requirements for reliability, security, and high performance in particular environments.

Prototyping remains the best method for obtaining a true understanding of the character of experimental systems. This is certainly valid for local-area network data servers. The component technologies are available. The critical task is assembling the communication, data-management, and application systems into a well-integrated network data-service facility.

REFERENCES

1. Astrahan, M. M., et al., "System R: Relational Approach to Database Management," *ACM TODS,* 1, 2 June 1976, pp. 97–137.
2. Bernstein, P. A., et al., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)," *IEEE Transactions on Software Engineering,* SE-4 (1978), pp. 154–169.
3. Bernstein, P. A., and N. Goodman, "Fundamental Algorithms for Concurrency Control in Distributed Database Systems," CCA-80-5, Computer Corp. of America, Cambridge, Mass. (1980).
4. Berra, P. B., "Data Base Machines," *ACM SIGIR Newsletter* (Winter 1977), pp. 4–23.
5. Blasgen, M. W., et al., "System R: An Architectural Update," IBM Research Report, San Jose, Calif. (1979).

6. Canaday, R. E., et al., "A Back-End Computer for Data Base Management," *CACM,* 17, pp. 575–582.
7. Chamberlain, D. D., et al., "Support for Repetitive Transactions and Ad Hoc Queries in System R," *ACM TODS,* 6 (1981), pp. 70–94.
8. Delvecchio, B., and P. Penny, "The PHLOX Project: Three Database Management Systems for Microcomputers," *ACM SIGSMALL-SIGPC Symposium* (1980), pp. 173–178.
9. Germano, F., Jr., "DSEED: A Distributed CODASYL Prototype System," Ph.D. Dissertation, Wharton School, University of Pennsylvania (1980).
10. Griffiths, P. P., and B. W. Wade, "An Authorization Mechanism for a Relational Database System," *ACM TODS,* 1 (1976), pp. 242–255.
11. Kaisler, S., "The Agency Personal Information System," *ACM SIGSMALL-SIGPC Symposium* (1980), pp. 114–125.
12. Lind, L., "An Actual Implementation of a Distributed Database on a Minicomputer," in *State of the Art Report on Distributed Databases,* INFOTECH (1979), pp. 187–202.
13. Lowenthal, E. I., "A Survey—The Application of Data Base Management Computers in Distributed Systems," *VLDB* October 1977, pp. 85–92.
14. Maekawa, M., and S. Ishii, "An Extensible Distributed Data Base System," *AFIPS Proceedings of the National Computer Conference,* 47 (1978), pp. 813–822.
15. March, S. T., and D. G. Severance, "The Determination of Efficient Record Segments and Blocking Factors for Shared Data," *ACM TODS,* 2 (1977), pp. 279–296.
16. Maryanski, F. J., et al., "A Prototype Distributed DBMS," *Hawaii International Conference on Systems Science, Vol. 2* (1979), pp. 205–214.
17. Maryanski, F. J., "Backend Database Systems," *Computing Surveys* 12 (1) (1980), pp. 3–27.
18. Maryanski, F. J., and P. Charoenpong, "An Architecture for Fault Tolerance in Database Systems," *ACM Annual Conference,* October 1980, pp. 389–398.
19. Metcalfe, R. M., and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM,* 19 (1976), pp. 395–404.
20. Nolan, R. L., "Restructuring the Data Processing Organization for Data Resource Management," *IFIP Information Processing 77* (1977), pp. 261–265.
21. Passafiume, J. J., and J. Rivan, "Providing Network Data Services Using a Backend Data Base Machine," *IEEE COMPCON,* February 1980, pp. 251–262.
22. Rosenthal, R. S., "The Data Management Machine, A Classification," *Workshop on Computer Architecture for Non-Numeric Processing,* May 1977, pp. 35–39.
23. Severance, D. G., and G. M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM TODS,* 1 (1976), pp. 256–267.
24. Stonebraker, M., et al., "The Design and Implementation of INGRES," *ACM TODS,* 1 (1976), pp. 189–222.
25. Su, S. Y. W., et al., "A Microcomputer Network System for Managing Distributed Relational Databases," *VLDB,* September 1978, pp. 288–298.
26. Ting, P. D., and D. C. Tsichritzis, "A Micro-DBMS for a Distributed Data Base," *VLDB,* September 1978, pp. 200–206.
27. Verhofstad, J. S. M., "Recovery and Crash Resistance in a Filing System," *ACM SIGMOD Conference,* August 1977, pp. 158–167.
28. Verhofstad, J. S. M., "Recovery Techniques for Database Systems," *Computing Surveys,* 10 (1978), pp. 167–195.
29. Wilkes, M. V., and D. J. Wheeler, "The Cambridge Digital Communication Ring," *Local Area Communications Network Symposium,* May 1979.

SOCIAL AND ORGANIZATIONAL IMPLICATIONS

Acceptance criteria for computer security

by WILLIAM NEUGENT
System Development Corporation
McLean, VA

ABSTRACT

Acceptance criteria define the degree of quality required and identify areas to be examined in evaluating the degree of quality. Three categories of computer security acceptance criteria are proposed: functionality, performance, and development method. Each is further divided into sub-categories. Aids in formulating requirements and criteria are noted, including the use of organizational policies and risk analysis methods. Quantification is shown as a volatile tool, since numbers are often treated as single data points rather than as ranges. A set of principles is presented, to be followed in formulating acceptance criteria. Illustrative principles are as follows: (1) Get a good start, (2) make sure everyone understands, (3) distinguish *shall* from *should,* and (4) explain why. The acceptance determination process is discussed, a key point being that intermediate products must be approved. The value of acceptance criteria is in making the product better and the judgment easier.

INTRODUCTION

There are no people more surprised than computer users who first confront a system built "according to their requirements." Some might recognize their feelings as similar to those experienced upon meeting a blind date: It's much easier to recognize the unacceptable than to define the acceptable.

This problem is particularly common in the area of computer security. One reason for the problem is that there is little awareness of the role played by acceptance criteria. People think of requirements definition as solely a process of defining what capabilities they need. They forget that requirements definition must also consider how product acceptability will be determined. The criteria for this acceptability decision are called acceptance criteria.

This paper proposes a categorization for acceptance criteria, along with a set of principles that can make their definition easier. The goal is to help people define computer security requirements in ways that both improve the resultant product and simplify the determination of product acceptability. The paper is concerned only with the development of software and hardware, although it has applicability in other areas.

ACCEPTANCE CRITERIA

Acceptance criteria are specialized security requirements. They are specialized because they represent a perspective different from that of other security requirements. Whereas normal requirements are typically formulated in response to the question "What do we need?" acceptance criteria respond to the question "How will we decide whether the product is acceptable?" These clearly are overlapping sets, since products are usually defined as being acceptable if they meet needs. The problem is that if only the first question is asked, needs will often not be sufficiently defined. The role of acceptance criteria is to ensure that the requirements include sufficient definition of (1) "What degree of quality is required?" and (2) "What will be examined in evaluating the degree of quality?"

Thus acceptance criteria are measurable or demonstrable features of required security functions that characterize their desired quality. They serve as decision criteria used to determine whether a product complies with security requirements. They also guide developers who must decide how much quality to build into a product.

Decisions on quality are made at all levels of a development effort. This is true because every design level serves as a set of requirements for the level below it (Figure 1). Each level tells *what* must be done by the level below and describes *how* to implement the level above. The process of telling what is

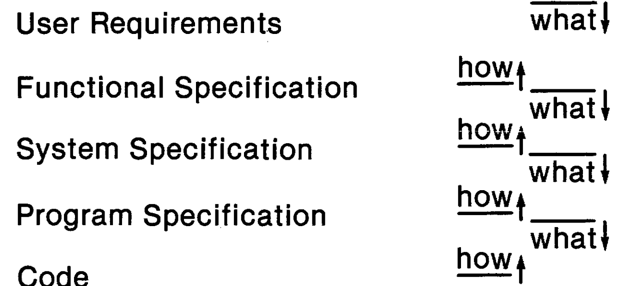


Figure 1—Relation between design levels

needed is a requirements definition process, and carries with it the need for some form of acceptance criteria. For example, criteria are needed at the functional specification level to help those defining the system specification decide how much redundancy to build in; and they are needed at the system specification level to help the program specifiers decide on the extent of error checking and handling.

This paper is concerned with acceptance criteria at the user requirements level. Since this level involves definition of the basic problem to be solved, criteria at this level are the most important. These criteria reside in the user requirements document. More specific forms of the criteria appear in many lower-level documents, the foremost being the functional specification and the acceptance test procedures.

From this general look at acceptance criteria, let us now examine them in more detail.

A STRUCTURE FOR ACCEPTANCE CRITERIA

The task in defining acceptance criteria is to describe control quality with an eye towards evaluation. To do this, one must first determine which characteristics of controls are the primary determinants of control quality. Three categories of determinants are proposed:

1. *Functionality*. What control functions are required?
2. *Performance*. What performance criteria must control functions achieve?
3. *Development method*. How must the system and controls be developed?

Each is discussed below.

Functionality

This category includes those things most often thought of as security requirements: control functions and data and sensitivity requirements.

Control functions

These include not only controls themselves, such as authentication and authorization functions, but also the functions required to manage and monitor them. Management includes such functions as changing authentication or authorization tables. This must consider issues such as who can change the tables and whether changes can be made dynamically. Monitoring includes the recording of security events such as errors and file accesses. The definition of monitoring functions must include what capabilities the system needs to measure its own performance. Examples are the measurement of resource use and response time. The operational system must be capable of reporting on measurements of its quality. Monitoring also encompasses the auditability of the function.

In defining control functions, there are several heuristic aids that can be used to help ensure completeness. These are similar to the who, what, why, when, and where of expository writing.

1. Control purposes: prevent, detect, or correct security exposures.
2. Violations thwarted by controls: disclosure, modification, denial of service, destruction.
3. Control functions: authorization (access control), authentication (identification), monitoring. This is sometimes expanded to include flow control, inference control, and encryption.

The definition of control functions can include factors such as when, how often, and how long (life span) the function is to be used; the level of detail at which it is performed; operating conditions and constraints; and the relationships among the functions. Additional and particularly important factors that need to be addressed for security are the amount of data sharing among users and the extent of user functional capabilities.[1] User acceptance and error tolerance needs must also be defined. User acceptance needs include guidance in what users will find acceptable so that they will not avoid or subvert controls. Error tolerance needs clarify how sophisticated or well trained the users are and how tolerant the system should be of errors they make.

Data and sensitivity requirements

Data requirements define system outputs, inputs, data elements, and data structures, along with estimated peak and average volumes and expected growth. Sensitivity requirements include sensitivity categorizations for data, software, hardware, and personnel positions. These categorizations must identify protection requirements for the various sensitivity categories. Consideration must be given to whether the processing or aggregation of data will change their sensitivity.

Performance

There is much more to control quality than proper functional operation. A number of qualitative acceptance criteria are listed here under the general heading *Performance.* They can be applied either to individual controls or to systems.

Availability

Define what proportion of time the system must be available to perform critical or full services. Availability incorporates many aspects of reliability, redundancy, and maintainability. It is often more important than accuracy. Some systems such as power supply systems require an availability of well over 99%. Others such as chemical process control systems and telephone network switching systems are almost as high. Security controls usually require higher availability than other portions of a system.

Survivability

Define how well the system must withstand major failures or natural disasters, where *withstand* includes the support of emergency operations during the failure, backup operations afterwards, and recovery actions to return to normal operation. Major failures are those more severe than the minor or transient failures associated with availability. Survivability and availability overlap where failures are irreparable, as in space systems and heart pacemakers.

Accuracy

Define how accurate controls must be. Accuracy encompasses the number, frequency, and significance of errors. Controls for which accuracy measures are especially applicable are identity verification techniques (e.g., using signature, voice) and communication-line error-handling techniques. Research in software quality metrics is applicable here.[2]

Penetration resistance

Define the needed resistance to the breaking or circumvention of controls, where *resistance* is the extent to which the system and controls must block or delay attacks. Cryptanalysis is an example of a technique for breaking a control (encryption). Creating and using a fraudulent system logon utility to discover passwords is an example of control circumvention. It is important to define who the penetrators might be: users, operators, application programmers, system programmers, managers, or external personnel. Keep in mind that most losses come from people performing their authorized tasks.

Response time

Define acceptable response times. Slow control response time can entice users to bypass the control. Examples of controls for which response time is critical are passwords (especially in distributed networks) and identity verification techniques. Response time can also be critical for control management, as in the dynamic modification of security ta-

bles. It is useful, in defining response requirements, to note the impact of varying levels of degradation.

Throughput

Define capacities that must be supported, where *capacity* includes the peak and average loading of such things as users and service requests. This can involve the use of performance ratios, such as total users versus response time.

Cost

Define acceptable costs to operate and maintain controls. Costs to build controls are also important and are included below under criteria for the development method. Getting the right amount of security means striking a balance between how much you might lose and how much you can afford to spend to reduce losses.

Development Method

Any craftsman knows the value of good tools. They make some tasks easier and other tasks possible. The same is true of the tools used to develop computer systems. The method used to develop controls is a major determinant of control quality. Important security aspects of the development are listed below.

Objectives

The importance of security relative to operational performance, cost, and other factors must be defined. This will help developers decide which objectives take precedence, should conflicts arise.

Project control

Define required management and technical project structures. An effective change control process is also required.

Resources

Define the amount of time and money available. These have a critical influence on control quality.

Development techniques

Define required design, programming, and test techniques. Design techniques can include adherence to security design principles (e.g., least privilege, complete mediation[3]) as well as generally desirable design principles such as simplicity and modularity. Formal specifications and verification might be required, as might the use of program description languages. Programming techniques can include the use of particular high-order languages and adherence to standards of good programming practice. Test techniques can include required test types or conditions and should usually include stress testing. Required measures of test coverage can be specified.[4] In addition to specifying the general use of particular techniques, one can also define areas in which assurance measures such as testing need to be emphasized. This gives different degrees of evaluation assurance for different controls.[5]

Documentation

Define what is required, when, and what it must contain. More acceptance decisions pertain to documents than to systems, since acceptance is often required of specifications and manuals. It is thus critical that required documentation contents be well defined. No definition can ensure quality, but a good definition can benefit the entire effort by improving the nature and timing of development decisions.

Developer trust

Define the amount of trust to be placed in developers. This may require investigations and clearances or even the use of particular people.

AIDS IN FORMULATING CRITERIA

Now that we have a clear view of what criteria are, we must consider how to go about formulating them. Some principles to assist in this process are given later in this paper. Here we consider useful aids.

Many aids are available in the form of laws,* standards,† and guidelines.‡ Whether or not these are mandatory in a specific case, they are useful guides. Another useful aid is an organizational or industry security policy that can be used as a guide to acceptable practice (and that ultimately might be used to establish objectives against which organizational performance can be measured, as in management by objectives). Unfortunately, few organizations have such a policy. To help fill this void, several professional organizations such as the Canadian Institute of Chartered Accountants[6] and the EDP Auditors Foundation[7] have defined general control objectives. Figure 2 is an example adapted from the Canadian work.

The control objectives are derived from types of loss that need to be controlled. There are several control technique objectives for each control objective and several techniques for each technique objective. The technique objectives must

*The Privacy Act, the Freedom of Information Act, the Foreign Corrupt Practices Act, and others, including many at the state level.

†The National Bureau of Standards (NBS) has issued several computer security standards, including the Data Encryption Standard (DES) and the DES Modes of Operation Standard. More are planned. NBS computer standards are mandatory for the executive branch of the federal government.

‡Many organizations, including NBS, have issued computer security guidelines. NBS guidelines address physical security, the Privacy Act, risk analysis, security of computer applications, contingency planning, and other areas. NBS guidelines that might be released shortly address the areas of user access authorization, security evaluation, security certification, and development of a security program.

CONTROL OBJECTIVE: To prevent or detect accidental errors occurring during processing by the EDP department.

CONTROL TECHNIQUE OBJECTIVE: There shall be some method to ensure correct files are mounted, switches are correctly set, and output files are properly allocated.

CONTROL TECHNIQUE: Computer files should be labeled internally and externally.

Figure 2—Structure for defining control requirements

be complied with, whereas the techniques serve as a menu of ways to achieve the technique objective. The two lower levels are particularly useful in defining security requirements. Unfortunately, the notion of quality, so crucial to acceptance criteria, is typically addressed only implicitly in such structures.

Risk analysis methods** are also helpful. They assist in deciding where to place controls and how much to spend on them. Most risk analysis methods require that estimates be made of how often each threat might occur and how much might be lost with each occurrence. They are therefore most reliable where there are good data on threat frequencies and losses, as is the case in the area of environmental risks as posed by fires and floods.

There are no good underlying data on hardware or software risks. Therefore the use of numeric frequencies and dollar values can be awkward in analyzing such risks. Despite this, risk analysis has value in the definition of requirements for hardware and software security. The value is simply in helping to systematically analyze risks to improve understanding and to make better judgments.

So risk analyses are useful. As was true for organizational or industry security policies, however, risk analyses are more helpful in identifying needed control functions than in identifying acceptance criteria associated with the controls.

QUANTIFICATION

The preceding discussion of risk analysis touched on the awkwardness of using numbers when good underlying data do not exist. Since quantification often plays an important role in both formulating and representing criteria, further discussion of this topic is required.

Numbers themselves are not the problem. They are helpful tools in making and recording decisions. The problem that arises is a people problem. People simply tend to be careless in their use of numbers.

The primary error is in treating numbers as single data points rather than as ranges. Numbers require some measure of their accuracy or flexibility to be associated with them. Without this, misunderstandings can result when the underlying judgments that give rise to numeric criteria are not as precise or inflexible as the numbers imply. The subjectivity involved in making decisions is well illustrated by the following description of how the National Aeronautics and Space Administration (NASA) defined the overall reliability acceptance criterion for the Apollo program.[8]

> When President Kennedy gave birth to Apollo, some of the best minds in the country were giving it one chance in ten of making it to the moon. But [NASA] engineers were choosing much better odds: 999 to 1. Caldwell Johnson, an engineer at the Manned Spacecraft Center in Houston, remembers how the odds were chosen.
>
> "The question of reliability came up," Johnson said not long ago. "Should 50 percent of the missions be successful? Should 9 out of 10 guys come back alive?
>
> "Or should it be 999 out of 1,000 guys? The cost of development is a function of reliability. If you can afford to lose half the spacecraft and half the men, you can build them [much] cheaper."
>
> While work on the Apollo design stopped in 1961, the question was debated for weeks. With nobody willing to make a decision, the engineering team turned to Robert Gilruth, then director of the Manned Spacecraft Center. Engineer Max Faget spoke up: "If we're successful half the time, that will be worth it."
>
> "No, that's too low," Gilruth said. "We can make 9 out of 10. Maybe 99 out of 100, lose one man out of 100 on lunar missions."
>
> "That's ridiculous," said Walt Williams, the director of the one-man Mercury. "Make it one in a million."
>
> "How about three nines?" Gilruth responded. "How about a reliability of 9–9–9?"
>
> And so it was.

Today NASA prefers to avoid the use of such numbers,[9] but this example reveals how a quantitative criterion can have a highly subjective derivation.

The need for a variance or confidence measure must be stressed, because the apparent clarity of numbers can create a sense of their legitimacy or authority that is difficult to dislodge. This authority can be improperly exploited through the intentional use of numbers to camouflage inadequate data and analysis. In such cases, numbers can serve more to promote judgments than to formulate them. The authority of numbers can also lead to unintentional misinterpretation, as when numbers are manipulated or used to make inferences that cannot be justified by the underlying data. This problem has led Touche Ross & Company to consider deleting the use of certain numbers from a control evaluation method that they have developed.[10] Touche Ross will replace the numbers with letters that, though serving the same purpose, are less susceptible to misinterpretation.

So numbers are useful but volatile tools. The challenge in using them is to assess the quality of the supporting data, accommodate this quality in the analysis, and reflect the resultant variance or confidence in the product.

An issue related to the subject of quantification is briefly mentioned here. That is the lack of a security metric. Despite much effort, there are no general metrics suitable for common use today by which security levels can be defined or

---

**Many methods are becoming available for analysis of computer security risks. Examples include an NBS guideline (FIPS PUB 65), System Development Corporation's "Risk Assessment Methodology" (developed for the Navy), Pansophic's PANRISK™, and the Fuzzy Risk Analyzer by Dr. Lance Hoffman of George Washington University.

measured—no inches, pounds, or degrees for representing security quality. The reason for this is that the widely differing types of security needs do not lend themselves to straightforward representation as general levels. There might be particular situations in which it is possible to define specific levels. In general, however, no useful metric exists to simplify the definition of acceptance criteria.

PRINCIPLES

Having examined what acceptance criteria are and what aids are useful in their formulation, let us now look in detail at the process of formulating criteria. This discussion is presented as a set of principles—rules of thumb to keep in mind when formulating criteria. These principles cannot guarantee success, but they might help avert failure.

1. Get a good start. Acceptance criteria are important guides for development. Their definition requires highly experienced people. They must also be defined early. In too many cases, criteria are derived during testing, when it is too late to influence development. Users and analysts must think about criteria at the start. For example, they can include with requirements a list of tests based on "impossible" things that might happen, but that might not occur to designers or programmers. Such tests, which might be included as formal acceptance tests, can thus provide early guidance.
2. Make sure everyone understands. Clarity is crucial. The criteria are agreements or, in many cases, contracts. They must be reviewed and approved by all parties, ideally before any contract is signed. To illustrate the importance of clarity, consider the criterion that an identity verification technique improperly reject no more than 1% of claimed identities that are in fact valid. This is a useful criterion, as long as everyone agrees that it includes only random errors, not intentional penetration attempts. In this and many other situations (e.g., denials of service), attacks by a penetrator can skew statistics.
3. Distinguish *shall* from *should*. Some things are more important or more achievable than others. The terms *shall* (or *must*), *should*, and *may* are often used to classify needs, reflecting whether they are mandatory, optional but recommended, or highly optional. Whenever used, their meaning must be precisely defined.
4. Explain why. Often the purpose to be served by requirements or criteria is not clear. It is difficult to understand the implications of criteria without knowing the reasons behind them. Hierarchies of control objectives, control technique objectives, and control techniques, as shown in Figure 2, are good for this purpose, as are narrative justifications.
5. Include measurement conditions. Surrenders can be unconditional; criteria cannot. Acceptance criteria must either indicate the conditions under which measurements will be made or reserve the right of the user to set the conditions at the time of evaluation. If the user understands the system well, the former approach is preferable; if not, the latter approach must be taken. Conditions can be complex, including system states, number and types of users and activities, points of measurement, factors counted, and so forth.

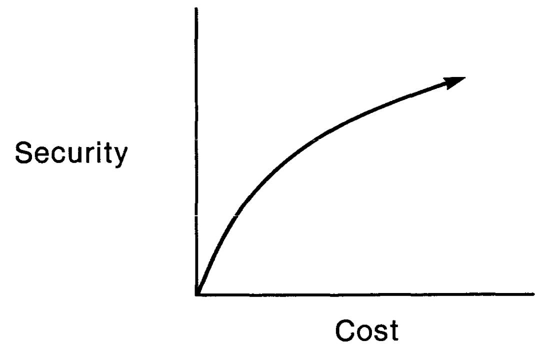


Figure 3—Costs for security

6. Remember that almost may be good enough. In computer security, almost is often all that is possible: 99% accuracy may be twice as expensive as 98%. Security costs tend to follow the curve shown in Figure 3.
7. Use numbers judiciously. Quantitative criteria must be founded on reliably measurable data and must reflect their accuracy and flexibility.
8. Do not let the criteria become the specification. A system built to pass a precise set of tests might do nothing else. Several types of criteria are needed to prevent optimization for one type.
9. If you cannot define the acceptable, define the unacceptable. An example from the NASA Voyager program was that "no single failure shall cause the loss of all data return from more than one science instrument or the loss of more than 50 percent of the engineering data."[11]
10. Do not ask for the impossible. It is commonly stated and accepted that absolute security is not achievable. This is true. There are, for example, no absolute defenses against human subversion, human error, or hardware failure. It is not meaningful, then, to say "the system must prevent data disclosure" when it is impossible, even with unlimited resources, to build a system that will absolutely ensure this prevention. Such needs must be phrased as objectives rather than as requirements.
11. Do not go overboard. It is possible to do too much. For example, measures of penetrability can vary, depending on penetrator costs, collusion, degree of access, system state, likelihood of detection, types and extent of loss, and other factors. Attempts to capture this detail might instead founder in it. It can be preferable to say, "The objective is that no users of Application A be permitted to gain control of the operating system."

ACCEPTANCE DETERMINATION PROCESS

We have now examined what acceptance criteria are and have looked at rules to follow in their formulation. To complete this

discussion, we must finally consider the acceptance determination process as a whole. Successful completion of the process is, after all, a condition for acceptance. Three points are stressed.

First, intermediate products must be subject to user approval. The development process is itself a system, requiring feedback and control. An initial set of criteria and a final decision are not enough. Once specifications are approved, compliance with them also becomes an acceptance criterion.

Second, it is a conflict of interest for developers to evaluate their own products. If quality is critical, it must be assessed independently, perhaps with the help of third-party expertise. For example, operating system vendors have been known to imply that their products can resist penetration by application programmers. This is rarely true. Customers who accept the word of the vendors might thus be unaware of vulnerabilities in systems they are using.

Finally, there can be no decision without choices. It is relatively easy to cancel a project when it is small and in its early stages. There is a threshold, however, beyond which many people seem to feel they cannot *afford* to stop. This tendency has made many casinos wealthy.

The Department of Defense sometimes avoids this cancellation dilemma by sponsoring the competitive development of specifications or even prototype products and funding the winner to continue on to implementation or production. Short of this, other options are available. The most common are to withhold acceptance pending completion of corrections or to accept conditionally, on the basis of an explicit agreement on which corrections will be made, when, and at whose expense. Many types of operational restrictions are also possible. Examples follow.

1. Adding procedural security controls
2. Restricting the system to the processing of only nonsensitive or minimally sensitive data
3. Restricting users to only those with approved access to all data being processed or to those with a sufficient clearance, based on an investigation
4. Restricting use of the system to noncritical situations where errors or failures are less severe
5. Removing dialup access
6. Removing especially vulnerable functions or components

CONCLUSION

Most users who define their own computer requirements neglect security. Security officers often consider it a success just to get computer security on the agenda. As should be clear from this paper, making the agenda is not enough. Thought must be given not only to needed security functions, but also to the required quality of those functions. Acceptance criteria for computer security are both necessary and achievable. Perhaps users can be motivated to act on this if they are reminded that good security defenses pay for themselves.[12]

In closing, two caveats are warranted. First, people and systems, no matter how well meaning, will never be perfect. Therefore, although criteria must be as precise as possible, decisions based on noncompliance require an enlightened blend of both toughness and tolerance. Reasonable people with flexible acceptance criteria and a spirit of cooperation will fare better than people with rigid criteria and a spirit of confrontation. Second, acceptance criteria by themselves are not sufficient to ensure successful development. No set of criteria can anticipate everything or supplant the need for later judgment. No criteria can offset inadequate developmental resources. Their value is in making the product better and the judgment easier. This is no small value.

ACKNOWLEDGMENTS

Much of this paper is adapted from work funded by the National Bureau of Standards Institute for Computer Sciences and Technology under U.S. Department of Commerce contract NB80SBCA0323.

The author is indebted to Dr. Charles Eldridge of System Development Corporation, Mr. John Gilligan of the Defense Communications Agency, Dr. Stephen Morse of MRJ, Inc., and Mrs. Zella Ruthberg of the National Bureau of Standards for their insights.

REFERENCES

1. "Secure-System Evaluation." In *1979 Summer Study on Air Force Computer Security*, 18 June to 13 July 1979, Cambridge, Massachusetts. The Charles Stark Draper Laboratory, Inc., 1979, p. 84.
2. Bowen, John B. "Are Current Approaches Sufficient for Measuring Software Quality?" *Proceedings of the Software Quality and Assurance Workshop*, 7, nos. 3–4 (1978). ACM Special Interest Group on Measurement and Evaluation.
3. Saltzer, J. H., and M. D. Schroeder. "The Protection of Information in Computer Systems." *Proceedings of the IEEE*, 63 (1975), pp. 1278–1308.
4. Miller, Edward F., Jr. "Tutorial, Program Testing Techniques." *Computer Software and Applications Conference 1977*. Software Research Associates, 1977.
5. Konigsford, William L. "Developing Standards for Operating System Security." *Computer Security Journal*, Spring 1981, p. 49.
6. Computer Control Guidelines, The Canadian Institute of Chartered Accountants, 1970. The figure was adapted from pages 46–49.
7. Control Objectives—1980, EDP Auditors Foundation for Education and Research, 1980.
8. O'Toole, Thomas, and Jim Schefter. "The Bumpy Road That Led Man To The Moon," *The Washington Post*, 15 July 1979.
9. Williams, Walter C. "Lessons From NASA." *IEEE Spectrum*, Vol. 18, No. 10, October 1981.
10. Davis, Keagle, Touche Ross & Co., private communication, 1980.
11. Williams, "Lessons From NASA," p. 80 (ref. 9).
12. IBM Corp. "Establishing a Data Processing Security Program." *Computer Security Manual*, Computer Security Institute, 1980, p. 1.20.

Private sector needs for trusted/secure computer systems*

by REIN TURN
California State University
Northridge, CA

ABSTRACT

Computer systems that have been subjected to formal verification of correctness of their access control mechanisms and that can provide multilevel security are called trusted systems. Their prototypes are now being developed under government programs and, to a much lesser scale, as a part of vendors' in-house research and development. While the need for trusted systems in national defense applications is well known, the need for trusted systems in private sector's business and industrial applications has been largely unexplored. This paper identifies several generic types of needs and incentives for the use of trusted systems, such as maintaining management control, complying with regulatory requirements, protecting computer representations of assets and resources, assuring safety and integrity, realizing certain operational economies, and enhancing marketing advantage or public image. It then examines the private sector's aspects of these generic needs, as well as disincentives that may surface. The paper concludes with an assessment of the prospects for commercial availability of trusted systems and the vendors' incentives for developing and marketing these systems.

*This paper is based on the Report R-2811-DR&E, *Trusted Computer Systems: Needs and Incentives for Use in Government and the Private Sector,* June 1981, which the author prepared for The Rand Corporation as a consultant. It was sponsored by the Office of the Undersecretary of Defense Research and Engineering.

INTRODUCTION

Computer systems are a necessity in the functioning of a modern, industrialized society. They are used by business and industry, and by government agencies, in a wide variety of applications, including financial transactions, research and product design, control of manufacturing processes, record keeping, long-term planning, and the general support of daily operations. Users expect these systems to have integrity and security, that is, correct functioning of programs, correct data values, and assurance that there have been no unauthorized access, modification or disclosure. In other words, users expect their computer systems to be trustworthy.

In a broad sense, a computer system consists of equipment, the operating system and application programs, data files or data bases, data communication networks, facilities, personnel, and users. Threats to system integrity and security may emanate from any of these subsystems, inadvertently or by deliberate design. A variety of protection techniques have been developed to counter these threats. For example, effective techniques exist for controlling physical access to computer systems. It has been much more difficult, however, to implement effective access controls within multi-user, resource-sharing computer systems where sensitive information is processed concurrently with other processing tasks, and where the trustworthiness of all users has not been established.

Within the computer, the access control function is implemented in the operating system programs, supported by various hardware mechanisms. However, for various design and implementation reasons, no existing operating system is fully secure—unauthorized users can surreptitiously disable or bypass the access control features of any current system. One solution to this problem would be the rigorous use of formal specification and certification techniques to prove that an operating system fully implements the desired security policy and that all access attempts are mediated, consistent with this policy. Computer operating system programs that have been subjected to formal certification of their access control features have been termed "trusted systems" by the Department of Defense Computer Security Initiative program.[1–4] It has defined a trusted system as one that "has sufficient hardware and software integrity to allow its use for simultaneous processing of multiple levels of classified and/or sensitive information."[1] In the present paper, the term is used more broadly to mean security-certified computer systems, and computer operating systems, in general.

The need for trusted systems in the national defense community's computer applications is well known. This need is less clear for civilian agencies of the federal government, for agencies of state and local governments, and especially for the private sector's business and industrial organizations. An analysis of the civilian governments' trusted system needs is given in Turn.[5] The present paper addresses the private sector's security needs and the potential of trusted systems to handle these needs. A projection of commercial availability of trusted systems is also made.

TRUSTED COMPUTER SYSTEMS

An important prerequisite for the development and certification of trusted systems is the precise identification of their access control mechanisms and the formulation of criteria for evaluation of the degree of protection they can provide. Many technical features can influence the overall integrity of operating system programs and the protection provided. Some features are essential regardless of the type of application or operating environment, but others are essential only in certain specific environments. Therefore, a particular system installed in one environment may provide sufficient security, while the same system in a different environment may be unacceptable. Accordingly, trusted systems can be categorized on the basis of their suitability for use in various operating environments. An important dimension in the categorization is the degree of certification of the systems' design and implementation.

Within the DoD Computer Security Initiative, seven protection levels have been developed.[1,6] They are cumulative in the sense that at each level the criteria for that level and all lower levels must be satisfied. When an operating system is evaluated, its rating will be determined by the highest protection level that is completely satisfied. The categorization criteria were defined so that systems rated at the lowest protection levels must meet certain security policy standards, even if the access control mechanisms are not judged sufficiently strong to counter certain subtle threats. For systems at higher protection levels, the emphasis is on evidence that the software, and ultimately the hardware, is correct. The protection levels are:

1. *Level 0: No protection.* A system that has no demonstrable ability to protect information.
2. *Level 1: Limited controlled sharing.* A system in which some attempt has been made to control access, but the controls are limited. For example, login authentication in a Level 1 system is based on passwords. (Most of the current operating systems provide Level 1 protection and are suitable for dedicated-mode operation.)
3. *Level 2: Extensive mandatory security.* A system in which minimal protection requirements are satisfied. Assurance is derived primarily from attention to protection during system design; extensive testing, including penetration testing, has been performed. Mechanisms in-

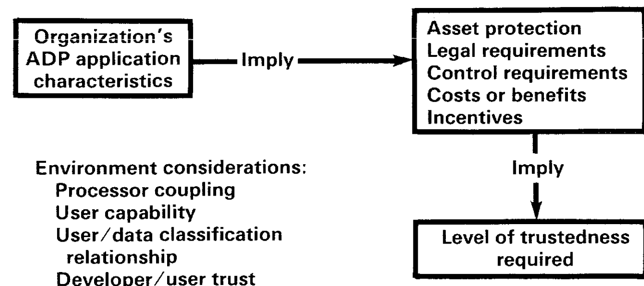
clude read and write authorization controls, virtual memory, and virtual machine architecture. (Some recent, mature operating systems provide Level 2 protection and are suitable for benign environments with need-to-know controls.)

4. *Level 3: Structured protection mechanism.* A system in which additional confidence is provided through methodical construction of protection-related software components and modern programming techniques, including a top-level specification. (The Multics operating system is an example of Level 3 protection in a benign environment with two levels of national-defense security, top secret and secret.)
5. *Level 4: Design correspondence.* A system whose protective-mechanism design has been formally specified and verified. Tests are generated from the formal design specifications, and operating system security kernels are used to implement complete mediation. (Examples are the KSOS-6, KSOS-11, and KVM/370 systems.) Level 4 systems are suitable for environments where limited user programming is permitted and three levels of security are allowed (e.g., top secret, secret, and confidential) in a reasonably benign environment.
6. *Level 5: Implementation correspondence.* A system whose software design and implementation have been formally specified and verified. Test cases are derived from the formal specifications. Extended provisions are provided for blocking covert information leakage paths. There are no examples of Level 5 systems at the present time. This protection level is suitable for environments where full user programming is permitted and three levels of security are allowed (top secret, secret, and confidential) in a reasonably benign environment.
7. *Level 6: Object code analysis.* In addition to meeting Level 5 requirements, Level 6 systems include object code analysis and object code to source code correctness proof, as well as additional hardware features such as extensive failure tolerance. There are no examples of Level 6 systems. Application environments would have full user programming and full multilevel security and would not have to be benign.

A specification of security mechanisms that would be required at each level is still being developed,[6] along with specifications of the administrative procedures that must be in place for the mechanisms to be effective, the threats that can be prevented, and the costs that arise. Given these specifications, the application areas and operational environments can be analyzed to determine protection levels required. Then the appropriate trusted systems can be selected from a security-evaluated set of trusted systems (i.e., from an Evaluated Products List that is expected to be developed by evaluating systems that are submitted by the industry). This process can be depicted graphically, as shown in Figures 1 and 2.

The protection environment of a trusted computer system is achieved through hardware and software access control mechanisms, including implementation in firmware or microcode, that control the sharing of information. These mechanisms, which comprise a trusted computer base (TCB) of the operating system, implement the "reference monitor"

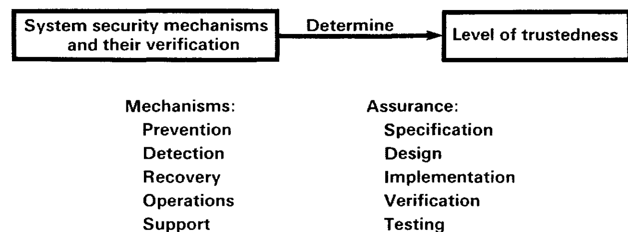
TRUSTED SYSTEM SELECTION



concept[7,8] for controlling when and how data are accessed.

In general, the TCB must enforce a given protection policy which describes the conditions under which information and system resources can be made available to users of the system. Protection policies specify precisely the rules for granting access to information in the various sensitivity categories and also cover the handling of such problems as unauthorized disclosure or modification of information, and damage to the system that can result in denial of service to authorized users.

TRUSTED SYSTEM EVALUATION PROCESS



Proof that a trusted system can enforce the desired protection policy requires formal approach to TCB design, implementation, and verification. Since the TCB contains all the protection-related mechanisms of the trusted system, proof of its correctness will imply that the rest of the operating system will also perform correctly with respect to the security policy. Ideally, protection policy and protection mechanisms should be treated separately in the system design, so that the TCB can be flexible and amenable to different environments and will not require rewriting or reverification to accommodate changes in policy. Details of trusted system and TCB design have been described elsewhere,[1–8] as have the security principles involved.[9]

The evaluation of industry-developed systems for possible inclusion in the Evaluated Products List for Defense Department purposes will be performed by a center established at the National Security Agency (NSA). It is envisioned that

this list will also be available for private sector users. The evaluation process will consist of four sequential steps, matched to the computer system development life-cycle development phases:[10]

1. *Preliminary evaluation:* Analysis of the TCB of a submitted system environment requiring trusted access controls. The purpose of the preliminary analysis is to determine whether the TCB has been sufficiently well designed and documented to warrant further evaluation. This step can be performed as soon as the proposed system has completed its concept formulation phase.
2. *Interactive evaluation:* An extension of the preliminary evaluation. This review will focus on whether the system satisfies the criteria for the level of protection specified in the preliminary evaluation. It will be based on a series of presentations by the developer and his documentation on the development phase of the system. The developer and the evaluation center will interact closely so as to assure that evaluation criteria are met and that discrepancies are found early in the development process.
3. *Final evaluation:* Analysis and testing of the production version of the proposed operating system to determine its strengths and weaknesses relative to the criteria for the specified level of protection. The developers will provide the evaluation center with a production-level system and the details of the test methods and procedures they have used to evaluate it. This step cannot be undertaken until the initial acceptance testing has been completed and the system is available for field testing. The final evaluation will determine the "actual" protection level of the system and where (and if) it is to be placed on the Evaluated Products List.
4. *Periodic reevaluation:* Required reevaluation of trusted systems on the Evaluated Products List that have been modified or enhanced. The evaluation center and the vendor will jointly analyze all system changes to evaluate the security-related aspects and to determine the extent of reevaluation needed.

A vendor who has a trusted operating system on the Evaluated Products List must maintain a master copy of the system in a physically secure facility, as well as assure the integrity of the copying process, so that users can verify that their copies are identical with the system that passed the evaluation.

The evaluation criteria to be used to determine eligibility for inclusion in the Evaluated Products List are still being developed.[6,10] Basically, they address two essential aspects of a trusted system: (1) completeness and adequacy of the protection policy that is implemented, and (2) verification of adequate implementation. In general, specific techniques or ways of implementation (i.e., hardware, software, or firmware) will not be prescribed.

TRUSTED SYSTEM NEEDS IN THE PRIVATE SECTOR

The definition of security and the requirements for it in the private sector tend to differ from those in the government. A recent analysis summarizes the need for computer security in private business and industry, especially in corporate management and operations systems based on ADP, as follows:[11]

1. Computers have become a basic resource in the operation of a business. The exception today is the *non-use* of computers in business function, not the use of computers. The end effect of this is an extensive business dependency on computer systems.
2. The concern in business and industry is with the consequences of interruptions of ADP support, including security failures: loss of production, loss of assets, loss of confidentiality, and loss of customer services, as examples.
3. The broad business incentive is the prevention of failure of the information-system portion of business systems. From the ADP management point of view, the security objective is to provide business managers with trusted information systems.
4. Security may be defined as knowing your business procedures, being confident of their correctness and completeness, and being sure that they are in place. In general, the DoD trusted system concepts are necessary but not sufficient for private-sector information security.

There are no standard security requirements or personnel clearance levels in the private sector, nor is there a consensus that these are needed. Various industry associations have developed security standards for their own members, however. The security function, like any other business function, is regarded by top management as an economic one—certain losses are viewed as tolerable if their prevention is too expensive or if loss prevention interferes excessively with business operations. However, because of federal or state laws, certain aspects of security are mandatory.

As in government agencies, trusted systems are needed in the private sector for the protection of assets and resources, regulatory compliance, maintenance of management control, and safety and integrity. Additional incentives may stem from potential improvements in operational economies, marketing advantages, and enhancement of public image.

Protection of Assets and Resources

Nearly every organization in the private sector that uses a computer system has automated its payroll, accounting, and inventory systems and is likely to use various MIS features as well—for financial planning, product development, market research, production scheduling, and so forth.

The assets that are stored in the computer system and thus exposed to security risks include financial records, information necessary for business functions, trade secrets, and marketing data. They are subject to internally perpetrated fraud, industrial espionage, and vengeful actions by disgruntled employees or others who may object violently to an organization's policies or activities. The data base on such computer-related crime in the private sector is thought to be substantial[12] (although most reports have been challenged as unverifiable).[13]

Information itself is an important resource in the operation and management of an organization. Management informa-

tion systems (MIS) are used both for decisionmaking in daily operation and for long-term organizational planning and guidance. The information used in strategic planning, along with the decisions themselves, is often very sensitive and must be protected against unauthorized access. In highly competitive industries, information on competitors' long-term development, production, and marketing plans is of great value, and the integrity of MIS data bases is extremely important. Moreover, the presence of inaccurate or deliberately falsified information can lead to decisions having very detrimental consequences. Trusted systems for MIS applications seem to be a necessity rather than a luxury.

Computer files containing sensitive information are subject to clandestine, unauthorized access by legitimate users of the system and, in some cases, by outsiders as well. If unauthorized actions can be detected, it is likely that their effects can be corrected, albeit sometimes at considerable expense and with substantial delays in the availability of correct information. If they are not detected, such actions can result in both the direct loss of assets or resources and the indirect losses that may ensue from operating without knowing that those assets or resources are missing or that the system has been tampered with. Numerous cases of such losses have occurred in the past; the problem is real, and it is serious.[12]

Trade secrets are another type of corporate asset. They are protected by law against unauthorized use by outsiders, provided they have been handled as secrets from the very beginning of their development. If computer systems are involved in the development of trade secrets, protection of programs and data is a requirement; at the very least, trusted systems could be implemented as a demonstration of concern over the security of the trade secrets being developed.

In general, managers tend to be quite skilled at providing adequate protection to manual accountings of assets and resources, using control techniques that have proven effective through years of use. However, these techniques are not directly transferable to the protection of computerized information. Moreover, high-level managers tend to be unfamiliar with protection techniques for computer systems (and with computers and data processing in general), and few have addressed the problem of protecting accountings of assets and resources maintained in such systems.

When certified trusted systems become available as off-the-shelf items, effective access controls can readily be provided. Trusted systems can augment the existing administrative controls and will eliminate the vulnerabilities of current computer systems. Moreover, individual organizations and agencies will not have to design their own protection mechanisms and will be spared the tasks of protection system verification and evaluation.

Regulatory Compliance

A sizable body of federal and state laws and regulations that affect automatic data-processing (ADP) systems and their management and control has evolved over the past several years. Collectively, these regulations

1. Prescribe secure processing and storage of certain categories of information (e.g., identifiable personal records on individuals).
2. Require assurances that full management control is exercised over ADP operations and use.
3. Require implementation of security techniques in ADP systems and facilities as indicated by risk assessments.

Compliance with such requirements may necessitate the use of trusted computer systems. For example, in the pharmaceutical industry, FDA regulations require accurate and controlled record-keeping. Trusted systems can provide the assurance that the required records are kept accurately.

Many suppliers of ADP services assure their customers (both external subscribers and internal users) that their data and programs are fully protected. Failure to provide that protection may lead to legal actions against them involving breach of contract. Trusted systems can provide the means to minimize protection failures and associated losses, thereby eliminating management vulnerability to breach-of-contract lawsuits or other claims of negligence and liability, including lawsuits filed by stockholders.

Legal admissibility of computer records as accurate representations of a corporation's financial status or business activity is an important consideration in modern business and industry. If records from a corporate computer system are not considered trustworthy by authorities, they may be ruled inadmissible, and the corporation may have to keep additional records using more expensive manual methods. Trusted computer systems may become a prerequisite for full legal acceptance of computerized accounting systems and reports.

The following federal record-keeping legislation affecting the private sector has been enacted or is pending in the Congress:

1. The Fair Credit Reporting Act of 1969 (15 U.S.C. 1687 *et seq.*) applies to organizations that collect, maintain, and make available for a fee creditworthiness information on individuals. The Act focuses on individual rights. Disclosure to clients is the credit bureaus' business; thus, prevention of unauthorized access is mainly intended to maintain data integrity (a requirement of the Act) and prevent data thefts. Bills to amend the Act and broaden its coverage to depository institutions (H.R. 1046) and to insurance carriers (H.R. 1047) are pending.
2. The Family Educational Rights and Privacy Act of 1974 (20 U.S.C. 1232g) applies to any educational institution that receives federal funds from the Department of Education. It grants certain privacy rights to students and their parents and restricts disclosure of educational records to third parties. Amendments to the Act are pending (H.R. 1048). Security provisions are needed in computer systems where student records are stored concurrently with other data, including student schedules, and in systems that are also used by students in course work. H.R. 1048 addresses the use of student records for research purposes and requires that "... adequate safeguards to protect the record or information be established and maintained by the recipient, including a program for removal or destruction of identifiers."

3. The Financial Privacy Act of 1980 (12 U.S.C. 3401) applies to banks and restricts access to depositors' bank transaction records by government agencies. Pending is a bill, H.R. 1046, which in part also addresses Electronic Funds Transfer Systems (EFTS) but includes no statements on EFTS security requirements. These systems are certain to be subject to future federal legislation.

The report of the Privacy Protection Study Commission[14] contains detailed data and specific recommendations regarding privacy protection and confidentiality of credit, financial, insurance, employment, medical, welfare, and educational records maintained by federal and state agencies and by organizations in the private sector.

Accountability requirements

The Federal Securities and Exchange Act of 1934 (15 U.S.C. 78) defines certain accounting requirements for publicly held corporations. These corporations are required to establish internal controls to safeguard assets against loss and to provide reliable financial records for internal use and for external reporting purposes. Similar requirements are established in state corporation codes. The internal control and auditing procedures implemented to comply with these statutes usually involve the following elements:

1. Competent, trustworthy personnel with clear lines of authority and responsibility.
2. Adequate segregation of duties.
3. Proper procedures for authorization.
4. Adequate documentation and records.
5. Proper procedures for record-keeping.
6. Physical control over assets and records.
7. Independent (internal) checks on performance.

In organizations that use ADP, these controls are applied to programs and data bases; to data acquisition, storage, and processing; to report generation; and to data communication software, hardware, and personnel. Development of effective controls and auditing procedures is still a difficult problem, but progress is being made.[15,16] The use of trusted computer systems promises considerable enhancement of control effectiveness.

The Federal Foreign Corrupt Practices Act of 1977 (P.L. 95–213) amends the Securities Exchange Act of 1934 by inserting Title I to strengthen the accounting and accountability requirements. Section 102 of Title I, Accounting Standards, states that:

(2) Every issuer which has a class of securities registered pursuant to section 12 of this title and every issuer which is required to file reports pursuant to section 15(d) of title shall—

(A) make and keep books, records, and accounts, which, in reasonable detail, accurately and fairly reflect the transitions and disposition of the assets of the issuer; and

(B) devise and maintain a system of internal accounting controls sufficient to provide reasonable assurance that—

(i) transactions are executed in accordance with management's general or specific authorization;

(ii) transactions are recorded as necessary to permit preparation of financial statements in conformity with generally accepted accounting principles or any other criteria applicable to such statements, and to maintain accountability for assets;

(iii) access to assets is permitted only in accordance with management's general or specific authorization; and

(iv) the recorded accountability for assets is compared with existing assets at reasonable intervals and appropriate action is taken with respect to any differences.

The impact of this Act on publicly held corporations is to require further strengthening of internal control and accountability along the lines described above. In 1980, the SEC proposed (and then withdrew) a set of rules[17] which discuss internal control and explain the notion of "reasonable assurance" as follows:

> The concept of reasonable, as opposed to absolute, assurance is incorporated in the proposed rules in recognition that it is not in the interest of shareholders for the cost of internal accounting control to exceed the benefits thereof. Such benefits, and in many cases such costs, are not likely to be precisely quantifiable. Therefore, many decisions on reasonable assurance will necessarily depend in part on estimates and judgments by management which are reasonable under the circumstances.

Improved internal control may bring about not only quantitative benefits, such as reduced exposure to theft of assets, but also qualitative benefits, including preservation of the good reputation of a company and its management.

International laws and regulations

Within the last 8 years, privacy and data protection laws have been enacted in several European countries and in Canada.[18] In addition, a convention on privacy protection is being ratified by the member countries of the Council of Europe,[19] and a set of voluntary privacy protection guidelines has been completed by the Organization for Economic Cooperation and Development (OECD),[20] which includes the United States, Japan, Canada, and Australia.

The foreign data protection laws and international agreements contain requirements for privacy protection, data confidentiality, and data security in international data transfers, especially when personal information on either natural or legal persons is involved. They affect so-called multinational corporations and the data-processing networks that provide services in Europe using U.S.-based computer systems.

OECD Guidelines Governing the Protection of Privacy and Transborder Flows of Personal Data, September 1980 (Annex, Part 2, sec. 11), state that "personal data should be protected by reasonable security safeguards against such risks

as loss or unauthorized access, destruction, use, modification or disclosure." The United States has voted to approve the OECD Guidelines and U.S. private-sector organizations that are affected have been urged to voluntarily abide by them.

Data-protection laws have been enacted in Austria, Canada, Denmark, France, Germany, Luxembourg, Norway, and Sweden and are pending in several other countries. The Austrian Data Protection Act (1978) applies to natural and legal persons. It requires that "the processor shall, having regard to economic feasibility and technical possibilities, introduce organizational, staff, technical and structural security measures. Such measures shall, having regard to the type of data and technical facilities and to the scale of processing, [ensure] that data are not unlawfully disclosed or brought to the knowledge of third parties and cannot be consulted, processed or disclosed by unauthorized persons."

The French Act on Data Processing, Data Files, and Individual Liberties (1978) states: "Any person processing personal data or ordering such processing shall thereby undertake, vis-a-vis the persons concerned, to see that all necessary precautions are taken to protect the data and in particular to prevent these from being distorted, damaged or disclosed to unauthorized third parties."

The Federal Data Protection Act of the Federal Republic of Germany (1977) spells out the security requirements in considerable detail:

> Where personal data are processed automatically, appropriate measures suited to the type of personal data to be protected shall be taken to ensure observance of the provisions of this Act:
>
> a. Unauthorized persons shall be refused admission to data processing facilities which process personal data that are restricted (admission control);
> b. Persons employed in the processing of personal data shall be prevented from removing storage media without authorization (leakage control);
> c. Unauthorized input into the memory and the unauthorized examination, modification or erasure of stored personal data shall be prevented (storage control);
> d. The use by unauthorized persons of data processing systems from which or into which personal data are disseminated by means of automatic equipment shall be prevented (use control);
> e. It shall be ensured that persons entitled to use a data processing system have access by means of automatic equipment only to the personal data to which they have a right of access (access control);
> f. It shall be ensured that it is possible to check and to establish to which bodies personal data can be disseminated by means of automatic equipment (dissemination control);
> g. It shall be ensured that it is possible to check and establish what personal data have been input in data processing systems, by whom and at what time (input control);
> h. It shall be ensured that personal data processed on behalf of other parties are processed strictly in accordance with the instructions of the principal (control of processing on behalf of other parties);
> i. It shall be ensured that data cannot be read, modified or erased without authorization during their dissemination or during transport of relevant storage media (transport control); and
> j. It shall be ensured that the internal organization of authorities of enterprises is suited to the particular requirements of data protection (organization control).

Nearly all the cited laws require the organizations that are affected to obtain some form of prior licensing or approval by data-protection authorities, who examine and evaluate the security features of the systems being examined. The use of trusted systems may satisfy many of the above requirements for internal access controls.

Management Control

Effective management control is required by various laws and regulations, but it should also be an organizational goal in its own right. An organization must have mechanisms in its structure, administrative procedures, and technical operations that minimize the potential for detrimental actions or events and that provide a high level of confidence that such events will be detected, if they do occur. With the advent and increasing use of ADP systems, many of the traditional means of implementing management control have become ineffective. They must be, and have begun to be, replaced by new means of control that take into account the environmental and functional changes brought about by the use of computers.[15,16]

It has been necessary to develop internal control and auditing techniques to assure accuracy and completeness of computer-based transaction processing, record maintenance, and reporting, and to provide access control and physical security to computer systems and data files. However, technical hardware and software measures are not sufficient to assure full management control by themselves. They must be supported by clear and consistently applied management procedures and, above all, they must have the full support of top-level management.

Verified trusted systems can provide the first part of the overall protection system—the technical mechanisms. Management actions and support must, of course, be provided by the organization itself.

The potential benefits of trusted systems are evident, yet certain tradeoffs must be acknowledged: Rigid control can stifle innovation and impair efficient use of computer resources. Moreover, beyond required regulatory compliance, the risk of losses must be weighed against economic pressures on an organization. At times, the risk of loss due to imperfect controls may be small when compared to the risk of not being able to function at all. For example, retail stores will always be subject to a certain amount of shoplifting, since attempts to eliminate it totally—for example, by manually searching every exiting customer—would be excessively costly as well as unacceptable to the customers.

Assurance of Safety and Integrity

A potential collateral benefit of trusted operating systems development and use relates to system safety and integrity. Computer systems are being increasingly used to implement control over systems that operate in "real time" and whose operation must be monitored continuously. Computers are used to detect deviations from correct operation and to apply remedial measures immediately in process control in oil refineries and steel mills, automated assembly lines, rapid tran-

sit systems and air traffic, onboard applications in aircraft, national-defense systems, and many other applications. All of these real-time control situations are characterized by the possibility of disastrous consequences in the event of failure. Thus it is imperative that steps be taken to assure the safety of personnel, facilities, and equipment. Hardware and software components in such systems must be highly reliable, and their integrity must be assured throughout their life cycles.

High levels of hardware reliability can be achieved by use of various failure-tolerant design techniques. Reliability and continued integrity of software and data bases are more difficult to achieve. Software-engineering techniques can increase software reliability considerably, but full assurance of the reliability of a software module will require formal verification of design correctness and of subsequent implementation as computer object code. Such verification is exceedingly difficult for all but very small programs.

The trusted system development effort is based on full verification of operating system program modules and their interfaces. Since real-time control systems are essentially special-purpose operating systems, the concepts of trusted systems and their verification are fully applicable. Thus, real-time control systems should be viewed and developed as trusted systems.

The same is true for computer-aided design (CAD) systems whose products affect public safety, and for other computer models that are used to make important design or policy decisions. Construction engineering, nuclear power generation, aerospace engineering, and econometric modeling are examples. In such systems it is necessary to assure that the integrity of the design programs or models is not compromised accidentally or intentionally. Trusted operating systems or their design and verification techniques can be used here to provide that assurance.

Operational Economies

The use of trusted systems may lead to certain economies in the operation of a computer facility. Whether or not such economies actually materialize will, of course, depend on specific situations and contexts. For example, an existing system that has been acceptably secure by virtue of the disruptive technique of scheduling sensitive processing to be done at times when the system is closed to other users could be replaced by a trusted system, which does not require periodic scheduling. Or trusted systems could obviate the need for extensive personnel background investigations (the so-called system-high clearance level mode of operation). In these cases, trusted systems can eliminate special security efforts and thereby reduce associated expenses. In other situations, the cost benefits of using a trusted system may be less clear-cut.

In general, the following operational economies may be achievable through the use of trusted systems:

1. Reduced duplication of data, equipment, or personnel required when dedicated systems are used for processing sensitive data.
2. Reduced requirements for personnel clearances or security procedures (e.g., stringent control of physical access to terminals).
3. Reduced insurance premiums for business risk or liability or management liability (in the private sector).
4. Reduced downtime losses and recovery costs that should result from the better design and implementation of trusted systems.
5. Elimination of the need for a dedicated processing shift (e.g., in private-sector organizations that use proprietary data extensively or that have trade secrets to safeguard).
6. Reduced need for highly trained operators and support personnel to apply access controls and other controls.

Marketing Incentives

Certain organizations that provide services involving their customers' assets—e.g., banks, savings and loan associations, and other financial institutions—must be able to insure that those assets are properly handled and safeguarded. These organizations, particularly the financial institutions, tend to be very competitive and therefore are always seeking new approaches that will give them competitive advantages. The use of trusted systems to reduce risks to customers' assets may give an institution a more favorable image even though all competing institutions may already provide adequate protection of assets through insurance coverage.

Clients of organizations such as computer service bureaus are concerned with the security of the data or programs they submit for processing or storage and are likely to choose an organization that can provide better safeguards, such as the use of trusted systems.

Finally, all government and private-sector organizations that interact with the public are concerned about their image. Government agencies make special efforts to explain their missions and to publicize the necessity or benefits of their operations. Private-sector organizations likewise expend resources to emphasize their concerns for the welfare of the public. A particularly important area of public concern is the collection and maintenance of personal information about individuals. The safeguards that are implemented to assure that personal information remains confidential and is not accessed or disseminated by unauthorized individuals can greatly enhance an organization's public image. Two or three corporations have already purchased advertising space in national magazines to emphasize their concern and to describe the approaches they are taking to assure confidentiality of personal data. Recent public-opinion surveys, including the Harris Poll directed by Alan Westin in 1978,[21] have demonstrated that there are strong public sentiments in favor of assuring confidentiality of personal information commensurate with individual privacy rights.

Public-image concerns also arise in the area of asset and resource protection. No organization wants publicity resulting from fraud or other substantial losses or because of having being victimized by a computer crime. The use of trusted systems can reduce the possibility of adverse publicity by reducing the probability of occurrence of such events.

Other Considerations

The all-important issues in the private sector are business economics, ability to remain competitive in the marketplace, and making a return on stockholders' investments. Acquisition of computer systems or any other equipment is a business decision made in view of these issues. Thus, there is a natural tendency in the private sector to view the acquisition of a trusted computer system also as a purely dollars-and-cents question. In addition, a trusted system either must be shown to be cost-effective in comparison with other security techniques that could achieve a comparable level of protection or it must provide additional benefits that justify any additional cost. The impact of trusted system implementation on the performance of the corporate computer system and any requirements to modify existing applications software of data bases are of particular concern. It is not surprising, therefore, that some private-sector ADP system managers are skeptical about the need for trusted systems in their organizations and about the cost-benefit aspects of trusted systems.

However, as discussed extensively in this section, the acquisition of a trusted system is not just a matter of business economics. There are numerous important considerations—protection of assets and resources, regulatory compliance, public image, management prudency—that are likely to be the deciding factors. In more technical terms, it is certainly true that while a trusted computer system can reduce the need for the more conventional security techniques, it does not eliminate entirely the need for physical, administrative, personnel, or communication security techniques, nor can it fully handle a denial-of-service threat by authorized users or system personnel. But it provides a trustworthy base for implementing sets of discretionary protection mechanisms for monitoring denial-of-service threats and generating tamperproof evidential audit-trail records.

Concerns over performance or efficiency losses resulting from the use of trusted systems and the need to justify what some people see as a deliberate reduction of service are valid and understandable, as are concerns over possible large-scale conversions of applications software or data bases. Many performance concerns are based on a single, experimental data point—the preliminary results in KSOS-11 development where emulation of the UNIX operating system on PDP-11 computers resulted in a substantial performance slowdown on the untuned system. However, in KSOS-6, implementation of the UNIX emulator on the SCOMP hardware (a specially modified Honeywell Level 6 minicomputer) has resulted in a much smaller performance slowdown. There is a general trend in the development of applications software to include features that are also very useful for implementing performance-efficient trusted systems; thus, performance loss is likely to be a much lesser problem in the future, and there is some reason to believe that the performance costs of trusted systems will be negligible, or even nonexistent, as the experience base grows.

Any sizable application software or data base conversions that are required by the acquisition of a trusted system are certainly cause for concern. However, if the TCB is compatible with an existing (untrusted) operating system, software that ran under the operating system can be run on the trusted operating system with only minimal conversion. Such compatibility was a design goal for the KVM/370 and KSOS and has been successfully demonstrated with the KVM system. If the TCB and the existing operating system are not compatible, the conversion could be a significant part of the price of having a trusted application.

In general, concerns over performance losses or software conversion have been expressed whenever important innovations have been introduced, including the present-generation operating systems, with their resource-sharing capabilities. However, as vendors have become more experienced, many of the perceived problems have either failed to materialize or have been solved effectively and efficiently. It is highly likely that this will be the case in trusted systems development as well.

PROSPECTS FOR AVAILABILITY OF TRUSTED SYSTEMS

Whether or not trusted operating systems will be widely available within the next 3 to 5 years in a sufficient range of protection levels and hardware bases to satisfy the needs of government and the private sector will depend on the computer industry's perception of the size of the potential marketplace; the costs of developing trusted systems, having them certified, and maintaining them (in the sense that software is maintained now); and the profits that manufacturers and distributors can expect to make.

The Potential Market

System software vendors are primarily concerned with whether or not a proposed system will have a sufficiently large market to justify its development costs. We must make a distinction here between (1) large vendors of computer systems and associated software and (2) software houses. The trusted system development decision is much more complex for large vendors, because they must consider the issue of compatibility of new software with existing applications, systems, and equipment, and that of maintaining compatibility in the future. The introduction of a new operating system (or a family of operating systems) is more difficult to justify for an organization whose existing software base is large. Software houses, on the other hand, are likely to have less stringent requirements for maintaining across-the-board compatibility with their existing products, but they are more dependent on vendors' changes of hardware bases.

This paper is not attempting to determine the quantitative marketing opportunities for trusted systems, but it can make some qualitative observations. First, there seems to be a consensus among vendors that the government (federal, state, and local) does not in itself constitute a sufficiently large market to support the development, certification, and maintenance of trusted systems. However, the market is not insignificant, and if future RFPs require the use of trusted systems, vendors may be compelled to produce them in order to remain viable in the government marketplace.

Second, most of the market for trusted systems in the civil-

ian agencies of the government and in the private sector will probably be for Level 1 through Level 4 systems (as defined earlier in this paper). Some organizations in which asset protection or safety is very important may need higher-level systems. Some of the latter will be developed to satisfy the DoD needs once the state of the art permits such development, regardless of other markets. In all cases, the important aspect is that these systems have been *certified* to provide the specified level of protection.

Production Of Trusted Systems

The trusted operating system concept involves the establishment of a completely separate, or virtual, environment within the computer for each concurrent user. Most existing operating systems are modifications of earlier batch-processing designs, updated to accommodate multiprogramming and time-sharing. In these systems, the mechanisms to accomplish shared concurrent use are scattered throughout the operating system, making the TCB very complex, and are not completely isolated from users. Thus, these operating systems are not currently secure, and it may be infeasible to upgrade them to the point where they become demonstrably secure (or reach a higher level on the Evaluated Products List).

Computer vendors recognize the implications of this problem—it affects much more than just the security aspects of a system—and they are gradually developing system architectures that can create fully isolated processing environments. But the need to maintain compatibility with existing systems weighs importantly against drastic changes, as does a certain inertia of designers who are familiar with existing architectures and design principles and therefore are reluctant to change. Users' system programming staffs have the same sort of inertia, and as a result, the few operating systems that do use virtual machine concepts, Honeywell's MULTICS and IBM's VM/370, have until quite recently found relatively little use even though they have been available for ten years.

The compatibility problem is not entirely untractable, however. The virtual machine concept permits each user to run his own operating system under the control of the virtual machine monitor (VMM), which is essentially transparent to users. This generality will necessarily result in some loss of performance, but the loss can be compensated by the faster hardware that is becoming available. The increasingly clear-cut needs for the capabilities that only trusted systems can provide will lead to greater user acceptance—and demand—which should provide a strong incentive for vendors to incorporate the necessary architectures in their new operating systems. For example, the VM/370 maintains many compatibilities for users of IBM systems.

Three trusted-system development prototypes, sponsored by the DoD Computer Security Initiative, are now being tested to demonstrate the feasibility of design, implementation, verification, and operational use of trusted systems.[1] As the marketplace for trusted systems expands, uncertainties such as the compatibility question will be resolved and vendors should begin to incorporate trusted systems technology into their new product lines.

Evaluation and Certification

Certifiable trusted systems are difficult to develop unless the criteria for certification are unambiguous, reasonable, and clearly stated. Three sets of factors have thus far been identified by the Initiative program:[6, 22] protection policy, mechanisms, and assurance. While the policy may vary from user to user, the mechanisms and assurance tend to employ a common set of technical approaches. The protection policies, too, form a hierarchy, since the goals of each are the same, and differences are those of degree only. A basic trusted system framework can be "customized" to satisfy the user's protection policy by applying appropriate mechanisms. Certification will then be based on the embedded policy.

A protection policy specifies the conditions under which information and computer resources may be shared, typically placing controls on the disclosure and modification of information. Given a clear and concise formal statement of protection goals, it will be possible to evaluate whether or not the system meets those goals.

To be effective, the hardware and software mechanisms that enforce the protection policy must be complete and verifiable. They must also be self-protecting against unauthorized actions or inadvertent intrusions by users or their programs. Operating systems that are poorly designed will not only fail to confine users to their authorized actions and data, but they may also undermine discretionary protection mechanisms provided by the users in applications programs. Thus, evaluation must necessarily concentrate on operating systems and their related software and hardware controls, particularly those relating to detection and prevention of policy violations, recovery from errors, and system operations and maintenance.

Absolute assurance that implemented mechanisms can provide the protection that they promise will never be possible, but steps can be taken in the design, implementation, and validation phases of a trusted system's development to raise confidence to a high level. Such techniques include top-down design, structured programming, and other techniques collectively known as modern programming practices.

Trusted Systems Support

Computer software tends to be a complex commodity in that throughout its life cycle, numerous changes are inevitably made to meet modified design requirements, to increase efficiency, and to improve user interfaces. These changes are usually the vendor's responsibility, and an operating system typically moves through a series of releases. Any new release of a trusted system that involves changes of critical portions of the TCB will require reexamination of the previous certification. In such cases, if the system is to keep its rating, the Evaluation Center and the vendor must jointly analyze the changes and the extent of recertification needed. Clearly, it is important for the vendor to minimize changes in the TCB (but changes in the non-security-relevant portions of the system can be made as needed, since they will not involve recertification).

CONCLUDING REMARKS

The DoD Computer Security Initiative program is now demonstrating the feasibility of designing and implementing trusted computer systems that can provide high levels of protection to data, programs, and processing in certain constrained operational environments. Ultimately, full, multilevel secure operation will be possible in unconstrained operational environments. But, of course, physical, administrative, personnel, and communications security will always be required.

A federal-government-level trusted system Evaluation Center that will establish an Evaluated Products List of trusted systems has been established at NSA. Before an Evaluated Products List can be created, however, the need for trusted systems in the government and the private sector must be sufficiently great for system vendors to perceive a marketplace beyond national-defense requirements that warrants submission of their systems for evaluation.

Trusted systems can contribute effectively to the solution of the growing problems of protection of assets and resources, compliance with laws and regulations, assurance of safety and integrity, and implementation of full management control. In addition, trusted systems may provide operational economies, marketing advantages, and public-image enhancement. They are needed in a variety of applications that constitute a market that should be of considerable interest to vendors and that should strongly encourage participation in trusted system development efforts. The use of trusted systems is in the interest of private business and industry, as well as of public policy, public safety, and national welfare.

As the use of trusted systems in business and industry expands, such use is likely to become a standard of good practice for management control and protection of computer-based assets, resources, or customer data. Failure to employ trusted systems could eventually be construed by insurance carriers, external auditors, regulatory agencies, customers, contract granters, and stockholders as management practice that is not prudent and reasonable.

ACKNOWLEDGMENTS

The author wishes to acknowledge substantial contributions made to the Rand Corporation report,[5] on which this paper is based, by Gary Martins and Willis H. Ware of the Rand Corporation, Marvin Schaefer of the System Development Corporation, and Stephen T. Walker of the Department of Defense. Other contributors were Walter L. Anderson of the General Accounting Office; Gary Bearden of Tenneco, Inc.; Louise G. Becker of Congressional Research Service; Sheila L. Brand of HHS; Dennis Branstad of NBS; Harry DeMaio of IBM; Edwin Jacks of General Motors; Seymour Jeffery, formerly of NBS, now at TRW; Paul Karger of Digital Equipment; Theodore Lee of Sperry-Univac; Joseph Millen and Grace Nibaldi of Mitre; Donn Parker of SRI International; Kenneth Pollock of the GAO; Oliver Smoot of CBEMA; and Edward Springer of OMB.

REFERENCES

1. Walker, S. T. *DoD Computer Security Initiative: A Status Report and R&D Plan,* Information Systems Directorate, Assistant Secretary of Defense, Communications, Command, Control, and Intelligence, Department of the Defense, Washington, D.C., March 1981.
2. Walker, S. T. "The Advent of Trusted Operating Systems." *AFIPS, Proceedings of the National Computer Conference* (Vol. 49), 1980, pp. 655–665.
3. *Proceedings of the Seminar on the DoD Computer Security Initiative Program,* National Bureau of Standards, Gaithersburg, Md., July 17–18, 1979.
4. *Proceedings of the Second Seminar on the DoD Computer Security Initiative Program,* National Bureau of Standards, Gaithersburg, Md., January 15–17, 1980.
5. Turn, R. *Trusted Computer Systems: Needs and Incentives for Use in the Government and the Private Sector.* R-2811-DR&E, The Rand Corporation, Santa Monica, California, June 1981.
6. Nibaldi, G. M. *Proposed Technical Evaluation Criteria for Trusted Computer Systems.* M79-225, The MITRE Corporation, Bedford, Massachusetts, October 25, 1979.
7. Schell, R. R. "Security Kernel Design Methodology." *Proceedings of the Seminar on the DoD Initiative Program.* National Bureau of Standards, Gaithersburg, Md., July 17–19, 1979, pp. E-1–E-21.
8. Anderson, J. P. *Computer Security Technology Planning Study.* ESD-TR-73-51 USAF Electronics System Division, Hanscom AFB, Massachusetts, October 1972.
9. Ware, W. H. (ed.) *Security Controls for Computer Systems.* R-609, The Rand Corporation, Santa Monica, California, February 1970.
10. Trotter, E. T., and P. S. Tasker. *Industry Trusted Computer System Evaluation Process.* MTR-3931, The MITRE Corporation, Bedford, Massachusetts, May 1, 1980.
11. Jacks, E. L. "Computer Security Interest in the Private Sector." *Proceedings of the Second Seminar on the DoD Computer Security Initiative Program.* National Bureau of Standards, Gaithersburg, Md., January 15–17, 1980, pp. E-1–E-10.
12. Parker, D. B. *Crime by Computer.* New York: Scribner, 1976.
13. Taber, J. K. *On Computer Crime (Senate Bill S.240), Computer/Law Journal,* 1 (1979), pp. 517–543.
14. *Personal Privacy in an Information Society.* Report of the Privacy Protection Study Commission, Washington, DC, July 1977.
15. Russell, S. H., T. S. Eason, and J. M. Fitzgerald. *System Auditability and Control Study: Data Processing Control Practices Report.* SRI International for the Institute of Internal Auditors, Altamonte Springs, Florida, 1977.
16. Ruder, B., T. S. Eason, M. E. See, and S. H. Russell. *Systems Auditability and Control Study: Data Processing Audit Practices Report.* SRI International for the Institute of Internal Auditors, Altamonte Springs, Florida, 1977.
17. *Statement of Management on Internal Accounting Controls.* Securities and Exchange Commission, *Federal Register,* 45, (1980), p. 40134ff.
18. Turn, R. (ed.) *Transborder Data Flows: Concerns in Privacy Protection and Free Flow of Information.* Arlington, Virginia: AFIPS Press, 1979.
19. *Convention on Protection of Individuals with Regard to Automatic Processing of Personal Data.* Council of Europe, Strassbourg, France, January 28, 1981.
20. *Guidelines on the Protection of Privacy and Transborder Flows of Personal Data.* Paris: OECD, 1980.
21. *The Dimensions of Privacy.* Sentry Insurance Company, Stevens Point, Wisconsin, 1978.
22. Nibaldi, G. H. *Specification of A Trusted Computer Base (TCB).* M79-28, The MITRE Corporation, Bedford, Massachusetts, November 30, 1979.

Impacts of information system vulnerabilities on society

by LANCE J. HOFFMAN
The George Washington University
Washington, D.C.

ABSTRACT

After briefly presenting examples of potential vulnerabilities in computer systems which society relies on, the concept of risk analysis is introduced and applied to a simplified model for a nation's financial system. A sampling of specific technical safeguards to ameliorate the risk in this (or any) computer system is then given. The paper concludes with examples of questions to be asked before committing to any new technological system.

INTEGRATED INFORMATION SYSTEMS

As computer applications in many countries have become increasingly sophisticated, the operators of these systems have become increasingly concerned about the unforeseen consequences of total reliance on them and have become more and more aware of the vulnerability of these systems to failures.

Society's dependence on the uninterrupted operation of large information systems has been increasing.[13] As these systems have grown larger, more complex, and more centralized, the potential societal loss from their failure or misuse has also increased. When society becomes highly dependent on the reliable functioning of a single integrated technological system or small collections of such systems, the possibility of a "domino-like" collapse of several of the individual connected units could be disastrous. The failure of the Northeast power grid in 1965, which blacked out much of that section of the United States (including all of New York City), is an example.

Other risks may be in the form of economic losses, such as the failure of an automated check-clearing system or a national automated securities market. Banks or brokerage houses could be severely damaged in a matter of minutes, long before it was discovered that the system had failed. The potential victims would be the owners of the failed system, individuals with accounts, correspondent organizations, and (if the failure cascaded through other institutions) all of society.

Still other risks may entail social costs such as would occur if a centralized criminal history system or an electronic funds transfer (EFT) payment system were misused by a government or a private firm to exert undue control over individuals.[2]

Large, centralized systems are not all bad; in addition to cost and functional advantages, a large nationally networked information system may provide greater availability than a single-site system by supplying instant backup to nodes that fail. However, there may also be a greater risk that the entire system will collapse if an unlikely or unexpected combination of events occurs. While the likelihood of this may be very low, the consequences might be extremely damaging in terms of physical, economic, and/or social costs.

How can we estimate the potential risks of such catastrophic events? To do this, we turn to the field of risk analysis—the study of estimating loss from adverse events.

RISK ANALYSIS

Risk analysis often is used to estimate the exposure of system components to various threats and to allocate resources to provide both technical and nontechnical safeguards against various threats.[3,15,19,20,21]

We prefer inexact tree-based risk analysis methods[4,14,16] which use linguistic terms such as "high," "medium," and "low." One can use fuzzy set theory[17] or other means to implement these and to allow the estimator to provide a degree of confidence for the estimates; this can then be taken into account when risk is computed. The computed results are then less prone to instill false confidence, because they are modified to reflect estimator confidence in the inputs and given in words rather than in numbers. We also believe that linguistic estimates are more useful for subjective risk analyses which deal with human error or social risk. There are myriad problems in obtaining numeric input data.[15] Nonnumeric input data are easier to obtain and allow the use of structural risk analysis techniques that force the disclosure of assumptions in the input data.

Tree-based risk analysis has recently been made available in computer systems, making sensitivity analysis (asking "what if" questions) much easier and inexpensive than with many other systems.

A TREE-STRUCTURED MODEL FOR ONE POTENTIALLY VULNERABLE SYSTEM—A DEPOSITORY FINANCIAL SYSTEM

A simple model which treats separately paper, "old electronic" (wire transfers, etc.), and electronic funds transfer transactions of a depository system is shown in Figure 1. Figure 2 treats that depository system as a subsystem of a financial system involving a number of financial institutions. Figure 3 treats the world as a system composed of several interrelated financial systems.

Obviously the model here is simplified; in particular, the

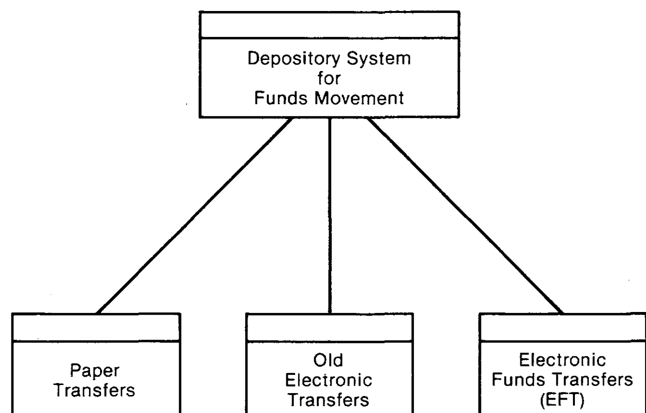


Figure 1—Tree structure of a depository system for funds movement

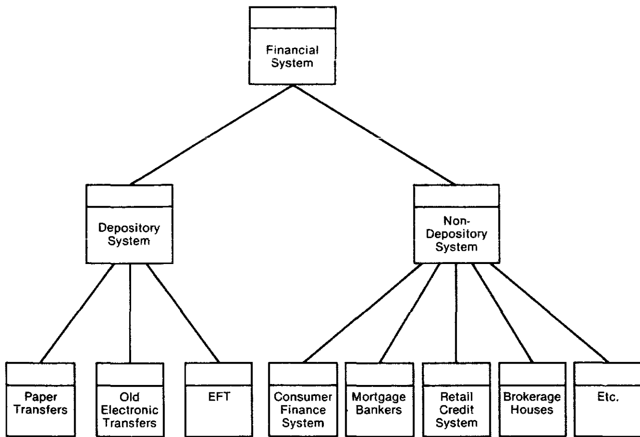


Figure 2—Possible tree structure of a financial system

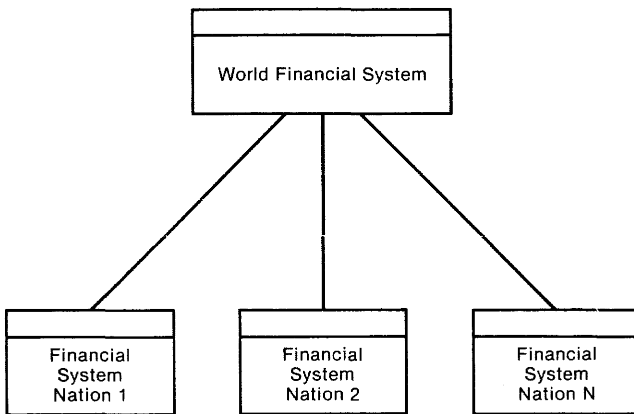


Figure 3—Possible tree structure of the world financial system

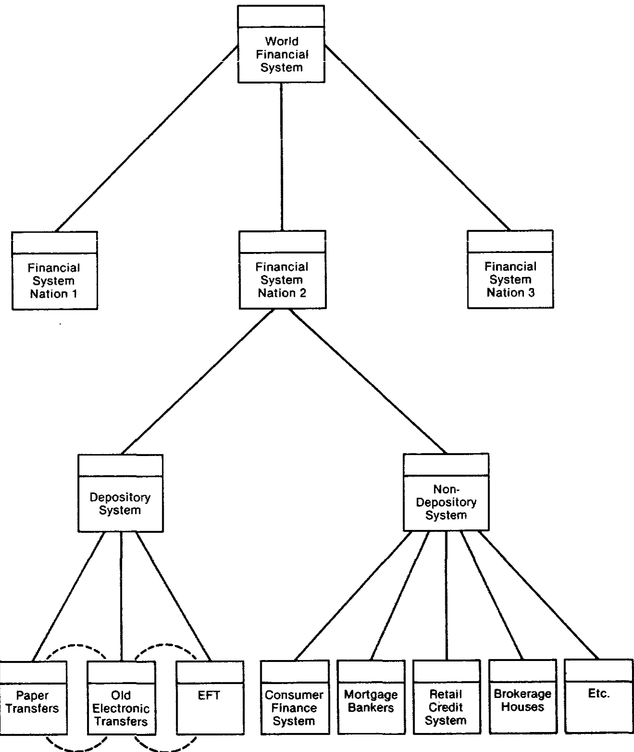


Figure 4—Detailed tree structure for simplified model of world financial system. (Broken lines are not part of the tree structure; they indicate significant interdependencies.)

dotted lines in Figure 4 indicate just some of the large flows of information which constitute threats and which must be taken into account in any risk analysis but which are not considered here. In addition, we do not address here (although a larger tree could) non-operational problems such as potential limited blockade against the import of spare parts for computers.[13] Nevertheless, we hope our simplified model bears enough resemblance to reality to highlight basic concepts and that other tree-based models can be suggested for societal institutions so that tree-based risk analysis can be used in assessing the vulnerabilities of these institutions.

EXAMPLE RISK ANALYSES OF THE FINANCIAL SYSTEM

We now consider the financial system model outlined above and use it to evaluate vulnerability risks for a "low risk" scenario and a "high risk" scenario. The estimates given are meant to be illustrative only—each reader can supply his or her own. We are only attempting to give examples of how risk analysis can be used to evaluate the potential vulnerabilities of financial systems; we make no claims about the validity of the specific data or computations used.

Low Risk Scenario

Figure 5 illustrates the low risk scenario. Here we consider the relative weights (or importance) of the financial systems of Nation 1, Nation 2, and Nation 3, respectively, as LOW, HIGH, and LOW (with respect to the risk associated with the higher level World Financial System).

At the bottom level of Figure 5, the likelihood of failure and severity of failure for the various subsystems are input data, and risks computed from these are shown in the top half of some boxes in Figure 5. Risks for subsystems can also be estimated; these are shown in *italics* in the top half of some boxes in Figure 5. Given this data, which is estimated or computed from input data, risks for higher level subsystems can be *derived.*[4,14] Derived risks for (the higher level subsystems of) Figure 5 are shown in CAPITAL LETTERS in the top half of some boxes in Figure 5. Details on possible risk computation methods are given in Schmucker;[14] in essence, we consider some of these analogous to weighted sums, but using nonnumeric data.

In this "low risk" scenario, the systems with the highest associated risks are the financial systems of Nation 1 and Nation 3.

Higher Risk Scenario

Risks computed using slightly different data (the weights of the subsystems of the Depository System of Nation 2 have

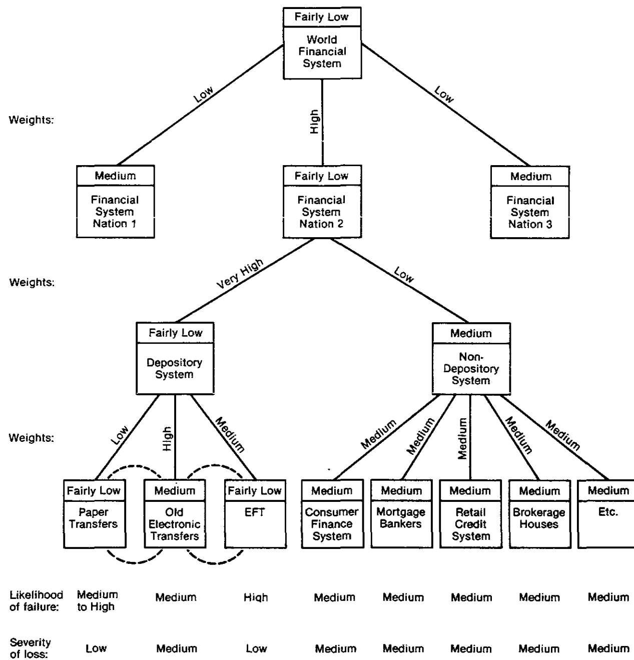


Figure 5—Financial system low risk scenario

been juggled, and the severity of a loss due to the EFT subsystem has been changed from "LOW" to "HIGH") are shown in Figure 6. Given this scenario, the system with the highest associated risk is now the financial system of Nation 2, and within it, the greatest contributing subsystem is the EFT subsystem.

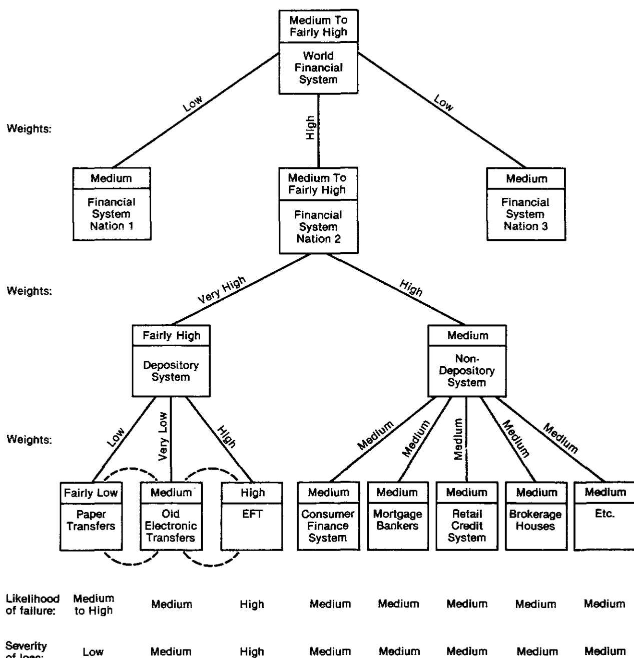


Figure 6—Financial system higher risk scenario

INTERCONNECTION OF INFORMATION SYSTEMS AND JOINT VULNERABILITY

Due to the complexities of real-world systems, the subsystems are interdependent (as shown by the dotted lines in Figures 4, 5, and 6). A well-publicized failure in one EFT system may subtly influence vulnerabilities in other EFT systems (for example, by inviting attempts against similar systems). If, in turn, there is a relatively large number of successes of exploiting vulnerabilities in a number of EFT systems, this may affect the depository system of a country. Our simple model above does not reflect many relationships; for example, it does not show the effects of an increase of vulnerabilities in one country on vulnerabilities in another country.

We've shown in the preceding pages how one can structure a simplified model of the world financial system as a tree and separately evaluate the risks to it and to each nation's financial subsystem. We can also insert our world financial system of Figure 4 into a larger World Information System (Figure 7). After weights and estimates of severity and likelihood are added to Figure 7, we can perform a sensitivity analysis to determine the one or two most critical subsystems. Then we can focus our efforts to protect the overall system by strengthening the most critical subsystems using non-technical and technical safeguard methods.

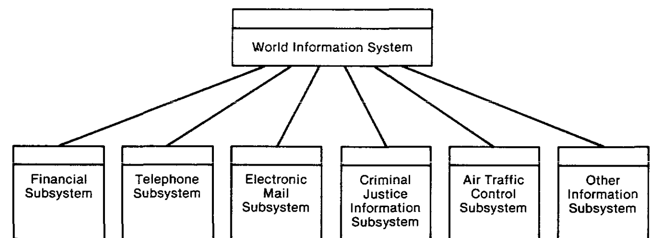


Figure 7—World information system

AMELIORATING SECURITY PROBLEMS IN COMPUTER SYSTEMS

We can fairly easily determine the greatest contributors of risk and then assign safeguards at the appropriate places in a system's tree. Applying appropriate countermeasures (for example, adding additional procedural and administrative controls) might transform the risk indicators of Figure 6 to those of Figure 5.

In addition, one can always use countermeasures which are external to the system under discussion: additional legal safeguards, an improved physical security program to deter threats, additional training of auditors to detect variances from the norm, etc. Preventive action can take a number of forms in computer systems, but the reader should keep in mind that the majority of computer security problems are not resolved by technical security measures such as those we are discussing here. Rather, they are resolved by the more traditional non-technical solutions: physical security, legal and administrative restrictions, and policies and procedures. These

non-technical methods are generally well known to the managerial and audit community.[6] Because technological security measures are less commonly known, we shall here mention a couple of the most useful. Many more are described in much more detail.[7,8,18]

SOME TECHNOLOGICAL COMPUTER SECURITY MEASURES

Computer systems can identify and authenticate users by various methods. The use of an identification number or a user's name together with a computer account number and a password is typical. Computer systems can also control access by levels. Top managers, for example, can be granted access to more information than lower level personnel. Computer systems can also differentiate based on category (need to know); users having only a need for financial data can be prevented from access to medical data. When attempts to access information in computer systems are denied, a log of information about the attempt can be recorded for later analysis.

Another method of protection is the use of cryptography for storage and transmission of information. Plaintext information is encrypted by a transmitting device. The encrypted message, or ciphertext, is transmitted across a relatively insecure medium (such as a communication line) to a receiving device. Only if the human sender and the human receiver possess the same (secret) digital key is the information intelligible to the receiver (Figure 8). Otherwise, only unintelligible ciphertext is seen.

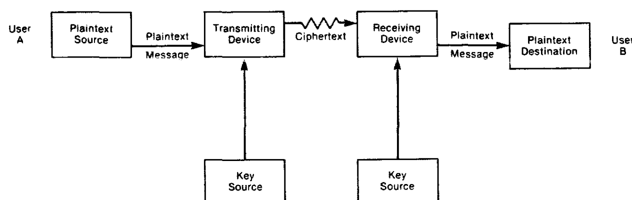


Figure 8—Cryptographic system

One possible cipher is now a U.S. national standard,[9] mandated for use by all civilian applications of the U.S. Government which require cryptography and supplied as hardware by several U.S. manufacturers. Another possible cipher technique is public key cryptography.[10] It also has the capability of transmitting signatures in such a way that it is very difficult for someone to send a message and then later disavow it.

DESIGN PRINCIPLES FOR COMPUTER SECURITY

There are some generally accepted principles to follow in the secure design of computer systems. Two of the most important are early critical review and user acceptability.

1. Early critical review
   Exposing the frailties of a system to a number of bright people during the planning stages will facilitate corrections before the system is implemented; it is better to have bugs discovered early by invited commentators rather than later on by intruders.
2. User acceptability
   The human interface must be simple, natural, and easy to use; if not, users will bypass it, thus rendering the security system ineffective.

SOME WORRISOME QUESTIONS

System designers and policy makers should ask some questions before committing themselves to any new technological systems. The answers will not always be easy to find, but the questions will not go away. The following are examples of such questions:

- How can we balance the risks society may encounter against the benefits it may receive, under conditions where failure rates appear to be relatively low, but potential losses may be high?
- How can society retain the option to end its dependence on a particular technology if it has unanticipated, undesirable effects? How can it avoid becoming "locked in" to the use of an information system forever?
- How can society provide alternatives to persons choosing not to use or live under a given system?

ACKNOWLEDGMENTS

Discussions with Fred W. Weingarten of the U.S. Congress, Office of Technology Assessment (OTA) and Information Policy, Inc., and with Zalman Shavell of OTA were very helpful in developing some of these concepts. Hans Peter Gassman provided some of the initial inspiration for writing this by inviting an earlier paper on the topic for a workshop sponsored by the Organization for Economic Cooperation and Development. Kathy Hoffman ruthlessly pointed out unnecessary jargon. However, any responsibility for conceptual or other errors is solely that of the author.

REFERENCES

1. Sweden, Ministry of Defence, *The Vulnerability of the Computerized Society—Considerations and Proposals,* 1980.
2. U.S. Congress, Office of Technology Assessment, *Computer-Based National Information Systems; Technology and Public Policy Issues,* September 1981.
3. U.S. National Bureau of Standards, *Guidelines for Automatic Data Processing Risk Analysis,* FIPS PUB 65, August 1979.
4. Hoffman, L. J. and L. A. Neitzel. "Inexact Analysis of Risk." *Computer Security Journal,* Vol. 1, No. 1 (Spring 1981), Hudson, Mass.
5. Meadows, C. *Identifying the Greatest Contributor to Risk in a Tree Model.* The George Washington University, Department of Electrical Engineering and Computer Science, Research Report GWU-EECS-81-09, May 1981.
6. Martin, James. *Security, Accuracy, and Privacy in Computer Systems,* Englewood Cliffs, N.J.: Prentice-Hall, 1973.
7. Hoffman, L. J. *Modern Methods for Computer Security and Privacy,* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1977.
8. Fernandez, E. G., R. C. Summers, and C. Wood. *Database Security and Integrity,* Reading, Mass.: Addison-Wesley Publishing Co., 1981.
9. U.S. National Bureau of Standards, *Data Encryption Standard* FIPS PUB 46, Washington, D.C. 1977.

10. Hellman, M. E. "An Overview of Public Key Cryptography." *IEEE Communications Society Magazine,* November 1978, pp. 24–32.
11. Denning, D. E. "Are Statistical Data Bases Secure?" *AFIPS Conference Proceedings of the National Computer Conference* (Vol. 47) 1978, pp. 525–530.
12. Denning, D. E., and J. Schlörer. "A Fast Procedure for Finding a Tracker in a Statistical Database." *ACM Transactions on Database Systems* (Vol. 5) No. 1, March 1980, pp. 88–102.
13. Tengelin, U. "The Vulnerability of the Computerised Society." in *Proceedings of the High Level Conference on Information, Computer, and Communications Policies for the 80's,* Organization for Economic Cooperation and Development, Paris, 1980, pp. 359–377.
14. Schmucker, K. "Fuzzy Sets, Natural Language Computations and Risk Analysis." The George Washington University, Department of Electrical Engineering and Computer Science, Research Report GWU-EECS-81-10, May 1981.
15. Okrent, D. (Ed.). "Risk-Benefit Methodology and Application: Some Papers Presented at the Engineering Foundation Workshop." September 22–26, 1975, Asilomar, Calif., School of Engineering and Applied Science, University of California, Los Angeles. Report UCLA-ENG-7598 (NTIS-PB-261920), December 1975.
16. Hoffman, L. J. "Tree-Based Risk Analysis Using Inexact Estimates." Report GWU-IIST-81-14, Department of Electrical Engineering and Computer Science, The George Washington University, Washington, D.C., June 1981.
17. Zadeh, L. A., K. S. Fu, K. Tanaka, and M. Shimura (Eds.). *Fuzzy Sets and Their Applications to Cognitive and Decision Processes.* New York: Academic Press, 1975.
18. Denning, Dorothy. *Cryptography and Data Security.* Reading, Massachusetts: Addison-Wesley, 1982.
19. Okrent, D. "Risk-Benefit Evaluation for Large Technological Systems." *Nuclear Safety* (Vol. 20) No. 2, March–April 1979, pp. 148–164.
20. Lowrance, W. W. *Of Acceptable Risk: Science and The Determination of Safety.* Los Altos, Calif.: William Kaufmann, Inc., 1976.
21. Rowe, W. L. *An Anatomy of Risk,* New York: John Wiley and Sons, 1977.

Uniform help facilities for a cooperative user interface

by PHILIP J. HAYES
Carnegie-Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

This paper describes the design of the help and explanation component of a user-friendly operating system command interface called COUSIN. The facility can provide two kinds of information: (1) static descriptions of the various subsystems that can be invoked, their parameters, and the syntax that must be used; (2) dynamically generated descriptions of the state of the current interaction, why that state has arisen, and what the user's options for action are. Both types of information are presented in the same way through a network of small text frames connected by semantically motivated links in the style of the ZOG system. Frames containing static information are generated automatically for each subsystem from a declarative description of the subsystem which is also used by COUSIN for its other services, including spelling and grammar correction and interactive error resolution. Dynamically generated frames are incorporated temporarily into the static network with semantic links appropriate to the current command context.

INTRODUCTION

The COUSIN (COoperative USer INterface) project at Carnegie-Mellon University is engaged in a wide-ranging program of research[1,2,3,4,5] aimed at producing interactive command interfaces that appear to their users more friendly and cooperative than most presently available interfaces and are thus less frustrating and more productive to use. This research includes work on flexible, error-correcting parsing, interactive error resolution, and the topic of this paper: interactive help facilities. The following paper presents the design and rationale for the help facility that we are currently implementing for use with the COUSIN interface.

The COUSIN help facility provides explanations of two kinds:

- Static explanations: Descriptions of the aspects of the system that do not normally change during the course of a terminal session, such as what subsystems are available through the interface, what are their parameters, what is their syntax, what is their purpose, and similar aspects.
- Dynamic explanations: Information dependent on the immediate context of the request, such as what the user's current options for action are, what is the last command performed, what the interface expects the user to do next, how the interaction came to be in its present state, and related information.

COUSIN distinguishes these types of explanation because of the way they are generated: the static explanations are prestored and displayed on demand, whereas the dynamic explanations must be generated on the fly. However, since this distinction is not likely to be of great concern to the user, a design goal of COUSIN is that the two types of explanation be presented in as uniform and integrated a way as possible. Integration is particularly important if a dynamic explanation (e.g., what commands are currently available) leads the user to request a static explanation (e.g., the syntax for one of the available commands).

By far the most common way to provide online help is to use canned text messages. Such messages are either written into the system by the system designer specifically for interactive use, as in the SOS editor,[6] or extracted by an indexing scheme from an online version of the system manual, as in the RdMail electronic mail system[7] or the CMULisp system.[8] This approach is susceptible to two problems: first, the blocks of canned text may be too long and ill-structured, so that the user must search through irrelevant material to get to the information actually needed; and second, the blocks of text may be insufficiently linked or cross-referenced, so that the user may be unable to locate the needed information even if he has found a related piece of information.[9] To avoid these problems, a second design goal for the COUSIN help facility is to present information in fine-grained chunks that are heavily interlinked and cross-referenced.

The help and explanation facility for COUSIN satisfies the two design goals mentioned above by following an approach similar to that of the ZOG menu-selection system,[10] also developed at CMU. ZOG allows its user to traverse a network of small (one-display terminal screen) text frames linked together in an arbitrarily complex way by semantically motivated links. Each frame consists of a text part and a menu of links (whose descriptions themselves often convey information). Selecting one of the links causes the display of the frame pointed to by the link. COUSIN's help and explanation facility consists of a similar network of text frames connected by mnemonically named semantic links; the frames are of variable size, but usually contain less than a screenful of information. This arrangement clearly satisfies the requirement that the help information be available in small-grained chunks and allows for a set of interconnections rich enough to make information related to that of the current frame easy to obtain. Moreover, it provides a convenient way of integrating static and dynamic help. The static help can be represented as a prestored frame network, and the dynamic help can be expressed through frames that are constructed and linked into this static network on the fly. This makes the presentation of static and dynamic help uniform and allows the dynamic explanations to provide convenient access to related parts of the static network.

A further advantage of the ZOG approach for COUSIN is that the static explanation net can be generated automatically from information already required by COUSIN for other graceful interface functions. Briefly, COUSIN requires a declarative description of each subsystem with which it is used. These *subsystem descriptions* contain, among other things, details of the subsystem's parameters, their types, whether they are optional, and the syntax used to express them. Using this information, COUSIN can accept commands from the user, check them for validity, fill in defaults, correct some errors and ambiguities, interact with the user to correct the others, and finally transmit the corrected command to the underlying system. More important for present purposes, COUSIN can use the same information, supplemented by text strings to explain the purpose of subsystems and their parameters, to construct a fine-grained and highly interconnected static explanation network for the subsystems concerned. The following sections discuss the construction and structure of this static explanation network and examine how dynamic explanation frames can be generated and linked into it on the fly.

AUTOMATIC CONSTRUCTION OF STATIC EXPLANATION NETWORKS

This section describes the structure of the static help offered by the COUSIN interface; i.e., the explanations COUSIN can give about the subsystems it provides an interface to, their invocation syntax, their parameter types and structure, and related matters. We also describe how these static explanations are constructed automatically from the declarative subsystem description databases from which COUSIN also obtains the information it needs to provide its input checking and correction services. Generating the static help frames automatically enables the subsystem designer to provide a structured set of online documentation for the subsystem with little incremental effort. The designer needs to construct a subsystem description in any case to use COUSIN's other services. To produce the help documentation, he merely needs to add some text fields to the subsystem description to indicate the purpose of the subsystem and its various parameters, and the rest can be done by COUSIN. We believe that the effort involved in producing documentation in this form is far less than that involved in producing documentation with the same content by hand. Moreover, the resulting documentation will have a considerably more uniform structure, both within a single subsystem and, more important, across a wide range of subsystems. Glasner and Hayes[11] give details of the way this approach was employed to produce online documentation for an ancestor of the present COUSIN interface, and Fenchel[12] has used similar techniques to generate user-oriented descriptions automatically from a parser's grammar.

Before examining the form of the explanations COUSIN generates, we should first look at the structure of the subsystem descriptions from which they are generated. One of the applications of COUSIN is to provide a command-level interface to the UNIX operating system on a VAX 11/780; i.e., to provide an alternative to the standard Unix shell. As described above, COUSIN requires each subsystem, in this case each Unix command, to have a declarative subsystem description. Figure 1 shows an abbreviated subsystem description (there are actually many more optional parameters) for the "cz" command, used in our department to print files on the Dover, a local hard-copy output device.

The details of this notation are not important for present purposes. It is enough to note that the "Schema" indicator introduces a property list whose indicators ("alignment," "copies," etc.) name the parameters, optional and required, of cz, and whose corresponding values give more detailed information about the types, defaults, and related matters concerning those parameters. As mentioned previously, this information, together with the "Syntax" property, is used by COUSIN to provide input checking and correction on the syntax and parameter values of invocations of cz.

The same information in conjunction with the "Explanation," "Description," and "Summary" fields can also be used to generate a tree-structured net of explanation frames describing cz at varying levels of detail, corresponding to different levels of the tree. The root of the tree, for instance, is a brief description of the purpose and calling syntax of cz, shown in Figure 2.

```
[SubsystemName: czarina
Schema: [
  alignment: [FillerType: SelectionSet
              Set: (vertical rotated)
              Default: vertical
              SynType: FlaggedOption    BinaryFlag: r
              Description: "determines whether the printing will be in
                            standard orientation on the page (vertical),
                            or rotated 90 degrees (rotated)"
              Summary: [rotated: "prints parallel to the short
                                  side of the paper"
                        vertical: "prints parallel to the long
                                  side of the paper"
                       ]
             ]
  copies: [FillerType: Integer    LowerBound: 1
           Default: 1
           SynType: Option     Marker: c
           Summary: "prints <integer> copies"
           Description: "determines how many copies will be printed"
          ]
  font: [FillerType: String
         Default: Gacha10
         SynType: Option     Marker: f
         Summary: "uses font specified by <string>"
         Description: "determines the font used to print ascii files"
        ]
  tobeprinted: [FillerType: File
                Number: +
                SynType: Argument
                Explanation: "a list of files to be printed"
               ]
 ]
Syntax: [Format: OptsArgs
         OptionTag: -       OptionJuxtaposition: on
         ArgumentOrder: (tobeprinted)
         CommandName: cz]
Explanation: "prints the specified files on the Dover,
              converting ascii to press format if necessary."
]
```

Figure 1—An example subsystem description

```
COMMAND   cz (czarina)

USAGE     cz [options] tobeprinted+

PURPOSE   prints the specified files on the Dover,
          converting ascii to press format if necessary.

LINKS     1.  options
          2.  tobeprinted - a list of files to be printed
          3.  Unix syntax - general
```

Figure 2—Root description frame for cz

If, after reading this text frame, the user decides more information is needed, he can select one of the links, which will result in the display of another frame. The "Unix syntax" link leads to a manually constructed frame explaining the general style of Unix syntax. The tobeprinted link leads to the frame shown in Figure 3. From this frame, the user can obtain the general syntax for files (from another hand-constructed frame), if that is what he needs. Taking the "options" link from the root frame would lead to a list of options, as shown in Figure 4.

Taking the "rotated" link from this frame would lead to the frame shown in Figure 5.

```
ARGUMENT  tobeprinted - a list of files to be printed

NUMBER    one or more

DEFAULT   none

SYNTAX    file1 ... filen

LINKS     1. file syntax - general

          2. notation
```

Figure 3—Frame for tobeprinted parameter of cz

```
Summary of the options available with cz
        (Unix abbreviated syntax in parentheses)

LINKS   1.  rotated (-r)
                prints parallel to the short side of the paper

        2.  copies <integer> (-c <integer>)
                prints <integer> copies

        3.  font <string>  (-f <string>)
                uses font specified by <string>
```

Figure 4—Frame for optional parameter of cz

```
PARAMETER  alignment - determines whether the printing will be in
                       standard orientation on the page (vertical),
                       or rotated 90 degrees (rotated)

FILLER     one of {vertical rotated}

DEFAULT    vertical

SYNTAX     rotated  (-r)
                prints parallel to the short side of the paper

           vertical
                prints parallel to the long side of the paper
```

Figure 5—Frame for alignment parameter of cz

Along with the specific links mentioned above, each frame has links that allow the user to return to the previous frame, go to the root of the present tree of frames, and obtain explanations of the notation being used or of the organization of the help system in general. There is no space here to give details of the program used to generate the static help frames shown above, but comparison between the subsystem description and the frames should convince the reader that only a relatively straightforward rearrangement of the information in the subsystem description is necessary.

A complete static help facility for COUSIN is obtained by applying the frame-generation process to the subsystem descriptions of each subsystem that can be invoked through COUSIN. A set of trees of text frames results, with the tree nodes linked together in ways illustrated previously. The static help network is completed by hand-generated auxiliary frames that provide general information about syntactic conventions and about the organization and notation of the help system itself.

The user obtains the display of a static help frame by giving the name of the frame as a parameter to the help command. The name of the root frame for a subsystem is the same as the name of the subsystem; therefore "help cz" will result in the display of the first frame shown above. Frames below the roots of the trees are named incrementally according to the links that should be followed to get to them; thus the last frame shown above is called cz.options.alignment and may be accessed directly by this name as well as by following links from the cz base frame. Of course, when frames go out of a subsystem tree, this results in a frame having more than one name; thus cz.tobeprinted.file is also called file.syntax.

DYNAMICALLY GENERATED EXPLANATIONS

Besides the network of static help frames, the COUSIN help facility can also display dynamically generated help frames to give the user contextually dependent help. The kinds of questions these frames are designed to answer include the following: What is the current state of the interaction, what can I do now, what does the system expect me to do, how did the interaction get into this state? The dynamically generated frames are of exactly the same form as the static frames and will normally have links to the static network. Static and dynamic explanations thus appear uniform and integrated to the user.

Currently, dynamic frames are generated in only one kind of situation: when the user makes a request for help without giving the name of a static help frame. When this happens, COUSIN constructs a frame containing the following information:

1. The current state or mode of the interaction.
2. Why the interaction is in this state.
3. A list of actions the user is likely to want to take.
4. Links to other frames that allow the user to find out about the complete range of options available.
5. Links to frames describing contextually relevant subsystems and aspects of subsystems.

A frame of this sort is put together out of canned text templates, one for each of the various components; variables in the template are filled in by contextually appropriate tokens. An example will make this clearer.

Suppose the user gives the command

cz -c 2 rotated foo.press

to the main command level of COUSIN. The command is to print two copies of foo.press in 90-degree rotated mode. Suppose also that the user does not have read permission on foo.press. COUSIN prints out the message:

no read permission for foo.press
envedit〉

where the second line is a prompt for the *environment editor.* For present purposes, we can regard an *environment* as a set of parameter/value pairs for a subsystem invocation. When an error in a command line is detected, COUSIN makes up an environment from the command, then enters the environment editor to allow the user to alter the problematic parameters without having to retype the rest. Now suppose that the user in this case is unsure of what is happening and so makes a nonspecific request for help by typing "?" or "help." Following the recipe listed above, COUSIN then constructs the help frame shown in Figure 6. Note how the LINKS section gives more detail about the command the user is most likely to want to use, as well as pointers to the other commands available. The other links are to relevant frames in the static network.

```
You are in environment editing mode through which you can change the
value of slots in the current environment, which was constructed from:
   cz -c 2 rotated foo.press

You are in this mode because "foo.press" does not satisfy the
requirement that the "tobeprinted" parameter of cz be a readable file.

LINKS    1. correct - environment editor command to cycle through and
                      change incorrect slots
         2. other environment editing commands
         3. environments - general information
         4. cz
         5. tobeprinted parameter of cz
         6. files
```

Figure 6—A dynamically constructed help frame

CONCLUSION

This paper has described the design of the help and explanation facility that we are currently implementing for the COUSIN operating system command interface. COUSIN will provide online help and explanations through display of text frames connected in a network by semantically motivated links in the style of the ZOG[10] system. Most of the frames containing information that does not normally change over the course of a terminal session can be generated automatically from declarative representations of the various subsystems to which COUSIN interfaces. These declarative subsystem descriptions are necessary for other aspects of COUSIN's operation but must be supplemented by a number of text fields to provide good help frames. In addition to this static help network, COUSIN will provide dynamically generated, contextually sensitive explanations about the current state of the interaction. These dynamic explanations can be assembled out of predefined templates into the same text frame form as the static help and temporarily linked into the static network so that the help system as a whole appears consistent and uniform to the user.

ACKNOWLEDGMENTS

A program written by Pedro Szekeley demonstrated the feasibility of constructing the static help network automatically from subsystem descriptions.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

REFERENCES

1. Ball, J. E., and P. J. Hayes. "Representation of Task-Independent Knowledge in a Gracefully Interacting User Interface." *Proceedings of the 1st Annual Meeting of the American Association for Artificial Intelligence,* Stanford University, August 1980, pp. 116–120. Obtainable from American Association for Artificial Intelligence, 445 Burgess Drive, Menlo Park, CA 94025.
2. Carbonell, J. G., and P. J. Hayes. "Dynamic Strategy Selection in Flexible Parsing." *Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics,* Stanford University, June 1981, pp. 143–147.
3. Hayes, P. J. "A Construction Specific Approach to Focused Interaction in Flexible Parsing." *Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics,* Stanford University, June 1981, pp. 149–152.
4. Hayes, P. J. "Anaphora in Limited Domain Systems." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* Vancouver, 1981, pp. 416–422. Obtainable from American Association for Artificial Intelligence, 445 Burgess Drive, Menlo Park, CA 94025.
5. Hayes, P. J., and G. V. Mouradian. "Flexible Parsing." *Proceedings of the 18th Annual Meeting of the Association for Computational Linguistics,* Philadelphia, June 1980, pp. 97–103.
6. Carnegie-Mellon University, Computer Science Department. *Son of Stop-Gap (SOS).* Originally developed at Stanford AI Lab; the help facility was added at CMU. 1978.
7. Carnegie-Mellon University, Computer Science Department. *RdMail Message Management System.* 1980.
8. Carnegie-Mellon University, Computer Science Department. *TOPS LISP.* 1978.
9. Hayes, P. J., J. E. Ball, and R. Reddy. "Breaking the Man-Machine Communication Barrier." *Computer,* 14 (1981), pp. 19–30.
10. Robertson, G., A. Newell, and K. Ramakrishna. "ZOG: A Man-Machine Communication Philosophy." Technical Report, Computer Science Department, Carnegie-Mellon University, August 1977.
11. Glasner, I. D., and P. J. Hayes. "Automatic Construction of Explanation Networks for a Cooperative User Interface." Technical Report, Computer Science Department, Carnegie-Mellon University, 1981.
12. Fenchel, R. S. "Integral Help for Interactive Systems." Technical report UCLA-ENG-8051, Computer Science Dep't., UCLA, 1980.

Natural-language help in the Consul system

by WILLIAM MARK
USC/Information Sciences Institute
Marina del Rey, California

ABSTRACT

If we use the model of asking an expert, it is fairly clear what users want from a help system: a service that tells them how to do something they want to do. But current help systems aren't like this. They can tell the users about system capabilities, but not in relation to what they want to do. Most help systems are simply databases of online documentation—system manuals, not system experts. Like system manuals, using them to figure out how to do something or to figure out what went wrong is a last resort—when no one else is around.

Providing real expert help requires reasoning in terms of models of what the user wants to do and what the system can do. These models also make it possible to provide facilities for natural-language understanding and generation. Thus, users can deal with the system in much the same way that they deal with a human expert: by asking questions and receiving advice in English.

These ideas are being tested in the Consul system, a research prototype currently under development at the USC Information Sciences Institute.

1. WHAT IS HELP?

Users of an interactive system need help whenever they encounter an obstacle that prevents them from performing a task. They may need to find out how to do something (*How do I get rid of the messages Smith sent me yesterday?*), how to describe an object (*What has to be in a message?*), or why something unexpected occurred (*What happened to the message that was on my screen?*). This need for help may be expressed by the system rather than by users, as when a user runs into an error condition (*You can't forward a message you have composed*). The point is that the need for help is a feature of users' conceptual framework: it expresses the difference between their state of knowledge and their expectation of how to specify a task to be performed by the system.

Users' conceptual framework may be very different from the system's. Even though the question *How do I get rid of the messages Smith sent me yesterday?* makes perfect sense to a user, it may not relate directly to anything in a mail system whose "messages" always reside in a central database and are never sent to (and can never be deleted by) users (e.g., [SIGMA 79]). The user's problem must be mapped into the system framework before it can be solved. Then, since the user may not understand the solution in system terms, it must be mapped back into an answer in terms of the user's framework.

2. CURRENT HELP MECHANISMS

Let's contrast how this is done in the two currently existing interactive help mechanisms: asking a system expert and using an online help system. When a user asks a system expert for help, the expert mentally translates the user problem into the system framework, decides what the user must do, and then translates back into the user's terms in order to explain it to the user. Given the question *How do I get rid of the messages Smith sent me yesterday?* the expert (1) realizes that the user actually wants to delete the citations (records) of a particular set of messages from the mailbox; (2) knows that the system has a command that can filter citations in mailboxes by sender and date and a command for deleting lists of message citations; and (3) tells the user how to invoke the two commands in sequence to achieve the desired result.

Current online help systems operate in a very different manner (for a discussion, see Relles and Price[3]). They contain documentation of system features (which can be thought of as precomputed mappings from the system framework to the user framework), usually indexed in terms of what the system can do, not what the user can do with the system. In other words, the user is responsible for translating need for help into retrieval requests (or menu selections) in order to access the system's documentation database. This often makes it difficult for the user to get to the information wanted (he/she might type "? message" and get a lot of irrelevant information; he/she might have to wade through a list of system commands and guess that "delete" is the one wanted; and so forth). Once the user finds a relevant piece of documentation, he/she may not understand how it relates to the problem (he/she may discover that the delete command takes "a list of messages" and not realize that filters can be used to produce that list).

This points up two shortcomings of current online help systems:

- No explicit representation of the user's framework (the user can't express the problem, but can only search for possibly relevant documentation).
- No flexibility in mapping questions into answers (even if the user could express the problem, it may cut across the precomputed mappings stored in the help database).

There is evidence[3,5] that these shortcomings are sufficient to discourage most users from using online help at all.

In the Consul system we are trying to overcome these problems by building knowledge into the system—models of what the user wants to do and what the system can do—and providing help by reasoning in terms of these models. The goal is to allow the user to be able to deal with the system in much the same way he/she deals with a human expert: by asking questions and receiving advice in English.

3. HELP IN THE CONSUL SYSTEM

The Consul system[2] is based on a representation of user and system knowledge in a central knowledge base (see Figure 1). This knowledge includes a model of what users want to do with interactive systems, a model of what interactive systems can do, and a set of mapping rules for translating between the two frameworks. When a particular interactive service like a mail system is added to Consul,[1] the knowledge base (including the mapping rules) is particularized to take into account the distinctive features of the service (this process is discussed in the next section).

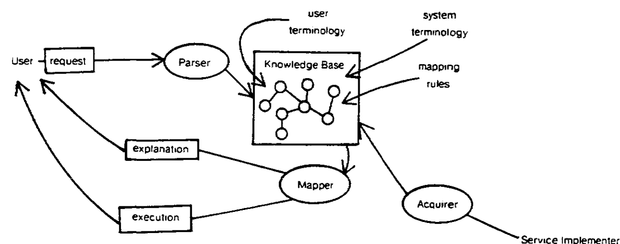


Figure 1—The Consul system

The system's activity consists of mapping descriptions in user terminology into system terminology and, in the case of help requests, back into user terminology. The process begins with a request from the user, expressed in natural language. The request is parsed and represented as user terminology in Consul's knowledge base. It is then mapped into system terminology, allowing Consul to determine whether it is a request for help or a system command. If it is a request for help, the appropriate information about system features (in system terminology) is isolated. This information is then mapped back into user terminology, and finally into English to provide the answer. If a request that Consul originally interprets as a system command cannot be executed, it is automatically reinterpreted as a request for help. This process can be illustrated in terms of the four help interactions mentioned in the first section:

How do I get rid of the messages Smith sent me yesterday? As mentioned above, responding to this request requires mapping of the notion of *messages* in the user framework into *message citations* in the system framework. Consul produces this mapping by first finding that its model of the mail system[4] does not allow the user to delete messages directly, thus preventing the request from being taken at face value. It then finds that it has a mapping rule that translates user statements about doing things to objects into statements about doing things to summaries of those objects (reflecting the fact that users often say things like "show me a list of my files" when they really want a directory listing, a list of summary information about files). This mapping rule translates the original request into a request to delete a list of message summaries. This can then be mapped into a sequence of the mail system's actual operations for filtering and deleting message citations. Examination of these operations reveals that they (collectively) require a user mailbox and a set of filters—in this case, a sender and a date—as arguments. This information is in fact the answer to the user's question, but it must be mapped into user terminology before actually being displayed to the user. The user terminology in the knowledge base is checked to see if there are constructs corresponding to those found in the system framework. In this case there are, and an English response is generated: "Lists of messages for deletion can be specified by any combination of selectors such as sender, range of dates, and message numbers, as in 'Delete the messages from Smith I received yesterday.'"

What has to be in a message? In this example, the term *message* can be taken at face value. That is, *message* in the user world is taken to refer to *message* in the mail system world as well. The mail system's representation of a message is therefore examined for required fields, which are collected, and, as before, mapped back into corresponding user terms to provide a response to the user: "Messages must have a valid addressee and usually have a subject and body of text."

What happened to the message that was on my screen? This request refers to an event (something "happening" to a particular message); Consul must therefore examine its history records. It first determines which message was most recently displayed on the user's screen, then looks for all of the events that involved that message since it was displayed. Many things could have "happened" to that message since it was displayed (e.g., other users could have received pointers to it in the central database), but few of these events would have been noticed by the user. The system must compare each event involving that particular displayed message with the events of the user world. Those that have significance in the user world are collected, mapped into their corresponding user terminology, and presented in the response: "I had to take it off the screen temporarily to show the list of messages you requested."

You can't forward a message you have composed. This help interaction is not requested by the user; it is generated by the system in response to a user request to forward a particular message (e.g., *Forward this message to Jones.*). Initially, the request is mapped into system terms, and Consul recognizes it as a command for system action. Let us assume that the message involved is one that the user has just composed, not one that he/she has received. The mail system will not forward composed messages. This is reflected in the Consul system by an inability to map the supposed command into an actual mail system execution sequence. Consul therefore knows that it must generate a help response. It compares the request (as mapped into system terms) with the forward function the user was trying to execute, thus finding the differences between the request as stated and the required form of the arguments of *forward.* These differences are presumably what prevented the command from being invoked in the first place. In this case, Consul discovers that the forwarding function in the mail system requires a "transmitted" message, while the message in the request is of type "draft." It can map this difference back into user terminology to generate the response shown above.

But in many cases (including this one), Consul can go further in responding to errors. If Consul can find system functions similar in intent to the one the user was trying to execute, it can use them as targets and find differences just as before. This allows it to suggest alternatives that accomplish the user's goal. In this case, the system discovers that *send* as well as *forward* could accomplish the user's presumed intent of getting a particular message to a particular user. Therefore, when the user makes his/her erroneous forwarding request, Consul actually generates the more complete help response, "You can't forward a message you have composed. You can send it to Jones instead."

4. ACQUIRING AND MAINTAINING THE HELP INFORMATION

To provide all this knowledge-based help, a lot of specific information about mail systems and how they are used must somehow be put into the machine. In fact, a major issue for all interactive help systems is how the necessary help information gets into the machine and how it is kept up to date in the face of system changes. The approach taken in the Consul system is to build in a general model of interactive services and how they are used, including a model of how users specify commands and ask for help. This general model is then used to solicit specific help information from the programmer of each actual service.

A programmer of a new service (e.g., the mail system) enters each program into Consul through a dialogue-oriented acquisition aid.[6] The acquirer uses Consul's model of inter-

active systems to ask the programmer how a program fits into the system knowledge base. A dialogue ensues as the acquirer asks the programmer questions in terms of its model and the programmer answers in terms of the program. For example, in acquiring the "forward" program, the acquirer will ask what in the program corresponds to the system model of the "destination" of a forwarding operation. The programmer will reply that it is the "mailbox" whose owner is the user specified as the "receiver" argument of the operation. The acquirer then checks to make sure that mailboxes are legal destinations according to its model, that they can be owned by users, and so on, until it is sure that the programmer's version of "destination" fits the model in the Consul knowledge base. If it does not, the acquirer pursues the dialogue until it discovers how "mailbox" relates to something that *is* legal in its model. In this way, Consul comes to understand each new program in terms of its knowledge base. The acquirer will not allow the program to be part of the Consul system until it sees how it fits into both the system and the user framework. Fitting into the user framework usually requires the construction of mapping rules, also done automatically during the acquisition dialogue.

Once acquisition is accomplished, Consul has all the information it needs to construct the necessary help responses. Since it knows about message forwarding in general—what it does, how users ask about it, what can go wrong with their requests—and it has learned how the particular forwarding operation of the new mail system fits into this scheme, it knows how to handle user requests for help and how to generate comprehensible error explanations. Moreover, every time a program or data structure is changed, the acquirer is automatically reinvoked, thus insuring that the knowledge base is always up to date.

5. CONCLUSIONS

The Consul system is an experiment in providing a natural-language interface, including natural language help, to users of interactive systems. It currently handles only a small part of a single interactive service—the mail system described in this paper. Much work remains to demonstrate its feasibility in cost and execution speed in a real working environment consisting of varied users and services.

ACKNOWLEDGMENT

This research is supported by the Defense Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223. Views and conclusions contained in this paper are the author's and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government or any person or agency connected with them.

REFERENCES

1. Lingard, Robert. "A Software Methodology for Building Interactive Tools." *Proceedings of the Fifth International Conference on Software Engineering,* 1981.
2. Mark, William. "Representation and Inference in the Consul System." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* IJCAI, 1981.
3. Relles, Nathan, and Lynne Price. "A User Interface for Online Assistance." *Proceedings of the Fifth International Conference on Software Engineering,* 1981.
4. Stotz, R., R. Tugender, D. Wilczynski, and D. Oestreicher. "SIGMA: An Interactive Message Service for the Military Message Experiment." *AFIPS Proceedings of the National Computer Conference* (Vol. 48), 1979.
5. Stotz, R., D. Wilczynski, S. Finkel, R. Lingard, D. Oestreicher, L. Richardson, and R. Tugender. *SIGMA Final Report: Lessons.* Information Sciences Institute, Technical Report ISI/RR-81-97, May 1981.
6. Wilczynski, David. "Knowledge Acquisition in the Consul System." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* IJCAI, 1981.

Programs as data for their help systems

by ELAINE A. RICH
The University of Texas at Austin
Austin, Texas

ABSTRACT

The goal of this research is to develop ways of representing the knowledge available to a help system in such a way that the system can actually reason with the knowledge rather than being restricted to simply retrieving and presenting stored answers to a restricted and anticipated class of questions. One kind of information that is useful to such an intelligent help system is knowledge of how the underlying system operates. This knowledge is contained in the code for the system. By exploiting system code as part of the help database, many problems of inconsistency between programs and their documentation can be avoided. In our initial investigations of this problem, we are representing the system code as a set of productions that are easier to manipulate than is code in most standard languages. As we develop techniques for answering questions by reasoning with knowledge about the system, we become increasingly able to answer the growing variety of questions that will occur as the language interface to a help system becomes more flexible.

INTRODUCTION

As complex software systems become more and more widely used by a larger and more heterogeneous group of users, it becomes increasingly important to provide, along with the systems themselves, good interactive help facilities. But as the size of a system grows, so too grows the size of the task of building the database to be used by its help facility. And building the database initially is only a small part of the problem. As the underlying system changes, the help database must also be maintained so that it always corresponds to the current version of the system. This is a large, boring, and all too often ignored task. A major goal of my research in building help facilities is to explore ways in which the task of building a database specifically for a help facility can be minimized by exploiting the underlying system itself as a major part of the required knowledge base. Using the system code as the help facility database guarantees that the two will always correspond.

A second reason for building a help facility based on the underlying system itself, rather than on a body of stored text that is fed to users on demand, is that a wider variety of user questions can be handled. If prestored text is used, all the questions to which the system will be able to respond must have been anticipated at the time the text was created so that an appropriate answer can be written. But if answers can be computed from the code of the system itself, this is not necessary. So, for example, questions such as "What is the difference between *a* and *b*?" can be answered by looking at what happens for *a,* looking at what happens for *b,* and comparing the results. An answer can then be generated that is based on that comparison. This flexibility becomes increasingly important as we move toward natural-language help systems in which the superficial flexibility of the system is high, leading users to expect comparable flexibility in the actual power of the system.

To explore the issues raised in the design of this sort of help facility, I have begun building a help system for the document formatter Scribe.[2] Scribe is a sufficiently complex program that most users never learn all of it. Thus a help system for Scribe will need to be able to handle a wide range of questions from a broad class of users, ranging from novices to experts. As discussed in the following sections, the approach we are taking to answering user questions by referring to the system code makes it possible to tailor the responses generated to the individual needs of all these users.

ANSWERING USER QUESTIONS FROM SYSTEM CODE

Although it is true that not all the kinds of questions that people will want to be able to ask a help facility can be answered by examining the system code, a great many of them can be, particularly if the code is well structured. There are three basic categories of questions whose answers can be derived from the code for a system:

1. The user gives a description of a result and wants to know what causes that result to appear. A few examples of this kind of question are as follows: "Why is my paper coming out single-spaced?" and "How can I get the page numbers printed at the bottom of the page?" As these examples show, sometimes result-description questions occur because an undesirable (or perhaps surprising) result has occurred and users are curious about the reason. Other times these questions describe a desired result, and users want to know what they can do to get it. In either case, the way to answer the question is to find the place in the code where the described result is generated. Then look to see what conditions must be satisfied for that particular code to be executed.
2. The user gives a description of a set of circumstances and asks what would happen if they occurred. A few examples of this kind of question are as follows: "What will happen if I change the reference format to alphabetic?" and "How does Scribe float figures?" As these examples show, circumstance-description questions occur both when users are curious about what will happen if they do some new thing and when they want to find out exactly how Scribe performs a function in which they are interested. The way to answer this type of question is to find the place in the code that corresponds to all the specified conditions being met. Then look to see what action is performed. Depending on the level of detail appropriate for the answer, either the action can be described as a single action, or the lower-level procedures that compose it can be described.
3. The user gives two descriptions and asks for the difference between them. An example of this sort of question is, "What is the difference between the itemize and the enumerate environments?" This kind of question is answered by searching the code to find out what happens in each of the two circumstances given. Those answers are then compared, and the dissimilar parts are reported as the answer.

By using these three mechanisms, a variety of user questions can be answered easily from the system code, provided that the structure of the code is uniform and corresponds well to the structure of the operations being performed.

REPRESENTING A PROGRAM AS A SET OF PRODUCTION RULES

There are two ways that one could build a complex system and the help facility for that system so that both use the same

code. One is to write the code for the system in the conventional way and then to write a help facility question answerer that can exploit that code. Scribe is written in Bliss, so for this experiment this approach would mean building a question answerer that reasons about Bliss code. The second approach is to develop a new notation for writing the system code and then to build both an interpreter for that system and a question answerer that manipulates it. I have chosen the latter path, for several reasons, including the following:

1. Bliss allows unconstrained use of global variables and side effects. This means that merely on the basis of a static examination of a particular code fragment it is not possible to guarantee much about the behavior of that fragment.
2. Bliss is not a typed language. This means that it is not possible to tell simply from looking at a piece of code what kind of object is being operated on. Since a question answerer will need to be able to give responses that describe what pieces of code are doing, it is important that it have access to information about the types of the objects being manipulated.

In addition to the restrictions that need to be put on the language in which code is written if that code is to serve as the basis for a question answerer, there are other limitations that need to be put on the way that code is written.

Often code is written with deeply nested conditional statements. To determine the exact set of circumstances under which a particular fragment of code will be executed, it is necessary to search up several levels and collect all the conditions that lead down the relevant path to the code. This will be a common operation for the question answerer; so to make its job easier, code should be written with conditions flattened so that all necessary conditions immediately precede the code they guard.

The question answerer must be able to answer questions at a variety of levels of detail. Broad questions should not be answered at the lowest level of detail. To make this possible, the code should reflect a top-down decomposition of the behavior of the system. This will make it possible for the question answerer to select the appropriate level of decomposition to answer each individual question.

The question answerer will generate answers that describe the operation of units of the program. Since people can only comprehend fairly small units at a time, it is important that the code be highly modular, each module corresponding to a comprehensible set of operations.

The only way that the question answerer will know that some intermediate results have a meaning that can be discussed, whereas others do not, is for the important results to be marked in the code. This suggests that function composition should be limited to a few levels and that the results of these compositions should be assigned names that correspond to their function in the task domain.

To make it easy to write code that lends itself well to use by a help facility question answerer, I have designed a production system language in which rules describing a system's performance under a variety of conditions can be written. The rules correspond to a top-down decomposition of the system. Each rule consists of a left-hand side describing the conditions that must be satisfied for the rule to fire and a right-hand side describing the actions that are performed if the rule fires. For example, one of the top-level rules describing the behavior of Scribe is

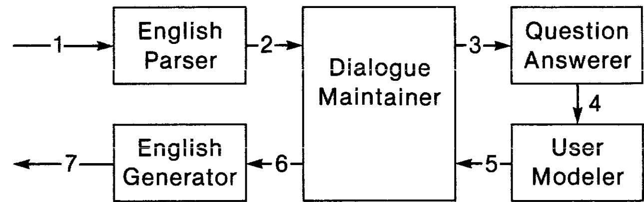


Figure 1—The components of a flexible interactive help facility

```
(PROCESS#FILE X : FILENAME) ➧
    (OPEN#FILE X)
    (FOREACH C : CHARACTER IN X (PLACE# C))
    (CLOSE#FILE X)
```

This rule says that if you want to process a file x, then open it, process each character of it, and close it.

This rule is simple. Its left-hand side consists only of the action that needs to be performed. Other rules have more complex left-hand sides reflecting the fact that the way an action is performed may depend on a variety of factors. As an example, consider the following rule for placing an individual character in the output file:

```
(PLACE#CHAR C : CHARACTER)
(EQUAL C END#OF#LINE)
(EQUAL FILEMODE FILL) ➧
    (DISCARD C)
    (PLACE#CHAR BLANK)
```

This rule says that if the system is trying to place a character, the character is the end-of-line character, and the text is being formatted in fill mode (in which each output line is completely filled and input line breaks are ignored), then ignore the end-of-line character but send a blank to the output file.

To experiment with code-based techniques of question answering, I am recoding Scribe as a set of rules such as these. Meanwhile, Scribe still exists and runs as a Bliss program. But this same production rule language could also be used to define a new system so that the rules would also serve as the code for the system. All that would be required would be an interpreter (and probably also a compiler) for the rule-based language.

THE OVERALL STRUCTURE OF THE HELP SYSTEM

In order to turn the rule-based question answerer that has just been described into a useful and complete interactive help facility, a set of other components is necessary. Figure 1 shows what these components are and how they communicate with each other.

The numbers in the figure indicate the information that passes between the components:

1. A user question stated in English.
2. A parsed form of the question.

3. An interpretation of the question in light of known patterns of dialogue structure (as described, for example, by Grice[1]). For example, the question "Can I get Scribe to make an index for me?" would be interpreted as "How can I get Scribe to make an index?"
4. The complete set of answers.
5. An appropriate subset of the answers, chosen to match the individual user's current interests and state of knowledge.
6. A response that makes correct references in the context of the current dialogue.
7. A properly worded English response.

The capabilities of each of the components of this system make possible a greater exploitation of the abilities of the others. For example, without the ability to compute the answers to questions by using knowledge about how the system behaves rather than by simply retrieving pieces of stored text, knowledge about an individual user and how much detail he/she is interested in is often wasted.

REFERENCES

1. Grice, H. P. "Logic and Conversation." In P. Cole & J. L. Morgan (eds.), *Studies in Syntax, Volume III.* New York: Seminar Press, 1975.
2. Reid, Brian. "Scribe: A Document Specification Language and Its Compiler." Ph.D. thesis, Carnegie-Mellon University, 1980.

The implementation of a cryptography-based secure office system

by CHRISTIAN MUELLER-SCHLOER
Siemens Corporation
Cherry Hill, New Jersey
and
NEAL R. WAGNER
Drexel University
Philadelphia, Pennsylvania

ABSTRACT

A cryptography-based secure office system is discussed, including design criteria and a specific implementation. The system is intended to be practical, simple, and inexpensive, but also highly secure. The implementation uses a hybrid scheme of conventional (DES) and public-key (RSA) cryptography. Randomly generated DES keys encrypt messages and files, and the DES keys themselves and a one-way hash of the messages are encrypted and signed by RSA keys. The system provides secure electronic mail (including electronic registered mail and an electronic notary public), secure two-way channels, and secure user files. Timestamps and a special signed file of public keys help decrease the need for an online central authority involved in all transactions.

INTRODUCTION

This article discusses the design and experimental implementation of a secure data processing system for an office or similar organization. The basic design should be usable by any number of users (from one to thousands or more), and the hardware may be distributed. We have attempted to design a practical, simple, inexpensive system whose underlying method of implementation is transparent to users. However, the principal design goal has been security. We also wanted to limit the need for an online (i.e., "always-present") central authority.

We visualize users each having access to a *personal work station* (PWS), i.e., a station with local computing power. While in use the hardware of each PWS will be assumed secure, but no security is assumed for the connections between PWSs or for an individual PWS when not in use. Users are not limited to a single PWS, but may sign on at any convenient one.

The system described here is intended as a supplement to the functions normally provided by operating systems and network managers. Our system provides the following basic functions:

1. Secure electronic mail, including electronic registered mail and an electronic notary public (one or more users who can authenticate a signature, provide a timestamp, and save a copy of the message)
2. Secure two-way communications channels (to give simultaneous interactive dialogue)
3. A secure user file system

Such a secure distributed system requires some use of cryptography.[1, 2] In order to achieve simplicity and low cost along with high security and ease of use, we used a hybrid system of conventional and public-key cryptography.

FUNCTIONAL DESCRIPTION

First we list the system's user-level security-related commands, suppressing other commands needed for an electronic mail system, such as "SearchMailbox," etc. The commands are independent of the particular form of implementation (whether conventional, public-key or hybrid like ours), and the user need not know anything about cryptography.

1. SignOn. (Input: Username, Password. Result: User is authenticated by the Password, and the PWS is initialized and thereby dedicated to the user.)
2. SignOff. (Input: None. Result: The PWS is deinitialized by overwriting sensitive areas and keys, etc.)
3. NewUser. (Input: Username, Password. Result: A new user is enrolled in the system with the password as the means of subsequent authentication.)
4. UpdatePublicKey. (Input: Username, Password. Result: A new public and secret key pair is substituted for the old.)
5. UpdatePassword. (Input: Username, New Password, Old Password. Result: A new password is substituted for the old.)
6. SendSecure. (Input: Destination-name, File [i.e., a "message"]. Optional input: Intermediate-destination-name, Request to register or notarize. Result: The file is timestamped, signed, and encrypted for the user whose name is Destination-name. In case of optional input, the file will first be routed to Intermediate-destination-name for registration or notarization.)
7. ReceiveSecure. (Input: Sender-name. Result: File from user implied by Sender-name is decrypted and authenticated.)
8. Register. (Input: Destination-name, Encrypted file. Result: The file is timestamped, signed, and forwarded.)
9. Notarize. (Input: Destination-name, Encrypted file. Result: The file is timestamped and signed, and a copy is retained before forwarding.)
10. AcknowledgeSecure. (Input: Sender-name, Encrypted file. Result: The file is timestamped, signed, and sent back to sender. This is for use with registered mail.)
11. OpenSecure. (Input: Destination-name. Result: A secure channel is created for immediate interactive use.)
12. CloseSecure. (Input: Destination-name. Result: The secure channel is closed.)
13. SaveSecure. (Input: Filename. Result: The file is saved in the user's mass storage in a secure way.)
14. RestoreSecure. (Input: Filename. Result: The saved encrypted file is made available as unencrypted cleartext.)

Initially, users must give the "NewUser" command with a password that they can remember. "NewUser" requires the physical presence of users at the central authority if authentication that a username corresponds to a particular physical individual is desired. "SignOn" must be given with a password matching the username. Messages or files of any sort can be sent to other users with the "SendSecure" command and can be received with the "ReceiveSecure" command. Various encryptions, decryptions, signatures, timestamps, and authentication steps are built into these commands at a lower level, as described below.

OVERVIEW OF OUR IMPLEMENTATION

In addition to the PWSs, our specific design uses a special component called the *crytoprocessor* (CP) to perform the various encryption/decryption functions, to store and generate keys, and to authenticate users. The CP is part of the PWS. We also use a special file called the *public-key file* (PKF). These public keys are signed with a *network secret key*. (The PKF also contains each user's secret key in encrypted form, as described later.) To replace the memorized username and password as entry to the CP, we can also use a data storage device called a *personal data card*. (Described in "Cryptographic Protection of Personal Data Cards," by C. Mueller-Schloer, submitted to the Seventh International Conference on Computer Communication.) The system design allows the simultaneous use of CPs implemented in either hardware or software. A software CP would be less expensive, perform more poorly and offer lower security than one in hardware. For a production system, a hardware CP could be made difficult to modify (for example, it could be embedded in epoxy), and this should increase security considerably.

Later sections describe the CP and the public-key file in more detail. Figure 1 gives a picture of these components.

We have chosen to use the data encryption standard (DES)[3] for the bulk of the encryption/decryption and to use the RSA public-key cryptosystem[4] for exchange of keys and signatures. DES is a natural choice because of its speed when implemented by inexpensive special hardware. If DES did not seem secure enough, one could switch to triple DES encryption[5] or to some other conventional system.

It is clear that DES alone would suffice for the complete implementation,[6] although with some considerable complications for key distribution and signatures. We have included public keys in a hybrid system because it places less burden on a central authority and allows more autonomy to users. We chose the RSA public-key scheme because it has been thoroughly studied and because of the symmetry between secrecy- and signature-encryption in that system. If RSA ever proves

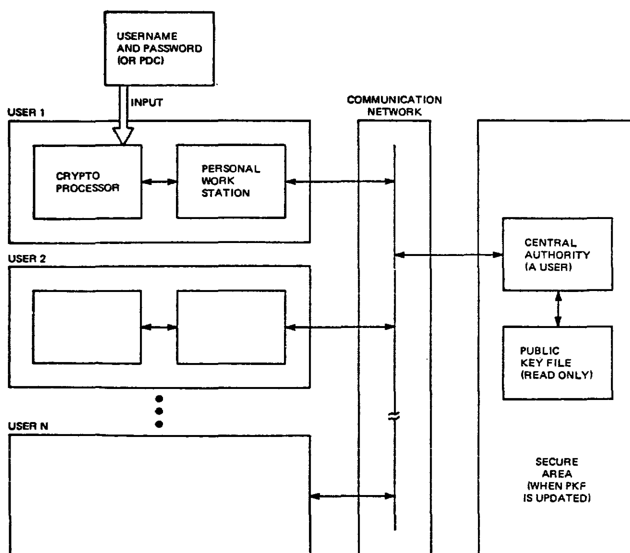


Figure 1—System overview

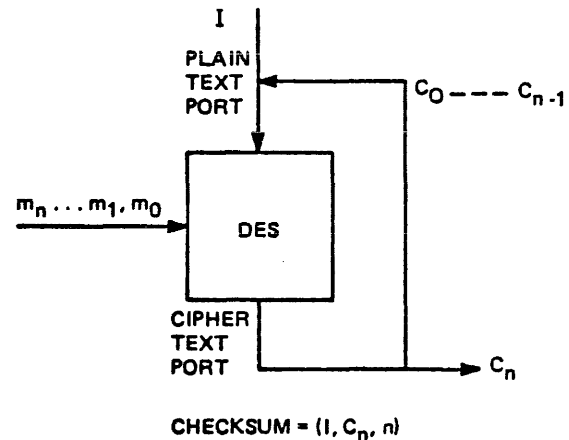


Figure 2—Checksum calculation (I = initialization vector, $m_0, m_1, \ldots, m_n$ = 56-bit blocks of message, $c_0, c_1, \ldots, c_{n-1}$ = 64-bit blocks of ciphertext)

insecure, we could switch to some other secure public-key system. The RSA scheme is slow even with special hardware,[7] and that is why we use a hybrid approach.

Because we wish to sign entire messages or files and yet want to apply the RSA signature only to a few blocks (to save encryption time), some sort of one-way hash function[8] is required. We use the DES to convert any text to a single 64-bit result, which we call a *checksum*. The method is illustrated in Figure 2. Since 64 bits are used, it is not feasible (assuming DES secure) for an opponent to construct an alternate text with the same checksum as the given text.[9, 10]

THE PUBLIC KEY FILE AND ENROLLMENT

The public RSA keys of all users are stored in a special *public-key file* (PKF).[11] (Also described in "The Cryptoprocessor: Hardware for a High-Level Cryptographic Instruction Set," by C. Mueller-Schloer, in preparation.) This file is accessible for reading except when created or updated by a distinguished user called the *central authority* (CA). The CA first generates one pair of RSA public and secret keys, called the *network public key* (PK.N), and the *Network Secret Key* (SK.N). When a user U executes "NewUser," the CA receives the username, the user public key (PK.U), the user secret key (SK.U) encrypted under a random local DES key (Kloc), and a special codeword (CW) for recovering the DES key. The CA forms the checksum (CS) of everything, adds a timestamp (TS), and signs these two with the network secret key. Thus a PKF entry looks like this:

Username, PK.U, <SK.U>Kloc, CW, {CS, TS}SK.N.

(Here < ... > is used for DES encryption and { ... } for RSA encryption.) When decrypted under PK.N, the precise format of the timestamp will serve to authenticate the entry.

As long as the SK.N remains secure, it will not be feasible for anyone to generate fake PKF entries, since the decrypted checksum must match the checksum generated from the first part of the entry. The timestamp is the time the entry was made, or the time the PKF was reconstructed. This timestamp

will prevent an old entry from being substituted for a current one. The time of each PKF reconstruction should be widely distributed; therefore each PKF entry must be timestamped no earlier than the overall timestamp.

An opponent can create fake network secret and public keys and then create a whole fake PKF. This can be defeated by wide dissemination of the network public key. If the personal data card is available, each user will have his/her own copy of the network public key, supplied when he/she enrolls.

Notice that we are not claiming for this system perfect security in the face of the "pervasive deceit" described by Simmons.[11] It is instead a practical approach.

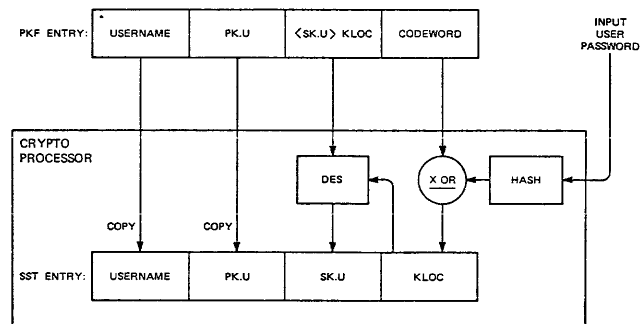


Figure 3—Use of public key file entry at "SignOn" time to initialize security status table of the cryptoprocessor

THE CRYPTOPROCESSOR

As mentioned earlier, the cryptography functions of our system are all concentrated in a component called the *cryptoprocessor* (CP). Initially, we implemented this component in software, but we are now proceeding with a multibus compatible hardware implementation based on the Intel 8086 microprocessor and a Western Digital DES chip. The CP performs the basic functions of encryption/decryption and key generation for both DES and RSA schemes. It provides a well-defined, high-level, crypto-oriented instruction set that cannot be modified from the outside and therefore helps prevent interference by an intruder. Certain sensitive data like passwords and keys are stored internally in CP and can be manipulated only by using the CP's instruction set. (A command "OutputSecretKey," for example, does not exist!) Since RSA key generation on a microcomputer is relatively slow, it will occur as background activity in the CP.

The cryptoprocessor has a protected memory section called the *security status table* (SST). The SST is mostly to be filled in at "SignOn" time, using the input username and password or, if available, the hardware *personal data card* (PDC). (See the next section.) The PDC and the CP hardware solution will be described in detail in subsequent papers ("Cryptographic Protection . . ." and "The Cryptoprocessor . . .," both by C. Mueller-Schloer, cited previously).

PROCEDURAL DETAILS

We have tried to use simplified versions of standard protocols. In particular, we have chosen a public key file[11] and timestamps[13] instead of more elaborate protocols.[6]

In most cases the timestamps on messages that are sent, received, and acknowledged will be relatively close in time, so both parties will agree on the time of a message. Timestamps on registrations or notarizations applied by a third party will serve to settle any disagreements.

Let us go over the actions that occur at "SignOn" time. As Figure 3 shows, the PKF entry contains the username; the user public key (PK.U); the user secret key (SK.U), encrypted under a special Local DES key (Kloc); and a special codeword used to hide Kloc. The Local DES key is an *xor* combination of the codeword and the input user password, so an opponent with access to the PKF cannot recover Kloc and hence cannot calculate SK.U. It is important that we allow passwords of arbitrary length (and encourage long, easily-remembered ones), so Kloc is formed by first getting the checksum of the password and then *xor*ing with the codeword. The codeword itself is formed during the "Newuser" command by *xor*ing the password checksum and Kloc.[14] Thus

$$\text{codeword} = \text{checksum } \mathit{xor} \text{ Kloc},$$

where Kloc is randomly chosen. Since *xor* is self-inverse,

$$\text{Kloc} = \text{checksum } \mathit{xor} \text{ codeword}.$$

In case a hardware personal data card (PDC) is used, everything is handled in the same way except that the SST information originates from the PDC rather than the PKF. "SignOn" results in the work station's being temporarily dedicated to the user performing the operation.

Now consider the "SendSecure" command. User U starts with a file F and destination name D. U generates a random DES key KT and forms <F>KT, the file F encrypted under KT. Then U signs and encrypts for D the DES key KT. Finally U forms CS1, the checksum of everything up to this point, adds a timestamp TS1, and signs this. Thus

$$U -> D, <F>KT, \{ \{KT\}SK.U \}PK.D, \{CS1, TS1\}SK.U$$

is sent to D, where U − > D serves as cleartext routing information.

In case the file is routed through a notary public N, N forms CS2, the checksum of everything, adds a timestamp TS2, and signs the result. The notary public N can also recover CS1 and check that it is the checksum of <F>KT. Thus N adds

$$\{CS2, TS2\}SK.N.$$

Finally the destination D can acknowledge by forming CS3, the checksum of everything received, and signing and encrypting this for U (along with a new timestamp TS3). Thus U receives back what N added on and

$$\{ \{CS3, TS3\}SK.D \}PK.U.$$

Of course there is no need for D to send back what U originally sent out. The notary public will keep a copy of what he added on, but not of the original encrypted file. Thus all

traffic is of a relatively small size except for the encrypted file itself.

When a secure two-way channel is opened, the result is that a common DES key resides in the SSTs of the communicating stations. For key distributions public key encryption is used.[15] The file security system uses the local DES key Kloc for file encryption/decryption. A highly secure registered mail system is under development.

CONCLUSION

We have designed and implemented an experimental secure communications system for a network of personal work stations. The underlying cryptographic mechanisms guarantee a high level of security but are totally transparent to the user. The functionality matches that of today's paper mail security procedures. The confinement of security-related processing to one hardware device (the cryptoprocessor) with a well-defined high-level instruction set allows for higher speed and better protection of sensitive areas. The use of public-key cryptography limits the need for a heavily involved central authority. Other than publishing one network public key, no predistribution of keys is necessary. Users are not restricted to their own workstation but are assured full mobility in the network.

ACKNOWLEDGMENT

During part of this research the second author was supported by a grant from Siemens Corporation.

REFERENCES

1. Lagger, H., C. Mueller-Schloer, and H. Unterberger. "Security Aspects of Computer Controlled Communications Systems" (in German). *Elektronische Rechenanlagen,* 22 (1980), pp. 276–280.
2. Denning, D. E. "Secure Personal Computing in an Insecure Network." *Communications of the ACM,* 22 (1979), pp. 476–482.
3. "Data Encryption Standard." Federal Information Processing Standard (FIPS) Publication No. 46, National Bureau of Standards, January 1977.
4. Rivest, R. A., A. Shamir, and L. Adleman. "A Method of Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM,* 21 (1978), pp. 120–126.
5. Merkle, R. C., and M. E. Hellman. "On the Security of Multiple Encryption." *Communications of the ACM,* 24 (1981), pp. 465–467.
6. Needham, R. M., and M. D. Schroeder. "Using Encryption for Authentication in Large Networks of Computers." *Communications of the ACM,* 21 (1978), pp. 993–999.
7. Rivest, R. L. "A Description of a Single-Chip Implementation of the RSA Cipher." *Lambda,* 1 (1980), pp. 14–18.
8. Merkle, R. C., and M. E. Hellman. "Hiding Information and Signatures in Trapdoor Knapsacks." *IEEE Transactions on Information Theory,* IT-24 (1978), pp. 525–530.
9. Mueller-Schloer, C. "The Usage of DES-generated Checksums for Electronic Signatures." Internal Report CRT-81-TM-043, Siemens Corporation, Cherry Hill, New Jersey, September 1981.
10. Davies, D. W., and W. L. Price. "The Application of Digital Signatures Based on Public Key Cryptosystems." In J. Salz (ed.), *Proceedings of the Fifth International Conference on Computer Communication,* The International Council for Computer Communication, 1980, pp. 525–530.
11. Simmons, G. J. "Secure Communications in the Presence of Pervasive Deceit." In *Proceedings of the 1980 Symposium on Security and Privacy.* IEEE Computer Society, 1980, pp. 84–93.
12. Merkle, R. C. "Protocols for Public Key Cryptosystems." In *Proceedings of the 1980 Symposium on Security and Privacy.* IEEE Computer Society, 1980, pp. 122–134.
13. Denning, D. E., and G. M. Sacco. "Timestamps in Key Distribution Protocols," *Communications of the ACM,* 24 (1981), 8, pp. 533-536.
14. Wagner, N. R., "Practical Approaches to Secure Computer Systems," Technical Report UH-CS-81-3, Computer Science Department, University of Houston, Texas, April 1981.
15. Popek, G. J., and C. S. Kline, "Encryption and Secure Computer Networks," *Computing Surveys,* 11 (1979), 4, pp. 331-356.

Criteria for a standard command language based on data abstraction

by DAVID BEECH
Hewlett-Packard Company
Palo Alto, California

ABSTRACT

A solution is offered to some fundamental problems that have thwarted previous efforts to develop a standard command language. The technical approach is based on the form of modularity provided by data abstraction, and this is introduced from the point of view of the end user, together with a discussion of the advantages and disadvantages that might be perceived at this level. This leads to the statement of a simple but stringent set of criteria for the inclusion of functional capabilities in a standard command language and the testing of various candidates against them. Some candidates are accepted and others rejected, resulting in an initial proposal for the scope of a standard command language that is small and simple enough to have a hope of success.

Do not multiply entities without necessity.

William of Ockham (1285–1349)

INTRODUCTION

Command languages are at a crucial stage of their development, with considerable pressure to define a standard command language, but a dearth of good proposals. Many users are becoming impatient for a uniform method of access as they are confronted with an ever wider variety of systems and heterogeneous networks, and they very reasonably hope that a standard command language will be a distinct improvement over existing command languages. Yet there have been committees working within the American National Standards Institute since 1969 without evident success. Other national groups have engaged in preliminary skirmishes, CODASYL has tried its hand, and there is even a danger that the ADA infantry will aim to persuade us that what is good for embedded military personnel is good for us too. Now the International Standards Organization has been called upon to find a way of bringing order out of chaos.

A dozen years with so little progress suggest that there are some fundamental problems that have not been properly addressed. It is the thesis of this paper that these problems can be attacked by means that are sound in theory and viable in practice. Some surgery is required—or, more precisely, the application of Ockham's razor, the philosophical principle of conceptual economy. The result could be a standard command language that would make a rational start to providing uniformity for the user and would offer a framework within which more widespread standardization could evolve.

Some of the fundamental problems besetting previous efforts have been:

1. A lack of criteria for placing any bounds on the potentially large set of commands that might be included as a defined part of the command language
2. A lack of consensus about the detailed semantics of system functions to be invoked by commands
3. The difficulty of making richness of system function comprehensible to the user
4. The temptation to make command languages too much like programming languages

Since some form of modularity holds promise as an approach to the solution of each of these problems, and *data abstraction* has been extensively developed as a means of achieving modularity in programming languages,[1–4] this is the conceptual tool I shall employ. The operations that can be performed on a system will be modularized, i.e., partitioned into sets of operations that can be performed on different types of object, such as particular types of database system or text editor. The command language will then provide for such operations to be invoked; but the definitions of their semantics will reside within the various object types, whose potential for standardization becomes a set of separable questions. These should be addressed by specialists in their functional areas, who could most effectively be organized into distinct standards committees within a modular framework.

This will lead to solutions to each of the problems listed earlier:

1. Criteria will be proposed which limit the command language itself to a handful of commands.
2. Semantic controversies will then be kept within the confines of particular types of object and need not always produce an outright winner; e.g., more than one type of database model may be offered.
3. Modularity helps the user by reducing arbitrary complexity to more comprehensible interactions within and between types of object.
4. The command language will be deprived of general-purpose programming capabilities; the implication is that programming languages must also be able to invoke the operations accessible from the command language.

USER VIEWPOINT

The data abstraction concept

An intuitive way of describing data abstraction is that data are pictured as residing in black boxes that conceal their representation. All that is known is the set of operations that may be applied to a particular type of black box, and the definition of the responses that will be returned (Figure 1). The responses may depend on previous operations, so one way of modeling this is to think of the black boxes as having states that may be changed by operations.

How does this affect the command language user's view of a system? First, consider users of a conventional command language. They issue sequences of commands to one large black box; e.g.,

```
logon beech
copy MCL MCL2
edit MCL2
  ...
compile MCL2
run
```

In this example the verbs are all distinct, since they are all being interpreted at the same level, as it were, by a single black box. This is the monolithic approach: the commands all belong to one language, symbolized by their being described

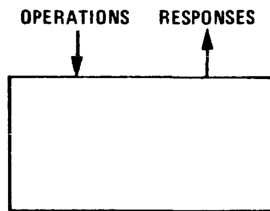


Figure 1—An abstract object

in one massive manual. It is the degenerate case of data abstraction.

I must emphasize that, for the purposes of this paper, I shall work with this verb-and-operands model of the essential information in a command without attending to important, but secondary, questions of alternative representation via syntactic sugaring, special keys, menus, prompting, and so forth.

Some commands, such as "edit," probably put the user into a *mode* in which subsequent inputs are directed to the text editor rather than the command language processor, and these inputs may include a separate sublanguage of commands to be used at this deeper level until it is decided to return to the top level. Certainly I shall retain the concept that a command passes the user's input/output port to the operation being invoked, and I shall return to the topic of modes shortly. But this is still a restricted way of offering modularity, which is especially clumsy when issuing a single command to a particular component of a system, requiring three commands in all: to enter the mode, issue the useful command, and return from the mode.

Suppose we call a black box an *object* and conceive of our outermost system object as being populated by interior objects that communicate with each other by means of operations and responses (Figure 2). Then we would achieve full flexibility if we could immediately name both an object and a particular operation to be performed on it, say with a qualified name of the form *obj.op*. Commands with certain simple names like "logon" might then be acted upon directly by the command language processor object, while those such as "mymail.read" would merely relay the "read" operation to the object "mymail," which might be my electronic mail system. Definition of such a "read" command would then be the business of "mymail," not of the command language, and the name need only be unique among the operations of "mymail." It is as though the command language processor object offered a "run" operation, which acted on any composite name to relay the specified command to (and response from) the specified object.

Thus the visible difference to the command language user could be very slight, but even this much difference might be too great! If there were only one "read" operation available to (or normally used by) a particular user, why should it be necessary to qualify the name just for the sake of some principle of modularity? This is a valid complaint, and I shall take it as a requirement for the naming scheme later that it should be possible, via controlled defaults or synonyms, to use single words to represent composite command names.

We are now in a position to see how the designer of an object type may choose to offer its functions only within a special mode, or always by direct invocation from the command language (or from programs), or both ways. The modal approach would put only one operation, e.g., "mymail.start," into the definition of the object; this would then include in its semantics the possibility of a dialogue, including data inputs from the user that had the form of commands directed to the mailer in its private language. The direct approach would put the specific operations into the interface, and the two approaches could, if desired, be combined to allow equivalent function to be obtained by either means.

Instances and names

We envisage command languages as being merely users of operations that are defined and implemented in programming languages.[5] (Note that these programming languages do not even have to be abstract type languages. They must just be able to implement callable routines that provide the semantics desired of an abstract type.) An abstract type is used by creating one or more instances of the type and performing permitted operations on these instances.

We must consider here the important distinction between general languages of the abstract type (e.g., CLU,[1] ALPHARD,[2] PLAIN[3]), with potentially multiple instances of a type; and "module" languages (e.g. MODULA-2[4]), which are similar but permit at most one instance of a module. This affects the way that a language specifies the creation and naming of instances; and the naming, at least, is bound to be relevant to the command language user. With a module approach, there is no need to distinguish between the name of the module and the name of the instance, whereas with an abstract type a command must indicate the instance and not the type (e.g., "mymail" and not "mailer_type"). I prefer the abstract type of approach as a more natural way of treating multiple instances than having multiple modules whose equivalence (apart from name) has to be determined by inspection. However, the difference to most command language users would normally be negligible, since they would just know what names to apply to the objects they wanted to use without worrying how they were derived.

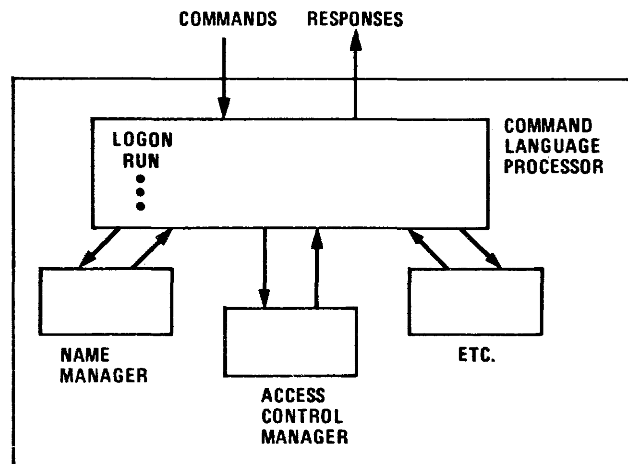


Figure 2—A system populated by abstract objects

Advantages and disadvantages

I have already touched on the ways in which this form of modularity can work to the advantage of the user: in making large systems more comprehensible by dividing them into manageable pieces, in encouraging well-specified interfaces, and in providing a route to a useful initial standard rather than none at all. Beyond this, it will allow for experience to be gained with other types of object before deciding whether they are ripe for standardization. This will include experimentation with alternative ways of doing similar things, an approach that is much harder to manage in a monolithic language.

It is time now to look for possible disadvantages. I have already stated that I intend to develop a naming scheme to make it possible to avoid the burden of name qualification. But, going below the syntax to the underlying modularization, what if the abstraction made by the system designer does not coincide with that which is most natural to the user? For example, commands to filer, editor, formatter, and photocomposer types of object may appear to the user to be all operating on an object of type "document." The use of synonyms will provide a superficial solution here, and this may often be sufficient. The simplest method is to replace all name qualification by single command names, so that the modularity is no longer visible. The more ambitious method, that of introducing differently grouped name qualifications, could be just as easy as a matter of naming; but the transformation of the semantics to fit the new conceptualization could be a very difficult exercise. The net result is that one has to live to some extent with the structure of a given system in order to retain one's sanity. One can always do at least as well as with the monolithic description, and usually much better, but one cannot easily produce arbitrary reconceptualizations.

A slightly more discomfiting criticism of this form of abstraction is its inherent asymmetry. One object is essentially selected as the principal operand of an operation, and the others are passed to it in subsidiary roles. This is often a natural reflection of relative importance, but not always. If an operation is designed to transfer a piece of information from an object of type x to an object of type y, in which abstract type should it be provided? In principle, a new type could be constructed, say xy, containing all symmetrical operations on the types x and y. However, in practice the combinatorial requirement for such types, and, even worse, for instances of an xy object for each pair of instances of x and y, would count heavily against such a solution. This is not likely to be a major problem for command language users; but, where it might prove difficult to remember which type a particular operation was associated with, a better solution would be to offer the corresponding operation in the other type(s) also.

Along a different line, a disadvantage of adopting an initial command language that is not all-inclusive is that "the good is the enemy of the best": it may delay or prevent the arrival of the universal command language. This objection had to be included for the pleasure of refuting it, since the history of efforts to date supports the view that the search for an impressively comprehensive command language is itself the principal agent of delay and appears capable of indefinitely delaying the production of anything whatsoever.

Finally, I shall surely be assailed for believing that it is sufficient for the command language to be able to perform operations on objects without also being able to construct procedures, to iterate and branch and declare its own variables, and generally to aspire to be a programming language. The defense here is that it is extremely difficult to design a good programming language, so the choice would be between a poor result and even longer delays; and that a new programming language is not needed for the purpose of putting logic around commands—existing programming languages are already extensively known and supported and can generally serve quite well. Improvements in language and implementation may be necessary in some cases for interactive use and convenient access to system function, but the payoff will then be enormous in terms of the avoidance of artificial discontinuities between what can be expressed in the command language and what in the programming language. One method of embedding system function in programming languages has been discussed in detail elsewhere.[6]

Taking stock

The results so far are mildly encouraging in improving comprehensibility, quite strong in delegating areas of disagreement to particular abstract types and in establishing a boundary between command languages and programming languages, and far too successful (for some tastes) in limiting the function included in the command language per se. Everything apart from "run" could be delegated to the abstract types, and this approach has been successfully embodied in the Lilith machine.[7] A command language standard with this single command in a suitable embodiment ought to be achievable, but it might be considered to miss an opportunity to introduce more widespread uniformity. Are there other operations which should not be left to designers of individual abstract types? If so, we should be prepared to admit them, and we accordingly propose slightly more generous criteria for inclusion of functional capability.

CRITERIA FOR INCLUSION

A particular functional capability should be considered for inclusion in a standard command language if and only if

1. It provides one of the following:
   a. The means for a user to begin or end a session in which commands are issued to a system
   b. A general means of invoking operations conceived as acting on abstract types of object whose semantics are not defined within the command language
   c. Action which it is desirable to define uniformly for all or most abstract types accessible from the command language
2. It is not already available in the command language, except perhaps with extreme inconvenience.

We proceed now to consider some candidates for inclusion. Invocation has to be present, and we shall not attempt to deal here with questions of its syntax or treatment of operand

types. Otherwise, apart from LOGON and LOGOFF, we shall be dealing with functionality which is common to multiple abstract types. Operations will be named in capitals where they correspond to potential explicit commands in a command language.

Possible ways of treating operations applicable to multiple types are

- A recursive subtype structure (cf. SIMULA[8]), where the subtype inherits the operations of the parent type
- A restricted two-level structure with all specific types known to the command language nested within a general object type
- Separate abstract types, some of whose operations are implicitly involved in command language operations such as "run"

The last method is selected for its simplicity and adequacy for our purposes.

LOGGING ON and OFF

The question of the inclusion of some form of LOGON and LOGOFF has been prejudged in its favor by one of the criteria. The reason for this strong line is the importance of enabling users to get started in a simple, standard way rather than receiving a disastrous first impression of the complexity and idiosyncrasies of computer systems. Symmetrically, but of less importance, users should be able to take their leave without embarrassment.

However, it is necessary to cater to the spectrum from the one-person computer that does not require any identification of the user to the highly protected system that has elaborate and specialized authorization tests. This can be done by allowing some systems to make LOGON and LOGOFF optional but requiring all to recognize them. The semantics of LOGON will include optional prompting for other forms of authorization if necessary, and LOGOFF will allow for some implementation-defined cleanup.

LOGON interacts with access control (see below) in that before it is performed (in systems that require it) only HELP and LOGON are available. The semantics of LOGON allow it to give the user some initial access rights, obtained from a "user authorization" object. It also interacts with naming (see below) in establishing an initial name space for the user. LOGOFF reverts to the HELP or LOGON situation.

Help

A uniform method of seeking help should be available to command language users so that they do not need too much recursive help in using this facility. The HELP command addressed to the "command language processor" object will sometimes produce information directly related to that object, e.g., how to logon; or influenced by that object, e.g., what menu of commands is available to the user at a given point; or even customized by that object as a result of forming an intelligent model of the user during previous interactions. But much of the information desired will be about the operations that may be invoked on other abstract types, and these types will be required to include "help" operations which may be invoked by the command language processor in order to support its uniform helpfulness. This does not preclude the direct invocation of these "help" operations, or indeed the provision of other more specialized kinds of help for particular abstract types.

Creation and destruction

Creation and destruction are the most fundamental operations common to different object types, serving as a prerequisite for, or veto on, all other operations on an object. Do they satisfy our criteria? The answer is clearly "yes" if uniformity is interpreted loosely and "no" if it is taken strictly. The major problems are with the widely differing parameterization of creation for different object types and the varying semantics of destruction of types of objects that may be shared or may cause cascading destruction of other objects.

We propose that CREATE and DESTROY be grudgingly admitted, with uniformity in the names of the commands and in their interaction with the naming and access control of the objects they deal with; but that they allow for type-dependent parameters and semantics beyond this.

Naming

The "run" operation must be able to resolve names of operations on any abstract type and possibly names being passed as operands. Its semantics become very weak if this resolution is system-defined, and it would be helpful to users of different systems to have a common method of name qualification and aliasing to overcome the need for names to be unique systemwide (or networkwide). The earlier requirement for synonyms and default name qualification can be satisfied by this more general approach, which we accept as satisfying the criteria.

We therefore postulate an abstract type "Name Manager" (NM), with an operation "resolve" implicitly used by the command language "run" and an operation "name" used by abstract types when intended names are passed to them in a "create" operation. A particular instance of an NM is associated with a user in the "user authorization" object, and it provides the initial name environment after a successful LOGON. Otherwise, the NM behaves like any other object accessible from the command language, and explicit operations on it may be invoked in the normal way. The NM abstract type can be defined to correspond to a conventional directory structure, i.e., a tree with additional links to make it a network, since there seems to be reasonable consensus that this is a satisfactory model. The set of explicit operations could be RENAME, REMOVE, EQUATE, and EXPAND (applied to an incomplete, possibly ambiguous, name).

Access control

If a system supports any form of access control, it is desirable to apply it as a uniform scheme across all types of object for it to be effective; so this appears to be another

strong prima facie candidate. With the data abstraction model, it is attractive to base access control on subsetting the permitted operations of the abstract types, with other refinements that we cannot pursue here. A similar approach is proposed to that used for naming: a "check access" operation is implicitly used by the command language "run," and other operations on the "Access Control Manager" (ACM) are explicitly available for direct invocation.

A system that does not wish to support access control can appear to the user as one that allows all operations except the explicit operations on the ACM. It can then use its normal optimized implementation, rejecting the ACM operations but not checking anything else.

We therefore admit access control to the command language, with explicit operations that could be GRANT and REVOKE.

Accounting

Accounting is another function relevant to all types of object. It could be treated similarly to naming and access control, since command language actions implicitly use system resources, and explicit operations could be provided that were directed to the "accounting manager" object. However, the latter operations might not be very widely accessible, and there is little consensus about the best way of charging for consumption of resources. Therefore, the decision proposed here is that the semantics of the command language actions should allow for system-defined accounting to be performed and that the command language should treat any accounting manager as an ordinary object about which it has no special knowledge.

Concurrent invocation

In the absence of a consensus on good language for concurrent invocation, we propose an interim treatment until suitable forms can take their rightful place alongside synchronous invocation in the command language. Particular types of concurrency manager might exist in different systems, with operations like START an operation on some other object, ENQUIRE_STATUS, and WAIT. An alternative approach, available with programming languages that offer concurrency and that support access to the desired operations on objects, is to invoke a processor of such a programming language and express the concurrency requirements in its language.

CONCLUSION

I have discussed the applicability of data abstraction to command languages and arrived at the view that it provides a good conceptual structure for issuing commands to operate on different types of object, leaving the initial definition and implementation of object types to fully-fledged programming languages.

The modularity inherent in this approach suggested some stringent criteria that could be applied to reduce to soluble proportions the problems of designing a potential standard command language. Applying these criteria, we admitted only the following commands:

- "Run" the operations on instances of abstract types
- LOGON and LOGOFF
- HELP
- CREATE and DESTROY
- Implicit "resolve" and "name", and certain explicit commands, to a name manager
- Implicit "check access", and certain explicit commands, to an access control manager

This is not to deny the possibility of or need for standardization of other system functions. On the contrary, it would encourage the timely and efficient consideration of such matters by experts working within a modular structure of separate committees, some of which already exist.

ACKNOWLEDGMENTS

Meetings of IFIP Working Group 2.7 have provided a helpful environment for the development of these ideas.

REFERENCES

1. Liskov, B., R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual. Lecture Notes in Computer Science, 114.* Berlin: Springer-Verlag, 1981.
2. Wulf, W. A., R. L. London, and M. Shaw. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Transactions on Software Engineering,* SE-2(1976), 4.
3. Wasserman, A. I., D. D. Sheretz, M. L. Kersten, R. P. van de Riet, and M. D. Dippé. "Revised Report on the Programming Language PLAIN." *ACM SIGPLAN Notices,* 16 (1981), 5.
4. Wirth, N. MODULA-2. Eidgenössische Technische Hochschule, Zurich, March 1980.
5. Beech, D. "What is a Command Language?" In D. Beech (ed.), *Command Language Directions.* Amsterdam: North-Holland, 1980.
6. Beech, D. "Modularity of Computer Languages." IBM United Kingdom Laboratories, TR 12.190, October 1980.
7. Wirth, N. "Lilith: A Personal Computer for the Software Engineer." *Proceedings of 5th International Conference on Software Engineering,* 1981, pp. 2–15.
8. Dahl, O.-J., K. Nygaard, and B. Myhrhaug. "The SIMULA 67 Common Base Language." Norwegian Computing Center, Oslo, 1968.

Integration of bottom-up and top-down contextual knowledge in text error correction

by SARGUR N. SRIHARI, JONATHAN J. HULL, and RAMESH CHOUDHARI
State University of New York at Buffalo
Amherst, New York

ABSTRACT

This paper presents an efficient method for the integration of two forms of contextual knowledge into the correction of character substitution errors in words of text: bottom-up knowledge in the form of character transitional probabilities and top-down knowledge in the form of a dictionary. The method is a modification of the Viterbi algorithm—which maximizes string a posteriori probability by using character confusion and transitional probabilities—so that only legal strings are output. The algorithm achieves its efficiency by using a trie structure representation of a dictionary in the search process. An analysis of the computational complexity and the results of experimentation with the approach are presented.

I. INTRODUCTION

Computer correction of errors in text is important for flexibility in communication between computers and people. The capabilities of present commercial machines for producing correct text by recognizing words in print, handwriting, and speech are very limited. For example, most optical character recognition (OCR) machines are limited to a few fonts of machine print or to text that is handprinted under certain constraints; any deviation from these constraints will produce highly garbled text. Moreover, human beings perform better than these machines by at least an order of magnitude in error rate, although human performance when perceiving a letter or phoneme in isolation is only comparable to that of commercial machines. This is due to human knowledge of contextual factors like letter (or phoneme) sequences, word dependency, sentence structure and phraseology, style, and subject matter as well as associated skills such as comprehension, inference, association, guessing, prediction, and imagination, all of which take place very naturally during the process of reading and hearing.

It is clear that programs that are able to correct errors in text need to be able to use contextual knowledge about the text as well as knowledge about the likely sources of textual errors. A number of programs for using some form of contextual knowledge in text error correction are described in the literature. Among these one can discern two basic approaches: those that are data-driven, or bottom-up, and those that are concept-driven, or top-down.

Data-driven algorithms for text error correction proceed by refining successive hypotheses about an input string. Examples of such an approach are those that use a statistical representation of contextual knowledge—e.g., a Markovian model of text source, which consists of a set of tables representing the probability of occurrence of a letter, given that a set of letters have occurred previously.

Concept-driven algorithms proceed with an expectation of what the input string is likely to be and proceed to fit the data to this expectation. Examples are algorithms that use implicit or explicit representations of dictionaries, syntax, and semantics.

In what follows we describe an algorithm that effectively merges a bottom-up refinement process based on the use of transition probabilities with a top-down process based on searching a trie-structure representation of a dictionary. The algorithm is applicable to text containing an arbitrary number of character substitution errors, such as that produced by OCR machines; thus the method excludes character deletion, insertion, and transposition errors that a typographical error correction algorithm needs to consider.

II. VITERBI ALGORITHM

The Viterbi algorithm (VA) is a method of computing the most probable word that could have caused the observed word. This probability is computed by taking into account the probabilities of confusion between letters and the probabilities of cooccuring n-grams.

Let the observed word be $X = X_0X_1 \ldots X_mX_{m+1}$ where X_0 and X_{m+1} are the delimiters of the m-letter word. The probability that a word $Z = Z_0Z_1 \ldots Z_mZ_{m+1}$ could have caused X is expressed by using Bayes decision theory as

$$P(Z/X) = [P(X/Z)^*P(Z)]/P(X)$$

where $P(X/Z)$ is the probability of observing X when Z is the true word, $P(Z)$ is the a priori probability of Z, and $P(X)$ is the probability of string X. Since $P(X)$ is independent of Z, the word Z that maximizes $P(Z/X)$ can be determined by maximizing the expression

$$G(X/Z) = \log P(X/Z) + \log P(Z).$$

Storing the $P(X/Z)$ distribution in memory is impractical because of the large number of combinatorial possibilities for X and Z. If we assume conditional independence among $X_0, X_1, \ldots, X_{m+1}$, then

$$\log P(X/Z) = \sum_{i=0}^{m+1} \log P(X_i/Z_i).$$

According to this assumption, the observed letters are independent of each other, which is valid for printed text but not necessarily so for cursive script. The probability $P(X_i/Z_i)$ is the probability of observing letter X_i when the true letter is Z_i, which is called the confusion probability.

If we assume that words are generated by an nth-order Markov source, then the a priori probability $P(Z)$ can be expressed as

$$P(Z) = P(Z_{m+1}/Z_{m+1-n} \ldots Z_m) \ldots P(Z_1/Z_0)^*P(Z_0),$$

where $P(Z_k/Z_{k-n} \ldots Z_{k-1})$ is called the nth-order transitional probability, i.e., the probability of observing Z_k when the previous n letters are $Z_{k-n} \ldots Z_{k-1}$.

In the case of $n = 1$,

$$P(Z) = P(Z_{m+1}/Z_m) \ldots P(Z_1/Z_0)^*P(Z_0)$$

and the word Z with maximum a posteriori probability is one that maximizes

$$G_1(X,Z) = \sum_{i=1}^{m+1} \log P(X_i/Z_i) + \log P(Z_i/Z_{i-1})$$

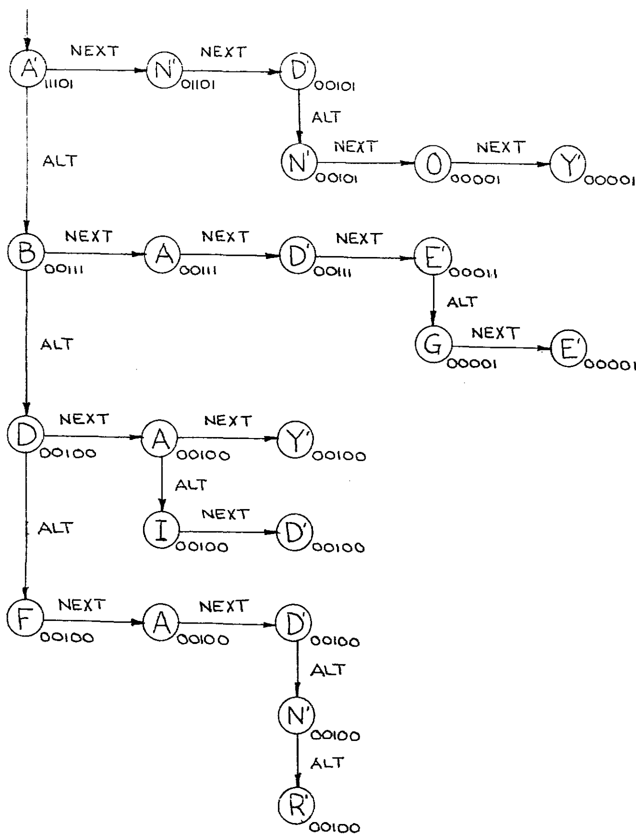


Figure 1—Trie structure representation of the dictionary A, AN, AND, ANN, ANNOY, BAD, BADE, BADGE, DAY, DID, FAD, FAN, FAR. Each node has a 5-bit word-length indicator array and word termination is indicated by a quote mark.

where it is assumed that $P(X_0/Z_0) = P(X_{m+1}/Z_{m+1}) = 1$, i.e., the delimiter symbol is perfectly recognized. In the case of $n = 2$, the corresponding expression is

$$G_2(X, Z) = \sum_{i=1}^{m+1} \log P(X_i/Z_i) + \log P(Z_i/Z_{i-2}Z_{i-1}),$$

where

$$P(Z_1/Z_{-1}Z_0) = P(Z_1/Z_0).$$

The VA is a method of finding the word Z that maximizes $G_i(X,Z)$ without having to compute all 26^m possible $G_i(X,Z)$. The method is based on a dynamic programming formulation, which leads to a recursive algorithm. Essentially, if L_j, $j = 1, \ldots 26$, represents the jth letter of the alphabet, then $\max[G_i(X_1 \ldots X_k, Z_1 \ldots Z_{k-1}Z_k = L_j)]$ over all possible values of $Z_1 \ldots Z_{k-1}$ can be computed trivially if we know the 26 values corresponding to $\max[G_i(X_1 \ldots X_{k-1}, Z_1 \ldots Z_{k-2}Z_{k-1} = L_r)]$, $r = 1, \ldots, 26$ over all possible values of $Z_1 \ldots Z_{k-2}$. This formulation reduces the complexity of the algorithm to $O(26^2)$, which is superior to $O(26^m)$, required by the exhaustive search. The algorithm can be viewed as a shortest-path algorithm through a directed graph of $26 \times m$ nodes, called a trellis. The negative of the log transitional probabilities is associated with the edges of the trellis, and the negative log confusion probabilities are associated with the nodes. The cost of a path is then the sum of all the edge and node values in the path.

Some generalizations of the VA have used either a fixed number of alternatives that is less than 26, called the modified Viterbi algorithm (MVA),[1] or a variable number of alternatives[2] for each Z_k. These alternatives can be determined by the letters that have the highest confusion probability.

III. THE TRIE

In contrast to the usual lexicographical organization such as that used in a desktop dictionary, several alternative structures have been described.[3] The decision to use such an alternative is based on the search strategy of the text manipulation algorithm and the memory available.

One of the ways to represent a dictionary and the one used here is the trie. The trie and its variations are discussed at length by Knuth,[4] and text enhancement systems that use it as their basis are described by Muth and Tharp[5] and by Kashyap and Oommen.[6]

The trie considers words to be ordered lists of characters, elements of which are represented as nodes in a binary tree. Each node has five fields: a token, CHAR; a word-length indicator array of bits, WL; an end-of-word tag bit; and two pointers labeled NEXT and ALTERNATE.

A node is a NEXT descendant if its token follows the token of its father in the initial substring of a dictionary word. It is an ALTERNATE descendant if its token is an alternative for the father's, given the initial substring indicated by the most immediate ancestor, which is a NEXT descendant (see Figure 1). Without loss of generality it is required that the lexical value of the token of each ALTERNATE descendant be greater than that of its father. The end-of-word bit is set if its token and the initial substring given to reach the token compose a complete dictionary word. The mth bit of the word-length-indicator array is set if the token is on the path of an m-letter word in the trie.

If a dictionary has been given as a trie, with fields initialized as above, the following function determines whether the character ch in a word X of m characters follows an initial substring given by a pointer p to the first possible character following this substring.

```
Function ACCESS-TRIE (var p : ptr; ch : char) : boolean;
begin
    if (p = nil) or (p^. CHAR > ch) or (p^.WL[m] = 0)
      then
          ACCESS-TRIE : = FALSE
      else
         if (p^.CHAR = ch)
           then
             begin
                 p: = p^.NEXT;
                 ACCESS-TRIE : = TRUE
             end
           else
           ACCESS-TRIE: = ACCESS-TRIE
                          (p^.ALTERNATE, ch)
end; (* ACCESS-TRIE *)
```

A version of this function will be used with the proposed bottom-up and top-down algorithm. It is interesting to note that the maximum number of recursive calls for any given initial substring is 25. If it were assumed that for all positions in any string the possible letters were uniformly distributed over all 26 possibilities, the average number of calls would be 12.5. This assumption is clearly unreasonable because of the nature of the trie and the English language itself. For example, with "AMPLIFYIN" as an initial substring, there is only one possibility for the tenth position. This characteristic is reflected in the experimentation discussed in Section VI where the average number of alternatives for all nodes in a sample trie was 1.62.

IV. DICTIONARY VITERBI ALGORITHM

The MVA is a purely bottom-up approach, whose performance may be unacceptable. For example, in experimentation with the MVA,[7] the best overall word correction rate was 46% when second-order word-length and position-independent (WLPI) statistics were used and 20% when first-order WLPI statistics were used. For an efficient contextual postprocessing system, this performance must be improved. One approach to the problem is to use top-down contextual information, in the form of a dictionary of allowable input words, to aid the bottom-up performance of the MVA.

One such method, known as the predictor-corrector algorithm,[8] uses an extension of the Bledsoe-Browning[9] approach. Given a word output by the MVA, it computes a score for the word. A constrained search and computation procedure is then carried out over the dictionary. This is a two-part method, in which the use of dictionary information is distinct from processing by the MVA. In this section an algorithm is proposed that integrates dictionary information with MVA processing. The resultant dictionary Viterbi algorithm (DVA) offers the advantages of a dictionary method in terms of legibility of output and correction rates but shows no increase in order of complexity from the MVA.

The formal statement of the text enhancement procedure based on the DVA follows.

The Algorithm

```
repeat
    GETWORD(X);    (* Read next word X *)
    DICTIONARY-VITERBI(X, Z);
    WORD-OUT(Z);   (* Output word Z *)
until end-of-file;
```

The procedure DICTIONARY-VITERBI, stated below, is for the case of a first-order Markov assumption and a fixed number of alternatives, d, for each letter. This is similar to the MVA and is performed to allow comparison of the complexity of the two algorithms.

Symbols and data structures

$L_1 \ldots L_{26}$ represent symbols A ... Z, and the delimiter symbol is ƀ.

C is a vector of d real numbers called the cost vector.
Q is a vector of d pointers into the trie, initially the root.
S is a vector of d character strings called the survivor vector.
$X = X_1 \ldots X_m$ is the input character string.
$Z = Z_1 \ldots Z_m$ is the output character string.
$\bar{A}$ is a $d \times m$ matrix of alternatives whose columns are labeled $A_1 \ldots A_m$.

Primitive functions

$MAX(a_1 \ldots a_d, u)$ returns the maximum of $\{a_1 \ldots a_d\}$, and the index of the maximum in u.
$CONCAT(s,L_j)$ concatenates character L_j at the end of string s.

```
Procedure DICTIONARY-VITERBI(X1 ... Xm, Z1 ... Zm);
    (*given an m-letter string X = X1 ... Xm as input,
      produce an m-letter string Z = Z1 ... Zm as output*)
    begin
        INITIALIZE(Ā);
        DICTIONARY-TRACE (Ā, C, Q, S, X1 ... Xm);
        Z := SELECT(Ā, C, S);
    end;
```

Procedure INITIALIZE selects the d most likely alternatives for each letter of the input word. This is done by choosing those d letters for which the sum of the log-confusion and log-unigram probabilities is greatest.

Procedure DICTIONARY-TRACE, which follows, returns a set of character strings in Vector S whose costs are defined by Vector C.

```
Procedure DICTIONARY-TRACE(Ā, C, Q, S, X1...Xi);
  begin (*C1, S1, Q1, Q2 are local vectors of d
    elements*)
    if i > 1 then begin
        DICTIONARY-TRACE(Ā, C, Q, S, X1...Xi-1);
        C1 := C; Q1 := Q;
        S1 := S; Q2 := Q;
        for j := 1 to d do begin
          for k := 1 to d do begin
            if ACCESS-TRIE(Q1(k), Ai(j))
              then gk := C1(k) + log P(Xi/Ai(j))
                + log P(Ai(j)/Ai-1(k))
              else gk := -inf end;
            C(j) := max (g1, ... ,gd, u);
                        Q(j) := Q1(u);
            if (C(j) < > -inf)
              then S(j) := CONCAT(S1(u), Ai(j))
              else S(j) := null;
            Q1 := Q2;
            end;
        end
    else begin (*i := 1 *)
      for j := 1 to d do
        if ACCESS-TRIE (Q(j), A1(j))
          then begin
            C(j) := log P(X1/A1(j))
              +log P(A1(j)/ƀ);
            S(j) := A1 (j); end
```

```
        else begin C(j) := -inf; Q(j) := nil;
                S(j) := null; end;
    end;
  end; (*DICTIONARY-TRACE *)
```

Function SELECT returns the most likely word by considering the cost of the transition from the final symbol to the trailing delimiter ƀ when the cost vector C and the survivor vector S are used. If all the values in C are equal to minus infinity, *X* is rejected and a null value is returned.

The integration of the dictionary into the Viterbi algorithm is done in Procedure DICTIONARY-TRACE by maintaining a vector of pointers into the trie. Each element corresponds to a survivor string. At each iteration of the k loop the kth element of this vector is passed to ACCESS-TRIE. If the corresponding initial substring concatenated with the letter indicated by the j index is a valid dictionary string (ACCESS-TRIE is true), the probability calculation is carried out. A weight of minus infinity is given to this alternative when a false value is returned in order to preclude the possibility of non-dictionary words being considered. At some iteration in the j loop, if all attempted concatenations fail to produce a valid dictionary string, the survivor for the corresponding node and its pointer are assigned null values. For some value of i, if all survivors are null, the input word is rejected as uncorrectable. This may happen when less than 26 possibilities are considered for each letter, but it will never happen when all candidates are allowed. This phenomenon is discussed in Section VI.

A variation of the DVA would be to use the pointer to the node that corresponds to the alternative chosen at the last step as a substitute for the explicit maintenance of survivor strings. If the trie included son-to-father pointers, this pointer would allow a path to be traced from the indicated node back to the first level of the trie to retrieve the output string. This would yield storage economy when the number of nodes was less than the number of locations required for the survivor strings because of the additional pointer required at each node.

The above algorithm considers a fixed number of d alternatives for each letter of the input string. A modification of the algorithm to include a variable number of alternatives is as follows. Within procedure INTIALIZE only letters for which the sum of the log-confusion and log-unigram probabilities is greater than an a priori threshold value t are chosen as alternatives.

V. COMPUTATIONAL COMPLEXITY

The complexity of the MVA derived by Shinghal and Toussaint[1] will be used to show the additional computation of the DVA. Only the general case of $n > 1$ and $1 < d < 26$ will be discussed here. The overall computation requirement of the DVA can be expressed as a function of n and d, as shown in the following:

$$D(n, d) = D_a(n, d) + D_p(n, d),$$

where $D_a(n, d)$ is the requirement for the selection of alternatives and $D_p(n, d)$ is the requirement for the path tracing. Since the selection of alternatives remains the same as in the MVA,

$$D_a(n, d) = 26n + n \sum_{j=1}^{\min(26-d,\ d)} (26 - j).$$

Path tracing involves two steps. Step 1 is the path tracing itself and step 2 is the evaluation of the last letter to blank transition. A trie look up τ is defined as the number of comparisons necessary in a call to ACCESS-TRIE. An addition and comparison are defined to equal one unit of computation.

Step 1 requires:

$$d^2(n-1)(\tau+3) + d(\tau+1) \quad \text{units},$$

which is an upper bound occuring when all trie look-ups are successful.

Step 2 requires:

$$(2d-1) \quad \text{units}.$$

Therefore,

$$D_p(n,d) = d^2(n-1)(\tau+3) + d(\tau+1) + (2d-1).$$

Therefore, the complexity of the DVA is:

$$D(n,d) = 26n + n \sum_{j=1}^{\min(26-d,d)} (26-j) + d^2(n-1)(\tau+3) + d(\tau+1) + (2d-1).$$

The complexity of the MVA[1] is:

$$V(n,d) = 26n + n \sum_{j=1}^{\min(26-d,d)} (26-j) + 3d^2(n-1) - nd + (2d-1).$$

Comparison of V(n,d) and D(n,d) shows no change in order of complexity, with both expressions increasing linearly as a function of n and quadratically as a function of d. The experimentally derived value of the average number of alternatives at a trie node of 1.62 suggests only an increase in the coefficient of d^2.

VI. EXPERIMENTAL RESULTS

To determine the efficiency and performance of the DVA and to compare this with the MVA, a data base was established and experiments were conducted.

English text in the Computer Science domain (Chapter 9 of *Artificial Intelligence,* P.H. Winston, Addison-Wesley, 1977) containing 6372 words was entered onto a disc file. Unigram and first order transition probabilities were estimated from this source. A model reflecting noise in a communications channel was used to introduce substitution errors into a copy of this text and confusion probabilities were estimated from this source. The same probability tables were used for all experiments.

A dictionary of 1724 words containing 12231 distinct letters was extracted from this text and a trie was constructed for use by the DVA. There were 6197 nodes in the trie and the aver-

age number of alternates for all nodes was 1.62. The frequency histogram of alternate path lengths is extremely skewed, with 4805 nodes having no alternatives but itself (path length 1) and 701, 240, and 128 nodes having alternate path lengths of 2, 3, and 4, respectively.

An example of input and output text to the DVA follows:

IF WE LOOI AT WHAT HAS PRODUSED LOMPUTER IMTELLIGENCE QO FAR, WE SEE MULTIPLE LAMERS, EACH OF WHICH RESTS ON PRIMITIVES OF THE NAXD TAYFR DOWM, FORMINC A HIERARCFICAL STRUCTURE WITH A GREAT DEAL INTERPOSED BETWEEN THE INTELLIGENT PRPHVEM AND THE TRANSISTORS WHICH ULTIMATELU SUPPODT IT. ALL OF THE CGMPLEXITU OF ONE KEVEL IS SUMMARIZFD ABD DISTILLED DOWN TO A BES SIMPLE ASOMIC NOTIONS WHICH AZE THE PRIMITIVES OE THE NEXT LAMER UP. BUT WITH SO MUCH INSULATIOP, IT CCNNOT POSSMBLY BE THAT THE DETAILFD NATURE OF THE LGWER LEVELS CAN MATTER TO WHAT HAPPENS AFOXE.

IF WE LOOK AT WHAT HAS PRODUCED ******** INTELLIGENCE SO FAR, WE SEE MULTIPLE LAYERS, EACH OF WHICH RESTS ON PRIMITIVES OF THE NEXT ***** DOWN, FORMING A HIERARCHICAL STRUCTURE WITH A GREAT DEAL INTERPOSED BETWEEN THE INTELLIGENT PROGRAM AND THE TRANSISTORS WHICH ULTIMATELY ******* IT. ALL OF THE COMPLEXITY OF ONE LEVEL IS SUMMARIZED AND DISTILLED DOWN TO A BUT SIMPLE ATOMIC NOTIONS WHICH ARE THE PRIMITIVES OF THE NEXT LAYER UP. BUT WITH SO MUCH INSULATION, IT CANNOT POSSIBLY BE THAT THE DETAILED NATURE OF THE LOWER LEVELS CAN MATTER TO WHAT HAPPENS ABOVE.

The output was produced by the DVA using a fixed number of alternatives with the depth of search set at 6; LOMPUTER, TAYFR, and SUPPODT were rejected; and BES was erroneously corrected to BUT instead of FEW. Rejections could be eliminated by increasing the depth of search, because a dictionary word could be located that could not be located previously because of the constrained nature of the trellis.

The performances of the DVA and the MVA were measured by the percentage of garbled words corrected when both algorithms were run on the same piece of text. A fixed and variable number of alternatives were used in both cases.

To contrast the complexity of the DVA and the MVA, a fixed number of alternatives was used for both algorithms, and the CPU time required to process a fixed input text was used for comparison. The same program was used in all cases, the only differences being those necessary to implement the particular version of the algorithm.

The results of applying the algorithm to the entire random sample of garbled text (of 6372 words) are summarized in Table I. Time figures are CPU seconds on a CDC Cyber 174. The DVA in all cases performed significantly better than the MVA without a dictionary: the best-case correction rate for the DVA was 87%; the corresponding figure for the MVA was 35%. To minimize the cost it is necessary to choose the minimum value of the depth of search (d) or the minimum threshold (t) that give the optimum correction rate. These were found to be 8 and −11, respectively. While the time requirement at optimum performance for the DVA differed by about a factor of 1.7 from the MVA, it is interesting to note that the best performance for the DVA in the variable-alternatives case was achieved at a cost significantly less than that using a fixed number of alternatives.

To show the effects of differing levels of contextual information on performance at the optimum parameter settings, the DVA was run with only top-down information by setting all transition probabilities equal; and the MVA was run without the trie, thus using only the bottom-up information provided by the transition probabilities. The correction rates were 82% and 35%, respectively—both less than the 87% provided by the combination approach.

TABLE I—Results of application of algorithm to entire random sample of garbled text

	Fixed Number of Alternatives					Variable Number of Alternatives			
	DVA		MVA			DVA		MVA	
d	% corr.	time (secs.)	% corr.	time (secs.)	t	% corr.	time (secs.)	% corr.	time (secs.)
1	1	770	1	732	−2	0	473	0	463
2	39	853	23	767	−3	0	491	0	479
3	61	946	29	808	−4	0	517	0	490
4	74	1085	33	858	−5	0	522	0	496
5	81	1256	34	916	−6	11	527	8	496
6	85	1474	35	989	−7	44	593	23	500
7	86	1754	35	1082	−8	76	803	33	614
8	87	2122	35	1175	−9	83	1025	35	695
9	87	2536	35	1287	−10	85	1239	35	769
					−11	87	1668	35	819
					−12	87	1668	35	915
					−13	87	1669	35	922

VII. CONCLUSIONS

A new algorithm for text enhancement has been presented that merges processing by the Viterbi algorithm with dictionary information stored in a trie. The results of experimentation with this algorithm were described; they show a correction rate significantly greater than counterparts of the algorithm that do not use a dictionary. An expression for the complexity of this algorithm has been derived and compared with that of one of its counterparts; it shows no increase in the order of complexity due to the addition of the trie. Because of its superior performance, this algorithm is suggested as a low-level word hypothesization component in a system focusing global contextual knowledge on the text enhancement problem.

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation Grant IST-80-10830. We wish to thank Addison-Wesley for permission to use portions of *Artificial Intelligence,* by P.H. Winston, in the experimentation. Thanks are also due to C.T. Chen for the program to build the trie structure from its word list.

REFERENCES

1. Shinghal, R., and G. T. Toussaint. "Experiments in Text Recognition with the Modified Viterbi Algorithm." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1 (1979), pp. 184–192.
2. Doster, W., and J. Schurman. "An Application of the Modified Viterbi-Algorithm in Text Recognition." *Proceedings of International Conference on Pattern Recognition,* Miami, Florida, December 1–4, 1980. Piscataway, New Jersey: IEEE Computer Society, 1980.
3. Peterson, J. L. "Computer Programs for Detecting and Correcting Spelling Errors." *Communications of the ACM,* 23 (1980), pp. 676–687.
4. Knuth, D. E. *Sorting and Searching.* The Art of Computer Programming, vol. 3. Reading, Massachusetts. Addison-Wesley, 1973.
5. Muth, F. E., and A. L. Tharp. "Correcting Human Error in Alphanumeric Terminal Input." *Information Processing and Management,* 13 (1977), pp. 329–337.
6. Kashyap, R. L., and B. J. Oommen. "An Effective Algorithm for String Correction Using Generalized Edit Distances." *Information Sciences,* 23 (1981), pp. 123–142.
7. Srihari, S. N., and J. Hull. "Experiments in Text Recognition with the Binary N-Gram and Viterbi Algorithms." SUNYAB/CS Technical Report 184, Department of Computer Science, State University of New York at Buffalo, July 1981.
8. Shinghal, R., and G. T. Toussaint. "A Bottom-up and Top-down Approach to Using Context in Text Recognition." *International Journal of Man-Machine Studies,* 11 (1979), pp. 201–212.
9. Bledsoe, W. W., and J. Browning. "Pattern Recognition and Reading by Machine." In L. Uhr (ed), *Pattern Recognition.* New York: Wiley, 1966, pp. 301–316.

DIALOGUE: Providing total terminal independence*

by DAVID VASKEVITCH
Standard Software Limited
Toronto, Ontario

ABSTRACT

A software tool, called DIALOGUE, makes application programs completely terminal-independent so that they can be used from both formatted screens and character mode terminals. The independence is achieved by providing programmers with a high-level record definition language for describing the data. This language isolates the programmer from the details of terminal interaction so they can be automatically handled in the most appropriate fashion at execution time.

To support various terminal types most effectively, DIALOGUE may generate a different interface for each brand of terminal. A unique aspect is its typewriter interface which supports character mode terminals so that they are indistinguishable (to the application) from formatted screen terminals. Devices such as light pens and OCR readers are also supported.

Copyright 1981, Standard Software Limited, Toronto, Ontario, Canada. Permission granted to publish in *AFIPS Conference Proceedings, 1982 National Computer Conference.*

1.0 INTRODUCTION

In a transaction-oriented environment, formatted screen terminals offer the potential of a user interface superior to that which has been historically possible.

Special control characters and escape sequences must be sent from the computer to the terminal to take advantage of screen formats.

This raises several issues:

1. Writing programs to support a formatted screen is technically complex.
2. Terminals made by different vendors are incompatible with each other.
3. Older terminals and hard-copy terminals do not support screen formats.

The result is that using formatted screens is complex, locks the user into particular equipment, and renders existing terminals useless.

A software tool, called DIALOGUE, solves the problems associated with formatted screens by doing the following:

1. Providing a high-level interface that makes it easy for the applications programmer to develop user interactions that use screen formats.
2. Supporting a variety of formatted screen terminals so that no lock-in takes place.
3. Supporting character mode terminals so that the application cannot distinguish them from screens, ensuring that any application can be accessed from any terminal.

2.0 THE RECORD DEFINITION LANGUAGE

The central concept in DIALOGUE is that transactions are described in terms of records and that the description occurs at a high enough level of abstraction that the details associated with terminal support are concealed. Later, DIALOGUE can take a record description and decide how best to interact with a user based on the facilities at the user's terminal.

A record definition consists of two components:

Form: The form definition describes the fields making up the record, along with "decorations," such as titles that make the record more understandable.

Commands: Associated with every form is a set of commands representing the valid actions for a user. A form may be associated with several command sets over time (one at a time). DIALOGUE does not actually execute the commands; it uses them to establish valid function keys (at a screen) and signals the application when a command has been entered.

2.1 Describing Records

An example of a record definition is shown below. Several key points will be of interest to most readers:

Fields: The most important elements in the record are fields. Each field has a "prompt" and a "value" and various other optional qualifiers.

Edit Checks: Both alphabetic and numeric fields are supported, along with various other edit checks and transformations (such as shifting to upper case, e.g., SHIFTED).

Groups: Fields may be grouped for users' convenience. On the screen, groups of fields are surrounded by boxes (when possible). On the typewriter special commands allow groups of fields to be skipped in one step.

Syntax: The definition language is free format. Special care was taken to make statements self-identifying so that delimiters, such as semicolors, are not required. All keywords may be abbreviated as long as they are unique.

One of the key design criteria for the definition language was that it be easy to use for both programmers and end users. In this way, record definitions could serve as part of the design interface between an end user and an analyst. The use of English words and the simple syntactic rules have helped make this possible. (See Figure 1.)

2.1.1 Automatic positioning

When describing a form to DIALOGUE, the programmer does not need to specify where the fields and titles are to be placed on the screen. DIALOGUE will automatically position the fields so that they look good to the human eye—a tricky proposition.

The rules used to lay out the form consist of a set of heuristics dealing with the eye's tolerance for misalignment along vertical sight lines. The rules work completely about 80% of the time; and when they do not work, usually only one or two fields need to be adjusted. Naturally, the programmer can override DIALOGUE for one or more fields at any time.

Automatic positioning is important for three reasons:

- Terminal independence—Different-size screens (e.g.,

```
FORM 'MAILFORM' (Comments are enclosed in brackets).
TEXT 'MAILING LIST QUERY' CENTER UNDERLINE.
GROUP 'IDENTIFICATION' CENTER
  FIELD 'SURNAME' ALPHA 20 MANDATORY SHIFTED
  FIELD 'NAME' ALPHA 15
  FIELD 'INITIALS' ALPHA 3
GROUPEND
GROUP 'ADDRESS' CENTER INVERSE BLINK (A STANDOUT).
  FIELD 'CITY' ALPHA 20 NOCLEAR
  FIELD 'STATE' ALPHA 20 NOCLEAR
  FIELD 'COUNTRY' ALPHA 20 NOCLEAR
                    SELECT 'CANADA USA'
GROUPEND
FIELD 'AGE' NUMERIC 2 RANGE 20..45
END

COMMANDS 'MAIL'
  NAME 'ADD' KEY 1 KEY 9 EXPLANATION "Add record to
  database"
  NAME 'QUERY' KEY 2 POINTER EXPLANATION "Retrieve
  record"
  NAME 'END EXIT QUIT LUNCH' KEY 3 NOREAD EXPLA-
  NATION "End of Program"
END
```

Figure 1—A sample record

24×132 and 33×80) may contain similar amounts of information and yet require totally different screen layouts. In addition, the presence or absence of boxes and display attribute characters may require differing field positioning from one terminal to another.

- Programmer productivity—Calculating field positions is time-consuming, tedious, and error-prone. Furthermore, the addition or deletion of a field may make it necessary to reposition all subsequent fields.
- User readability—Not having to specify x and y coordinates makes a form definition easier to read (and write) for the end user.

On balance, of course, automatic positioning is effective only if it does produce good-looking forms; and experience has shown that it does.

2.2 Commands and Function Keys

Given a form definition, a transaction can still not be completed until the user enters a command signaling the application that a record is ready to be processed. Commands are defined in sets. An example of a command set was shown as part of the "Mailform". Command sets may be written as part of a total transaction definition (Form and Commands) or separately. When written separately, the Form and the Command set become associated by application-level subroutine calls at execution time.

At a screen, a command becomes associated with one or more function keys (e.g., QUERY = F1 or F9), while at a character mode terminal the command is invoked by name (e.g., "QUERY").

3.0 THE SCREEN INTERFACE AND DEVICE INDEPENDENCE

At a formatted screen DIALOGUE interprets the record definition automatically to generate a form on the screen based on the capabilities of the terminal. In providing this degree of device independence two key design principles were followed:

- GREATEST COMMON MULTIPLE—One way to support a variety of devices involves the lowest-common-denominator approach—i.e., support only features found on all terminals; the more terminals handled this way, the fewer the features supported. DIALOGUE takes the opposite approach: an honest attempt is made to support all the useful features found on each type of terminal. In large part, this is possible because the Record definition language leaves most of the decisions about form presentation to DIALOGUE.
- FEATURE INDEPENDENCE—Programmers should not be able to build feature dependencies into applications. Thus, when a feature is supported that is not universally present on terminals, it is always supported in such a way that applications are not restricted by the absence of the feature. A particularly striking example is given later in the discussion of pointers.

3.1 Display Enhancements, Edit Checks, and Boxes

The visible appearance of the form on the screen is established through a series of control characters and escape sequences which call on various features found in the terminal. These features include

Display Enhancements: Fields (and other areas) may be highlighted by using display enhancements, such as INVERSE VIDEO, DIM, UNDERLINE, SECRET, BLINKING. These may be used alone or in combination, and they may be used to show a field in its normal state (e.g., the "blanks" in a form) or to flag errors (e.g., BLINKING). When available, unusual enhancements, such as color, may be used as well (e.g., flag a field in RED). Installations may choose the best enhancements for *each* terminal type to take advantage of its features.

Edit Checks: When possible, the terminal is asked to enforce edit checks directly, reducing the load on the computer and providing the user with more immediate feedback. However, if the terminal does not support an edit check, or supports it incompletely, it does not matter, because DIALOGUE will perform it instead.

Boxes: When supported by the terminal, boxes are always drawn around groups of fields. When available, vector drawing and repeat instructions are used to draw the boxes faster.

Pages: Multiple pages of memory are automatically used to store forms for reuse. This

feature can substantially improve response time in multiform transactions.

3.2 *Function Keys*

On a formatted screen, the function key is normally the user's only way of signaling the computer (normal keys typed at the keyboard update the screen without going through the communications interface). The current command set, established by the application, determines which function keys are valid at any point.

When a valid function key is pressed, DIALOGUE performs all its edit checks and flags any fields that do not pass. If this occurs, the user must correct the flagged fields and pick a function key again. To provide an escape mechanism, the programmer may establish "NOREAD" keys (e.g., END).

Once a record passes the edit checks, it is sent back to the application, along with a signal indicating which key was pressed. The default signal for a given command is the number of the first key specified (e.g., ADD = 1, QUERY = 2, END = 3).

3.3 *Pointers: Light Pens and Touch-Sensitive Screens*

A pointer is a device that selects a position on the screen and signals the computer. The signal is essential to the definition, since otherwise the computer would have no way of distinguishing selected positions from intermediate ones. Light pens, touch sensitive screens, joysticks and mice are all examples of pointers.

DIALOGUE supports the pointer by treating it as a form of function key—a strange definition at first. In the example, the QUERY command could be signified by a pointer (or by Key 2).

Normally, when a function key is depressed, DIALOGUE makes the position of the *cursor* available to the application, along with the record and the signal. When a pointer is used, the position of the *pointer* is returned instead. Except for the fact that the user must position the cursor separately before pressing the function key, there is no difference between the pointer and the key.

Thus, when available, the pointer is supported fully; but otherwise the function keys may be used instead, and the application does not distinguish the two cases (although it can if necessary).

4.0 THE TYPEWRITER INTERFACE

Any terminal that is not supported as a formatted screen is considered to be a typewriter by DIALOGUE. Typewriters include

1. Hard-copy terminals
2. Dumb screens (e.g., glass teletypes, word processors, etc.)
3. Screens operating at speeds too low for forms (e.g., 300 baud)
4. Screens not currently supported in forms mode

Users may also choose to run in typewriter if they prefer it.

The typewriter proceeds by prompting the user for each field (in a fashion familiar to any timesharing user). It should be immediately obvious that this process alone provides functional equivalence to the screen, because the end user can enter all the fields in a record using both interfaces (typewriter and screen). Further, the application cannot distinguish between the two interfaces: in either case it receives a complete, edit-checked data record.

The typewriter interface must do more than provide a means for entering data; it must allow the user to quickly and easily modify previously entered fields, as would be possible on the screen. Furthermore, it must allow the user to request a formatted display of his/her context (the record) and provide facilities for expert users to speed up their interaction. The mechanism used to accomplish this is the typewriter command.

4.1 *Typewriter Commands*

DIALOGUE's typewriter interface allows the user to enter a command at any time. Commands are distinguished from field values by an installation-defined unique character, typically the period. Special care is taken to ensure that this character can still be used in other contexts (e.g., entering numbers), since otherwise it would be impossible to choose a character without conflict.

Commands are divided into two categories: internal (or DIALOGUE) commands directed to DIALOGUE, and application (or external) commands that provide the user with function keys.

Help facilities allow the user to see a list of commands (internal, external, or both) and ask for an explanation of each. The previous example showed how explanations are specified for application commands. Commands may always be abbreviated as long as the abbreviation is unique within the current command set.

4.1.1 Typewriter Commands

A set of approximately 35 commands provides the user with complete facilities for examining and modifying records as and after they are entered. For example, when receiving an error message, the user can display the entire record (e.g., DISPLAY), modify a field (e.g., MODIFY SURNAME), and then reenter the record using an application command.

Special provision is made for the first-time user, who may not know any commands. First, the programmer can designate a default application command which is used if the user simply enters all the fields in a record and "falls off the end." Second, if one or more fields are then flagged, DIALOGUE will cycle the user through the flagged fields showing him the value in each (e.g., SURNAME (WASHINGTON) -). The user can keep the value or replace it by typing in a new one. When the user has cycled through all the flagged fields, the record is re-entered for him/her. Thus the user need not know commands or have to reenter the entire record for one mistake.

4.1.2 Advanced Commands

Several advanced facilities can make the typewriter interface uniquely efficient for expert users:

Automatic Responses:	If a field does not change, the user can establish an automatic response with the "remember" command (e.g., REMEMBER SEATTLE). Once an automatic response is set up, the associated field literally disappears from the transaction so that the user no longer needs to deal with it.
Resequencing:	Using a single command (DETOUR), users can rearrange the sequence of fields to please themselves. The application still sees the original record and field sequence.
Typeahead:	DIALOGUE's typeahead feature allows the user to anticipate any number of fields and enter their values, all at once, separated by commas (e.g., WASHINGTON, GEORGE, H). Once a field is anticipated by the user, its prompt is suppressed. This allows very terse input, since the user can enter any amount of information at any time. Typeahead may cross record *and* program boundaries.

4.2 Relative Efficiency

Although formatted screen mode is always easier to use, typewriter can actually be more efficient for the expert user because

1. Extraneous or constant fields can be totally suppressed with automatic responses.
2. Input may occur in any sequence as a result of detours.
3. Multiple records can be entered at one time using typeahead.

One net result of the efficiency of the typewriter is that expert users may use it in preference to block mode. This ensures that screen formats can be designed for ease of use, knowing that the typewriter is there for experts.

5.0 ALTERNATE INPUT DEVICES

A variety of devices may be attached to a terminal, allowing input alternatives to the keyboard. These include OCR wands, bar code readers, magnetic-stripe readers, and voice recognition units. These are essentially "field input" devices that can cause one field at a time (typically) to be entered and are known as readers.

DIALOGUE allows a field to be designated as READABLE. If a form contains a readable field, the reader can be used if present; otherwise the keyboard is used. If only one field is readable, reader input is always placed there. If several fields are readable, the cursor position is used to determine where to place a reader input.

This scheme allows readers to be used when present without either requiring their presence or requiring special application code. A particularly interesting scenario can even be imagined involving both a touch-sensitive screen (a pointer) and a voice recognition device (reader).

6.0 USING DIALOGUE

To the programmer DIALOGUE has three major components:

Forms:	Form definitions are usually written in edit files. These files are then referenced by name in the application. This degree of separation allows simple form changes to be made without even recompiling the application. In addition, form definitions may also be generated programmatically by the application.
Commands:	Although usually written as part of the form definition, command sets may be established separately.
Subroutines:	The subroutine gives the programmer control over the transaction. Most of the subroutines have self-explanatory names; the key ones are listed below: SETUPFORM (editfilename) READRECORD (data-area) WRITERECORD (data-area) ERROR (fieldname, message) MESSAGE (message) Other routines provide explicit cursor control and other detailed functions used less frequently than those above. Great care was taken to ensure that most work could be done with fewer than 10 subroutines.

A particularly attractive aspect of DIALOGUE is its natural-language structure. It appears equally attractive to programmers in COBOL, FORTRAN, Pascal or even BASIC. Unlike many other terminal handlers, DIALOGUE has no bias toward any one language.

7.0 CONCLUSION

DIALOGUE has been running in a commercial environment on the TANDEM computer for about two years and has been very successful there. Over 20 terminal types are supported, and both screen and typewriter mode are in active use. Both end users and programmers seem to like DIALOGUE, and the concept of terminal independence has proved workable.

The star user interface: an overview

by DAVID CANFIELD SMITH, CHARLES IRBY, and RALPH KIMBALL
Xerox Corporation
Palo Alto, California
and
ERIC HARSLEM
Xerox Corporation
El Segundo, California

ABSTRACT

In April 1981 Xerox announced the 8010 Star Information System, a new personal computer designed for office professionals who create, analyze, and distribute information. The Star user interface differs from that of other office computer systems by its emphasis on graphics, its adherence to a metaphor of a physical office, and its rigorous application of a small set of design principles. The graphic imagery reduces the amount of typing and remembering required to operate the system. The office metaphor makes the system seem familiar and friendly; it reduces the alien feel that many computer systems have. The design principles unify the nearly two dozen functional areas of Star, increasing the coherence of the system and allowing user experience in one area to apply in others.

INTRODUCTION

In this paper we present the features in the Star system without justifying them in detail. In a companion paper,[1] we discuss the rationale for the design decisions made in Star. We assume that the reader has a general familiarity with computer text editors, but no familiarity with Star.

The Star hardware consists of a processor, a two-page-wide bit-mapped display, a keyboard, and a cursor control device. The Star software addresses about two dozen functional areas of the office, encompassing document creation; data processing; and electronic filing, mailing, and printing. Document creation includes text editing and formatting, graphics editing, mathematical formula editing, and page layout. Data processing deals with homogeneous databases that can be sorted, filtered, and formatted under user control. Filing is an example of a network service using the Ethernet local area network.[2,3] Files may be stored on a work station's disk (Figure 1), on a file server on the work station's network, or on a file server on a different network. Mailing permits users of work stations to communicate with one another. Printing uses laser-driven xerographic printers capable of printing both text and graphics. The term *Star* refers to the total system, hardware plus software.

As Jonathan Seybold has written, "This is a very different product: Different because it truly bridges word processing and typesetting functions; different because it has a broader range of capabilities than anything which has preceded it; and different because it introduces to the commercial market radically new concepts in human engineering."[4]

Figure 1—A Star workstation showing the processor, display, keyboard and mouse

The Star hardware was modeled after the experimental Alto computer developed at the Xerox Palo Alto Research Center.[5] Like Alto, Star consists of a Xerox-developed high-bandwidth MSI processor, local disk storage, a bit-mapped display screen having a 72-dot-per-inch resolution, a pointing device called the mouse, and a connection to the Ethernet. Stars are higher-performance machines than Altos, being about three times as fast, having 512K bytes of main memory (vs. 256K bytes on most Altos), 10 or 29M bytes of disk memory (vs. 2.5M bytes), a 10½-by-13½-inch display screen (vs. a 10½-by-82-inch one), 1024 × 808 addressable screen dots (vs. 606 × 808), and a 10M bits-per-second Ethernet (vs. 3M bits). Typically, Stars, like Altos, are linked via Ethernets to each other and to shared file, mail, and print servers. Communication servers connect Ethernets to one another either directly or over phone lines, enabling internetwork communication to take place. This means, for example, that from the user's perspective it is no harder to retrieve a file from a file server across the country than from a local one.

Unlike the Alto, however, the Star user interface was designed before the hardware or software was built. Alto software, of which there was eventually a large amount, was developed by independent research teams and individuals. There was little or no coordination among projects as each pursued its own goals. This was acceptable and even desirable in a research environment producing experimental software. But it presented the Star designers with the challenge of synthesizing the various interfaces into a single, coherent, uniform one.

ESSENTIAL HARDWARE

Before describing Star's user interface, we should point out that there are several aspects of the Star (and Alto) architecture that are essential to it. Without these elements, it would have been impossible to design a user interface anything like the present one.

Display

Both Star and Alto devote a portion of main memory to the bit-mapped display screen: 100K bytes in Star, 50K bytes (usually) in Alto. Every screen dot can be individually turned on or off by setting or resetting the corresponding bit in memory. This gives both systems substantial ability to portray graphic images.

Memory Bandwidth

Both Star and Alto have a high memory bandwidth—about 50 MHz, in Star. The entire Star screen is repainted from memory 39 times per second. This 50-MHz video rate would swamp most computer memories, and in fact refreshing the screen takes about 60% of the Alto's memory bandwidth. However, Star's memory is double-ported; therefore, refreshing the display does not appreciably slow down CPU memory access. Star also has separate logic devoted solely to refreshing the display.

Microcoded Personal Computer

Both Star and Alto are personal computers, one user per machine. Therefore the needed memory access and CPU cycles are consistently available. Special microcode has been written to assist in changing the contents of memory quickly, permitting a variety of screen processing that would otherwise not be practical.[6]

Mouse

Both Star and the Alto use a pointing device called the mouse (Figure 2). First developed at SRI,[7] Xerox's version has a ball on the bottom that turns as the mouse slides over a flat surface such as a table. Electronics sense the ball rotation and guide a cursor on the screen in corresponding motions. The mouse is a "Fitts's law" device: that is, after some practice you can point with a mouse as quickly and easily as you can with the tip of your finger. The limitations on pointing speed are those inherent in the human nervous system.[8,9] The mouse has buttons on top that can be sensed under program control. The buttons let you point to and interact with objects on the screen in a variety of ways.

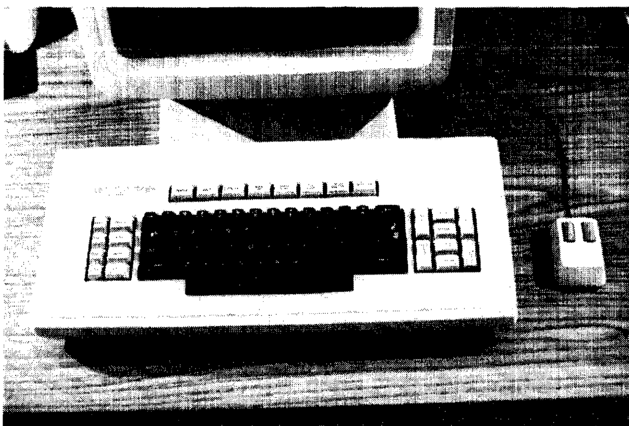


Figure 2—The Star keyboard and mouse

The keyboard has 24 easy-to-understand function keys. The mouse has two buttons on top.

Local Disk

Every Star and Alto has its own rigid disk for local storage of programs and data. Editing does not require using the network. This enhances the personal nature of the machines, resulting in consistent behavior regardless of how many other machines there are on the network or what anyone else is doing. Large programs can be written, using the disk for swapping.

Network

The Ethernet lets both Stars and Altos have a distributed architecture. Each machine is connected to an Ethernet. Other machines on the Ethernet are dedicated as *servers,* machines that are attached to a resource and that provide access to that resource. Typical servers are these:

1. *File server*—Sends and receives files over the network, storing them on its disks. A file server improves on a work station's rigid disk in several ways: (a) Its capacity is greater—up to 1.2 billion bytes. (b) It provides backup facilities. (c) It allows files to be shared among users. Files on a work station's disk are inaccessible to anyone else on the network.
2. *Mail server*—Accepts files over the network and distributes them to other machines on behalf of users, employing the Clearinghouse's database of names and addresses (see below).
3. *Print server*—Accepts print-format files over the network and prints them on the printer connected to it.
4. *Communication server*—Provides several services: The *Clearinghouse service* resolves symbolic names into network addresses.[10] The *Internetwork Routing service* manages the routing of information between networks over phone lines. The *Gateway service* allows word processors and dumb terminals to access network resources.

A network-based server architecture is economical, since many machines can share the resources. And it frees work stations for other tasks, since most server actions happen in the background. For example, while a print server is printing your document, you can edit another document or read your mail.

PHYSICAL OFFICE METAPHOR

We will briefly describe one of the most important principles that influenced the form of the Star user interface. The reader is referred to Smith et al.[1] for a detailed discussion of all the principles behind the Star design. The principle is to apply users' existing knowledge to the new situation of the computer. We decided to create electronic counterparts to the objects in an office: paper, folders, file cabinets, mail boxes, calculators, and so on—an electronic metaphor for the physical office. We hoped that this would make the electronic world seem more familiar and require less training. (Our initial experiences with users have confirmed this.) We further decided to make the electronic analogues be *concrete objects.*

Star documents are represented, not as file names on a disk, but as pictures on the display screen. They may be selected by pointing to them with the mouse and clicking one of the mouse buttons. Once selected, documents may be moved, copied, or deleted by pushing the MOVE, COPY, or DELETE key on the keyboard. Moving a document is the electronic equivalent of picking up a piece of paper and walking somewhere with it. To file a document, you move it to a picture of a file drawer, just as you take a piece of paper to a physical filing cabinet. To print a document, you move it to a picture of a printer, just as you take a piece of paper to a copying machine.

Though we want an analogy with the physical world for familiarity, we don't want to limit ourselves to its capabilities. One of the *raisons d'être* for Star is that physical objects do not provide people with enough power to manage the increasing complexity of their information. For example, we can take advantage of the computer's ability to search rapidly by providing a search function for its electronic file drawers, thus helping to solve the problem of lost files.

THE DESKTOP

Every user's initial view of Star is the Desktop, which resembles the top of an office desk, together with surrounding furniture and equipment. It represents a working environment, where current projects and accessible resources reside. On the screen (Figure 3) are displayed pictures of familiar office objects, such as documents, folders, file drawers, in-baskets, and out-baskets. These objects are displayed as small pictures, or *icons*.

You can "open" an icon by selecting it and pushing the OPEN key on the keyboard. When opened, an icon expands into a larger form called a *window*, which displays the icon's contents. This enables you to read documents, inspect the contents of folders and file drawers, see what mail has arrived, and perform other activities. Windows are the principal mechanism for displaying and manipulating information.

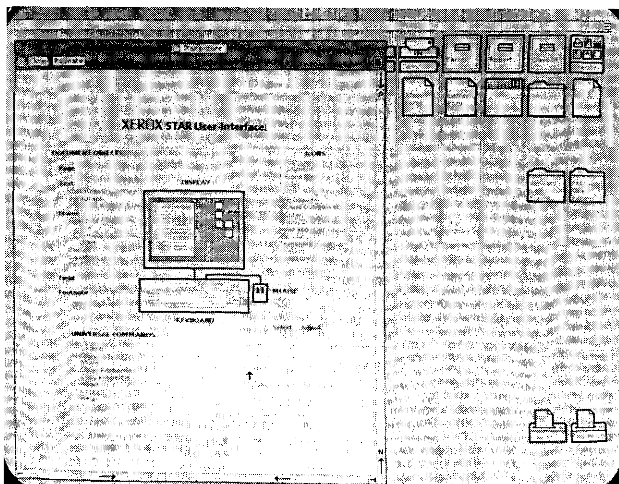


Figure 3—A "Desktop" as it appears on the Star screen

This one has several commonly used icons along the top, including documents to serve as "form pad" sources for letters, memos and blank paper. There is also an open window displaying a document.

The Desktop surface is displayed as a distinctive grey pattern. This is restful and makes the icons and windows on it stand out crisply, minimizing eye strain. The surface is organized as an array of 1-inch squares, 14 wide by 11 high. An icon may be placed in any square, giving a maximum of 154 icons. Star centers an icon in its square, making it easy to line up icons neatly. The Desktop always occupies the entire display screen; even when windows appear on the screen, the Desktop continues to exist "beneath" them.

The Desktop is the principal Star technique for realizing the physical office metaphor. The icons on it are visible, concrete embodiments of the corresponding physical objects. Star users are encouraged to think of the objects on the Desktop in physical terms. You can move the icons around to arrange your Desktop as you wish. (Messy Desktops are certainly possible, just as in real life.) You can leave documents on your Desktop indefinitely, just as on a real desk, or you can file them away.

ICONS

An *icon* is a pictorial representation of a Star object that can exist on the Desktop. On the Desktop, the size of an icon is approximately 1 inch square. Inside a window such as a folder window, the size of an icon is approximately ¼-inch square. Iconic images have played a role in human communication from cave paintings in prehistoric times to Egyptian hieroglyphics to religious symbols to modern corporate logos. Computer science has been slow to exploit the potential of visual imagery for presenting information, particularly abstract information. "Among [the] reasons are the lack of development of appropriate hardware and software for producing visual imagery easily and inexpensively; computer technology has been dominated by persons who seem to be happy with a simple, very limited alphabet of characters used to produce linear strings of symbols."[11] One of the authors has applied icons to an environment for writing programs; he found that they greatly facilitated human-computer communication.[12] Negroponte's Spatial Data Management system has effectively used iconic images in a research setting.[13] And there have been other efforts.[14,15,16] But Star is the first computer system designed for a mass market to employ icons methodically in its user interface. We do not claim that Star exploits visual communication to the ultimate extent; we do claim that Star's use of imagery is a significant improvement over traditional human-machine interfaces.

At the highest level the Star world is divided into two classes of icons, (1) data and (2) function icons:

Data Icons

Data icons (Figure 4) represent objects on which actions are performed. All data icons can be moved, copied, deleted, filed, mailed, printed, opened, closed, and have a variety of other operations performed on them. The three types of data icons are document, folder, and record file.

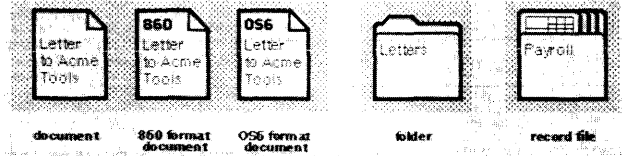


Figure 4—The "data" icons: document, folder and record file

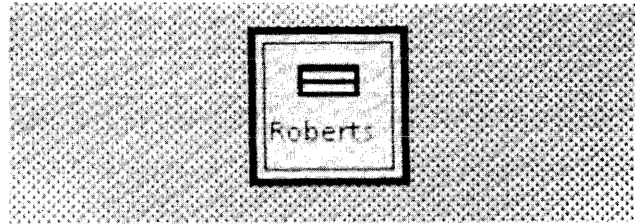


Figure 5—A file drawer icon

Document

A document is the fundamental object in Star. It corresponds to the standard notion of what a document should be. It most often contains text, but it may also include illustrations, mathematical formulas, tables, fields, footnotes, and formatting information. Like all data icons, documents can be shown on the screen, rendered on paper, sent to other people, stored on a file server or floppy disk, etc. When opened, documents are always rendered on the display screen exactly as they print on paper (informally called "what you see is what you get"), including displaying the correct type fonts, multiple columns, headings and footings, illustration placement, etc. Documents can reside in the system in a variety of formats (e.g., Xerox 860, IBM OS6), but they can be edited only in Star format. Conversion operations are provided to translate between the various formats.

Folder

A folder is used to group data icons together. It can contain documents, record files, and other folders. Folders can be nested inside folders to any level. Like file drawers (see below), folders can be sorted and searched.

Record file

A record file is a collection of information organized as a set of records. Frequently this information will be the variable data from forms. These records may be sorted, subset via pattern matching, and formatted into reports. Record files provide a rich set of information storage and retrieval functions.

Function Icons

Function icons represent objects that perform actions. Most function icons will operate on any data icon. There are many kinds of function icons, with more being added as the system evolves:

File drawer

A file drawer (Figure 5) is a place to store data icons. It is modeled after the drawers in office filing cabinets. The organization of a file drawer is up to you; it can vary from a simple list of documents to a multilevel hierarchy of folders containing other folders. File drawers are distinguished from other storage places (folders, floppy disks, and the Desktop) in that (1) icons placed in a file drawer are physically stored on a file server, and (2) the contents of file drawers can be shared by multiple users. File drawers have associated access rights to control the ability of people to look at and modify their contents (Figure 6).

Although the design of file drawers was motivated by their physical counterparts, they are a good example of why it is neither necessary nor desirable to stop with just duplicating real-world behavior. People have a lot of trouble finding things in filing cabinets. Their categorization schemes are frequently ad hoc and idiosyncratic. If the person who did the categorizing leaves the company, information may be permanently lost. Star improves on physical filing cabinets by taking advantage of the computer's ability to *search rapidly*. You can search the contents of a file drawer for an object having a certain name, or author, or creation date, or size, or a variety of other attributes. The search criteria can use fuzzy patterns containing match-anything symbols, ranges, and other predicates. You can also sort the contents on the basis of those criteria. The point is that whatever information retrieval facilities are available in a system should be applied to

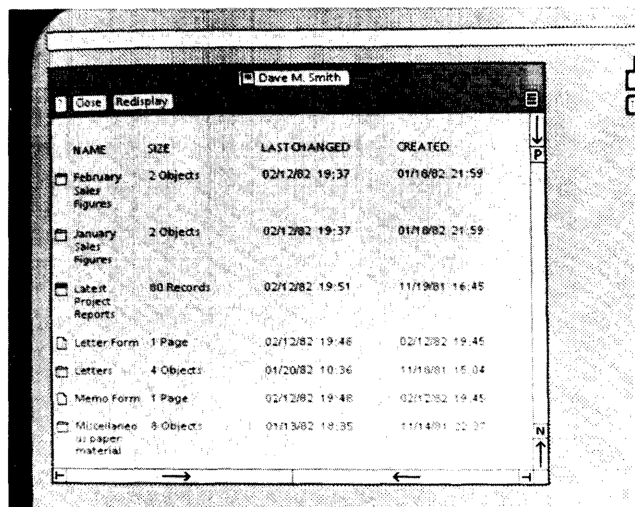


Figure 6—An open file drawer window

Note that there is a miniature icon for each object inside the file drawer.

the information in files. Any system that does not do so is not exploiting the full potential of the computer.

In basket and Out basket

These provide the principal mechanism for sending data icons to other people (Figure 7). A data icon placed in the Out basket will be sent over the Ethernet to a mail server (usually the same machine as a file server), thence to the mail servers of the recipients (which may be the same as the sender's), and thence to the In baskets of the recipients. When you have mail waiting for you, an envelope appears in your In basket icon. When you open your In basket, you can display and read the mail in the window.

Any document, record file, or folder can be mailed. Documents need not be limited to plain text, but can contain illustrations, mathematical formulas, and other nontext material. Folders can contain any number of items. Record files can be arbitrarily large and complex.

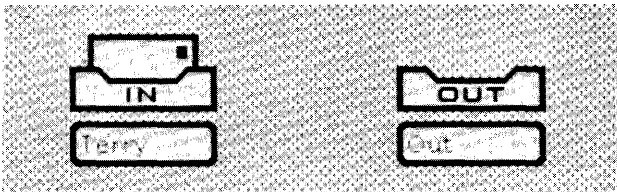


Figure 7—In and Out basket icons

Printer

Printer icons (Figure 8) provide access to printing services. The actual printer may be directly connected to your work station, or it may be attached to a print server connected to an Ethernet. You can have more than one printer icon on your Desktop, providing access to a variety of printing resources. Most printers are expected to be laser-driven raster-scan xerographic machines; these can render on paper anything that can be created on the screen. Low-cost typewriter-based printers are also available; these can render only text.

As with filing and mailing, the existence of the Ethernet greatly enhances the power of printing. The printer represented by an icon on your Desktop can be in the same room as your work station, in a different room, in a different building, in a different city, even in a different country. You perform exactly the same actions to print on any of them: Select a data icon, push the MOVE key, and indicate the printer icon as the destination.

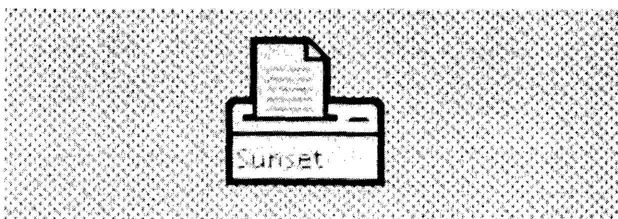


Figure 8—A printer icon

Floppy disk drive

The floppy disk drive icon (Figure 9) allows you to move data icons to and from a floppy disk inserted in the machine. This provides a way to store documents, record files and folders off line. When you open the floppy disk drive icon, Star reads the floppy disk and displays its contents in the window. Its window looks and acts just like a folder window: icons may be moved or copied in or out, or deleted. The only difference is the physical location of the data.

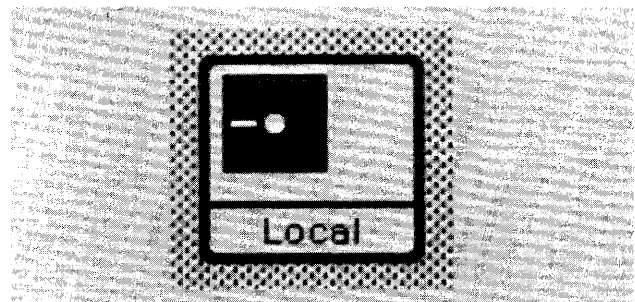


Figure 9—A floppy disk drive icon

User

The user icon (Figure 10) displays the information that the system knows about each user: name, location, password (invisible, of course), aliases if any, home file and mail servers, access level (ordinary user, system administrator, help/training writer), and so on. We expect the information stored for each user to increase as Star adds new functionality. User icons may be placed in address fields for electronic mail.

User icons are Star's solution to the naming problem. There is a crisis in computer naming of people, particularly in electronic mail addressing. The convention in most systems is to

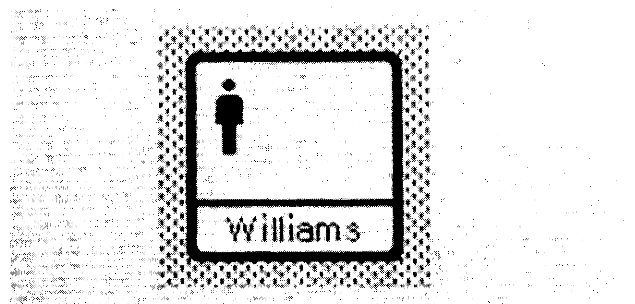


Figure 10—A user icon

use last names for user identification. Anyone named Smith, as is one of the authors, knows that this doesn't work. When he first became a user on such a system, *Smith* had long ago been taken. In fact, "D. Smith" and even "D. C. Smith" had been taken. He finally settled on "DaveSmith", all one word, with which he has been stuck to this day. Needless to say, that is *not* how he identifies himself to people. In the future, people will not tolerate this kind of antihumanism from computers. Star already does better: it follows society's conventions. User icons provide unambiguous unique references to individual people, using their normal names. The information about users, and indeed about all network resources, is physically stored in the Clearinghouse, a distributed database of names. In addition to a person's name in the ordinary sense, this information includes the name of the organization (e.g., Xerox, General Motors) and the name of the user's division within the organization. A person's linear name need be unique only within his division. It can be fully spelled out if necessary, including spaces and punctuation. Aliases can be defined. User icons are *references* to this information. You need not even know, let alone type, the unique linear representation for a user; you need only have the icon.

User group

User group icons (Figure 11) contain individual users and/or other user groups. They allow you to organize people according to various criteria. User groups serve both to control

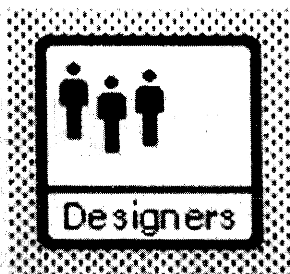


Figure 11—A user group icon

access to information such as file drawers (access control lists) and to make it easy to send mail to a large number of people (distribution lists). The latter is becoming increasingly important as more and more people start to take advantage of computer-assisted communication. At Xerox we have found that as soon as there were more than a thousand Alto users, there were almost always enough people interested in any topic whatsoever to form a distribution list for it. These user groups have broken the bonds of geographical proximity that have historically limited group membership and communication. They have begun to turn Xerox into a nationwide "village," just as the Arpanet has brought computer science researchers around the world closer together. This may be the most profound impact that computers have on society.

Calculator

A variety of styles of calculators (Figure 12) let you perform arithmetic calculations. Numbers can be moved between Star documents and calculators, thereby reducing the amount of typing and the possibility of errors. Rows or columns of tables can be summed. The calculators are user-tailorable and extensible. Most are modeled after pocket calculators—business, scientific, four-function—but one is a tabular calculator similar to the popular Visicalc program.

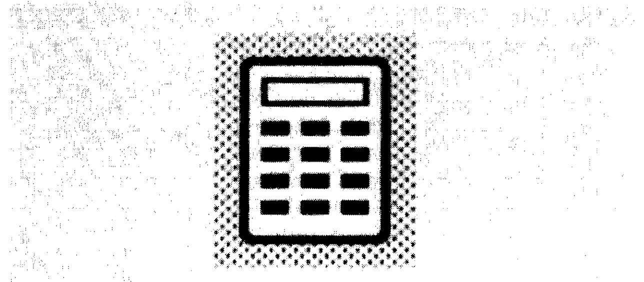


Figure 12—A calculator icon

Terminal emulators

The terminal emulators permit you to communicate with existing mainframe computers using existing protocols. Initially, teletype and 3270 terminals are emulated, with additional ones later (Figure 13). You open one of the terminal icons and type into its window; the contents of the window behave exactly as if you were typing at the corresponding terminal. Text in the window can be copied to and from Star documents, which makes Star's rich environment available to them.

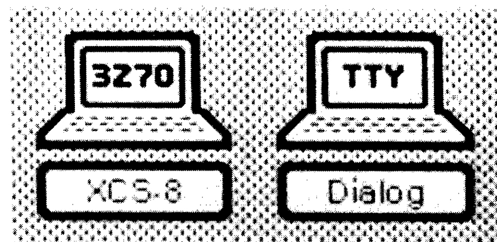


Figure 13—3270 and TTY emulation icons

Directory

The Directory provides access to network resources. It serves as the source for icons representing those resources; the Directory contains one icon for each resource available (Figure 14). When you are first registered in a Star network,

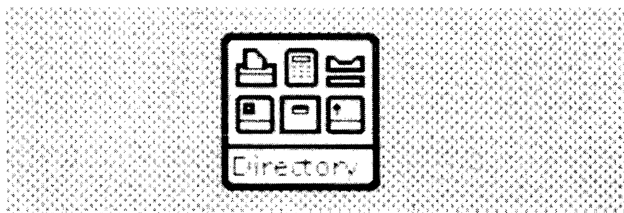


Figure 14—A Directory icon

your Desktop contains nothing but a Directory icon. From this initial state, you access resources such as file drawers, printers, and mail baskets by opening the Directory and copying out their icons. You can also get blank data icons out of the Directory. You can retrieve other data icons from file drawers. Star places no limits on the complexity of your Desktop except the limitation imposed by physical screen area (Figure 15). The Directory also contains Remote Directories representing resources available on other networks. These can be opened, recursively, and their resource icons copied out, just as with the local Directory. You deal with local and remote resources in exactly the same way.

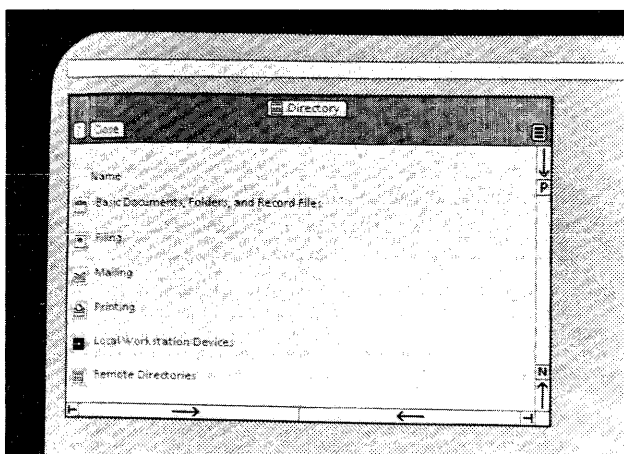


Figure 15—The Directory window, showing the categories of resources available

The important thing to observe is that although the functions performed by the various icons differ, the way you interact with them is the same. You select them with the mouse. You push the MOVE, COPY, or DELETE key. You push the OPEN key to see their contents, the PROPERTIES key to see their properties, and the SAME key to copy their properties. This is the result of rigorously applying the principle of uniformity to the design of icons. We have applied it to other areas of Star as well, as will be seen.

WINDOWS

Windows are rectangular areas that display the contents of icons on the screen. Much of the inspiration for Star's design came from Alan Kay's Flex machine[17] and his later Smalltalk programming environment on the Alto.[18] The Officetalk treatment of windows was also influential; in fact, Officetalk, an experimental office-forms-processing system on the Alto, provided ideas in a variety of areas.[19] Windows greatly increase the amount of information that can be manipulated on a display screen. Up to six windows at a time can be open in Star. Each window has a header containing the name of the icon and a menu of commands. The commands consist of a standard set present in all windows ("?", CLOSE, SET WINDOW) and others that depend on the type of icon. For example, the window for a record file contains commands tailored to information retrieval. CLOSE removes the window from the display screen, returning the icon to its tiny size. The "?" command displays the online documentation describing the type of window and its applications.

Each window has two scroll bars for scrolling the contents vertically and horizontally. The scroll bars have jump-to-end areas for quickly going to the top, bottom, left, or right end of the contents. The vertical scroll bar also has areas labeled N and P for quickly getting the next or previous screenful of the contents; in the case of a document window, they go to the next or previous page. Finally, the vertical scroll bar has a jumping area for going to a particular part of the contents, such as to a particular page in a document.

Unlike the windows in some Alto programs, Star windows do not overlap. This is a deliberate decision, based on our observation that many Alto users were spending an inordinate amount of time manipulating windows themselves rather than their contents. This manipulation of the medium is overhead, and we want to reduce it. Star automatically partitions the display space among the currently open windows. You can control on which side of the screen a window appears and its height.

PROPERTY SHEETS

At a finer grain, the Star world is organized in terms of *objects* that have *properties* and upon which *actions* are performed. A few examples of objects in Star are text characters, text paragraphs, graphic lines, graphic illustrations, mathematical summation signs, mathematical formulas, and icons. Every object has properties. Properties of text characters include type style, size, face, and posture (e.g., bold, italic). Properties of paragraphs include indentation, leading, and alignment. Properties of graphic lines include thickness and structure (e.g., solid, dashed, dotted). Properties of document icons include name, size, creator, and creation date. So the properties of an object depend on the type of the object. These ideas are similar to the notions of classes, objects, and messages in Simula[20] and Smalltalk. Among the editors that use these ideas are the experimental text editor Bravo[21] and the experimental graphics editor Draw,[22] both developed at the Xerox Palo Alto Research Center. These all supplied valuable knowledge and insight to Star. In fact, the text editor aspects of Star were derived from Bravo.

In order to make properties visible, we invented the notion of a property sheet (Figure 16). A property sheet is a two-dimensional formlike environment which shows the proper-

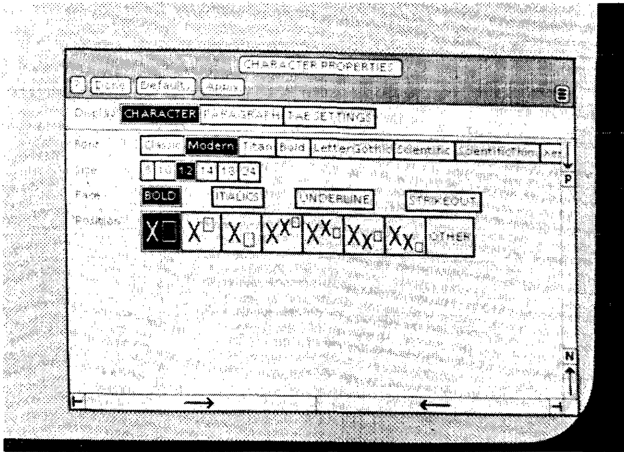


Figure 16—The property sheet for text characters

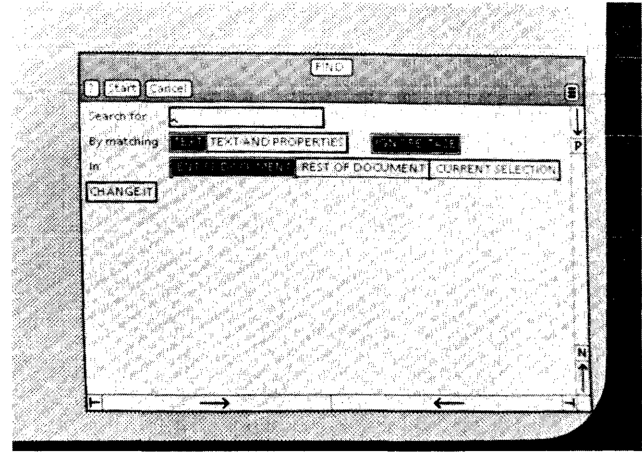


Figure 17—The option sheet for the Find command

ties of an object. To display one, you select the object of interest using the mouse and push the PROPERTIES key on the keyboard. Property sheets may contain three types of parameters:

1. *State*—State parameters display an independent property, which may be either on or off. You turn it on or off by pointing to it with the mouse and clicking a mouse button. When on, the parameter is shown video reversed. In general, any combination of state parameters in a property sheet can be on. If several state parameters are logically related, they are shown on the same line with space between them. (See "Face" in Figure 16.)
2. *Choice*—Choice parameters display a set of mutually exclusive values for a property. Exactly one value must be on at all times. As with state parameters, you turn on a choice by pointing to it with the mouse and clicking a mouse button. If you turn on a different value, the system turns off the previous one. Again the one that is on is shown video reversed. (See "Font" in Figure 16.) The motivation for state and choice parameters is the observation that it is generally easier to take a multiple-choice test than a fill-in-the-blanks one. When options are made visible, they become easier to understand, remember, and use.
3. *Text*—Text parameters display a box into which you can type a value. This provides a (largely) unconstrained choice space; you may type any value you please, within the limits of the system. The disadvantage of this is that the set of possible values is not visible; therefore Star uses text parameters only when that set is large. (See "Search for" in Figure 17.)

Property sheets have several important attributes:

1. A small number of parameters gives you a large number of combinations of properties. They permit a rich choice space without a lot of complexity. For example, the character property sheet alone provides for 8 fonts, from 1 to 6 sizes for each (an average of about 2), 4 faces (any combination of which can be on), and 8 positions relative to the baseline (including OTHER, which lets you type in a value). So in just four parameters, there are over $8 \times 2 \times 2^4 \times 8 = 2048$ combinations of character properties.
2. They show all of the properties of an object. None is hidden. You are constantly reminded what is available every time you display a property sheet.
3. They provide progressive disclosure. There are a large number of properties in the system as a whole, but you want to deal with only a small subset at any one time. Only the properties of the selected object are shown.
4. They provide a "bullet-proof" environment for altering the characteristics of an object. Since only the properties of the selected object are shown, you can't accidentally alter other objects. Since only valid choices are displayed, you can't specify illegal properties. This reduces errors.

Property sheets are an example of the Star design principle that *seeing and pointing* is preferred over *remembering and typing*. You don't have to remember what properties are available for an object; the property sheet will show them to you. This reduces the burden on your memory, which is particularly important in a functionally rich system. And most properties can be changed by a simple pointing action with the mouse.

The three types of parameters are also used in *option sheets*. (Figure 18). Option sheets are just like property sheets, except that they provide a visual interface for *arguments to commands* instead of *properties of objects*. For example, in the Find option sheet there is a text parameter for the string to search for, a choice parameter for the range over which to search, and a state parameter (CHANGE IT) controlling whether to replace that string with another one. When CHANGE IT is turned on, an additional set of parameters appears to contain the replacement text. This technique of having some parameters appear depending on the settings of others is another part of our strategy of progressive disclosure: hiding information (and therefore complexity) until it is

needed, but making it visible when it is needed. The various sheets appear simpler than if all the options were always shown.

COMMANDS

Commands in Star take the form of noun-verb pairs. You specify the object of interest (the noun) and then invoke a command to manipulate it (the verb). Specifying an object is called *making a selection.* Star provides powerful selection mechanisms, which reduce the number and complexity of commands in the system. Typically, you exercise more dexterity and judgment in making a selection than in invoking a command. The ways to make a selection are as follows:

1. With the mouse—Place the cursor over the object on the screen you want to select and click the first (SELECT) mouse button. Additional objects can be selected by using the second (ADJUST) mouse button; it adjusts the selection to include more or fewer objects. Most selections are made in this way.
2. With the NEXT key on the keyboard—Push the NEXT key, and the system will select the contents of the next field in a document. Fields are one of the types of special higher-level objects that can be placed in documents. If the selection is currently in a table, NEXT will step through the rows and columns of the table, making it easy to fill in and modify them. If the selection is currently in a mathematical formula, NEXT will step through the various elements in the formula, making it easy to edit them. NEXT is like an intelligent step key; it moves the selection between semantically meaningful locations in a document.
3. With a command—Invoke the FIND command, and the system will select the next occurrence of the specified text, if there is one. Other commands that make a selection include OPEN (the first object in the opened window is selected) and CLOSE (the icon that was closed becomes selected). These optimize the use of the system.

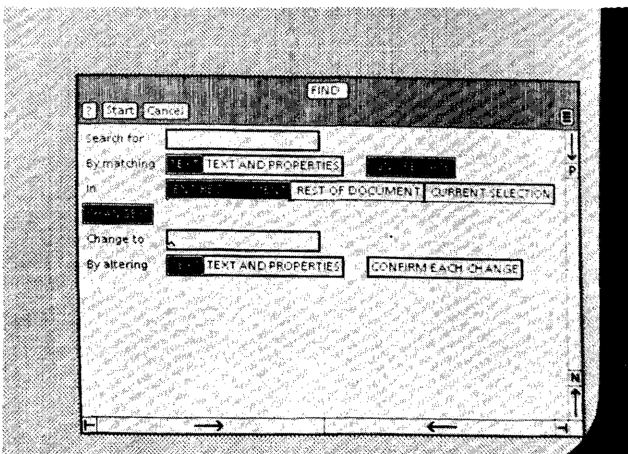


Figure 18—The Find option sheet showing Substitute options *(The extra options appear only when CHANGE IT is turned on)*

The object (noun) is almost always specified before the action (verb) to be performed. This makes the command interface *modeless;* you can change your mind as to which object to affect simply by changing the selection before invoking the command.[23] No "accept" function is needed to terminate or confirm commands, since invoking the command is the last step. Inserting text does not require a command; you simply make a selection and begin typing. The text is placed after the end of the selection. A few commands require more than one operand and hence are modal. For example, the MOVE and COPY commands require a destination as well as a source.

GENERIC COMMANDS

Star has a few commands that can be used throughout the system: MOVE, COPY, DELETE, SHOW PROPERTIES, COPY PROPERTIES, AGAIN, UNDO, and HELP. Each performs the same way regardless of the type of object selected. Thus we call them generic commands. For example, you follow the same set of actions to move text in a document as to move a document in a folder or a line in an illustration: select the object, move the MOVE key, and indicate the destination. Each generic command has a key devoted to it on the keyboard. (HELP and UNDO don't use a selection.)

These commands are more basic than the ones in other computer systems. They strip away extraneous application-specific semantics to get at the underlying principles. Star's generic commands are derived from *fundamental computer science concepts* because they also underlie operations in programming languages. For example, program manipulation of data structures involves moving or copying values from one data structure to another. Since Star's generic commands embody fundamental underlying concepts, they are widely applicable. Each command fills a host of needs. Few commands are required. This simplicity is desirable in itself, but it has another subtle advantage: it makes it easy for users to form a model of the system. What people can understand, they can use. Just as progress in science derives from simple, clear theories, so progress in the usability of computers depends on simple, clear user interfaces.

Move

MOVE is the most powerful command in the system. It is used during text editing to rearrange letters in a word, words in a sentence, sentences in a paragraph, and paragraphs in a document. It is used during graphics editing to move picture elements such as lines and rectangles around in an illustration. It is used during formula editing to move mathematical structures such as summations and integrals around in an equation. It replaces the conventional "store file" and "retrieve file" commands; you simply move an icon into or out of a file drawer or folder. It eliminates the "send mail" and "receive mail" commands; you move an icon to an Out basket or from an In basket. It replaces the "print" command; you move an icon to a printer. And so on. MOVE strips away much of the historical clutter of computer commands. It is more fundamental than the myriad of commands it replaces. It is simultaneously more powerful and simpler.

MOVE also reinforces Star's physical metaphor: a moved object can be in only one place at one time. Most computer file transfer programs only make copies; they leave the originals behind. Although this is an admirable attempt to keep information from accidentally getting lost, an unfortunate side effect is that sometimes you lose track of where the most recent information is, since there are multiple copies floating around. MOVE lets you model the way you manipulate information in the real world, should you wish to. We expect that during the *creation* of information, people will primarily use MOVE; during the *dissemination* of information, people will make extensive use of COPY.

Copy

COPY is just like MOVE, except that it leaves the original object behind untouched. Star elevates the concept of copying to the level of *a paradigm for creating.* In all the various domains of Star, you *create by copying.* Creating something out of nothing is a difficult task. Everyone has observed that it is easier to modify an existing document or program than to write it originally. Picasso once said, "The most awful thing for a painter is the white canvas. . . . To copy others is necessary."[24] Star makes a serious attempt to alleviate the problem of the "white canvas," to make copying a practical aid to creation. Consider:

- You create new documents by copying existing ones. Typically you set up blank documents with appropriate formatting properties (e.g., fonts, margins) and then use those documents as *form pad* sources for new documents. You select one, push COPY, and presto, you have a new document. The form pad documents need not be blank; they can contain text and graphics, along with fields for variable text such as for business forms.
- You place new network resource icons (e.g., printers, file drawers) on your Desktop by copying them out of the Directory. The icons are registered in the Directory by a system administrator working at a server. You simply copy them out; no other initialization is required.
- You create graphics by copying existing graphic images and modifying them. Star supplies an initial set of such images, called *transfer symbols.* Transfer symbols are based on the idea of dry-transfer rub-off symbols used by many secretaries and graphic artists. Unlike the physical transfer symbols, however, the computer versions can be modified: they can be moved, their sizes and proportions can be changed, and their appearance properties can be altered. Thus a single Star transfer symbol can produce a wide range of images. We will eventually supply a set of documents (transfer sheets) containing nothing but special images tailored to one application or another: people, buildings, vehicles, machinery. Having these as sources for graphics copying helps to alleviate the "white canvas" feeling.
- In a sense, you can even type characters by copying them from keyboard windows. Since there are many more characters (up to 2^{16}) in the Star character set than there are keys on the keyboard, Star provides a series of keyboard interpretation windows (Figure 19), which allow you to see and change the meanings of the keyboard keys. You are presented with the options; you look them over and choose the ones you want.

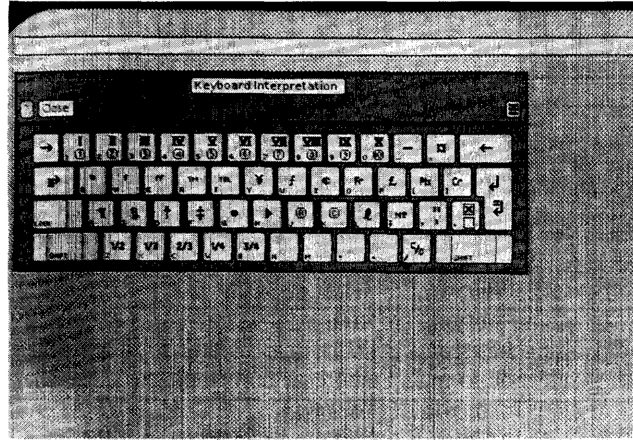


Figure 19—The Keyboard Interpretation window

This displays other characters that may be entered from the keyboard. The character set shown here contains a variety of common office symbols.

Delete

This deletes the selected object. If you delete something by mistake, UNDO will restore it.

Show Properties

SHOW PROPERTIES displays the properties of the selected object in a property sheet. You select the object(s) of interest, push the PROPERTIES (PROP'S) key, and the appropriate property sheet appears on the screen in such a position as to not overlie the selection, if possible. You may change as many properties as you wish, including none. When finished, you invoke the Done command in the property sheet menu. The property changes are applied to the selected objects, and the property sheet disappears. Notice that SHOW PROPERTIES is therefore used both to examine the current properties of an object and to change those properties.

Copy Properties

You need not use property sheets to alter properties if there is another object on the screen that already has the desired properties. You can select the object(s) to be changed, push the SAME key, then designate the object to use as the source. COPY PROPERTIES makes the selection look the "same" as the source. This is particularly useful in graphics editing. Frequently you will have a collection of lines and symbols whose appearance you want to be coordinated (all the same line width, shade of grey, etc.). You can select all the objects to be changed, push SAME, and select a line or symbol having

the desired appearance. In fact, we find it helpful to set up a document with a variety of graphic objects in a variety of appearances to be used as sources for copying properties.

Again

AGAIN repeats the last command(s) on a new selection. All the commands done since the last time a selection was made are repeated. This is useful when a short sequence of commands needs to be done on several different selections; for example, make several scattered words bold and italic and in a larger font.

Undo

UNDO reverses the effects of the last command. It provides protection against mistakes, making the system more forgiving and user-friendly. Only a few commands cannot be repeated or undone.

Help

Our effort to make Star a personal, self-contained system goes beyond the hardware and software to the tools that Star provides to teach people how to use the system. Nearly all of its teaching and reference material is on line, stored on a file server. The Help facilities automatically retrieve the relevant material as you request it.

The HELP key on the keyboard is the primary entrance into this online information. You can push it at any time, and a window will appear on the screen displaying the Help table of contents (Figure 20). Three mechanisms make finding information easier: *context-dependent invocation, help references,* and a *keyword search command.* Together they make the online documentation more powerful and useful than printed documentation.

- *Context-dependent invocation*—The command menu in every window and property/option sheet contains a "?" command. Invoking it takes you to a part of the Help documentation describing the window, its commands, and its functions. The "?" command also appears in the message area at the top of the screen; invoking that one takes you to a description of the message (if any) currently in the message area. That provides more detailed explanations of system messages.
- *Help references*—These are like menu commands whose effect is to take you to a different part of the Help material. You invoke one by pointing to it with the mouse, just as you invoke a menu command. The writers of the material use the references to organize it into a network of interconnections, in a way similar to that suggested by Vannevar Bush[25] and pioneered by Doug Engelbart in his NLS system.[26,27] The interconnections permit cross-referencing without duplication.
- The *SEARCH FOR KEYWORD command*—This command in the Help window menu lets you search the available documentation for information on a specific topic. The keywords are predefined by the writers of the Help material.

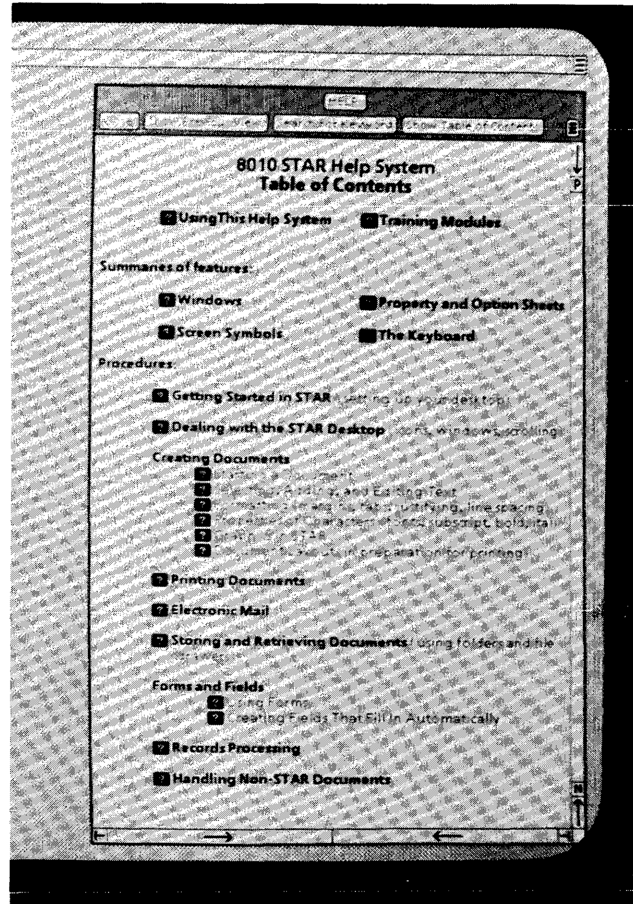


Figure 20—The Help window, showing the table of contents

Selecting a square with a question mark in it takes you to the associated part of the Help documentation.

SUMMARY

We have learned from Star the importance of formulating the user's conceptual model *first, before* software is written, rather than tacking on a user interface *afterward.* Doing good user interface design is not easy. Xerox devoted about thirty work-years to the design of the Star user interface. It was designed *before* the functionality of the system was fully decided. It was designed *before* the computer hardware was even built. We worked for two years *before* we wrote a single line of actual product software. Jonathan Seybold put it this way: "Most system design efforts start with hardware specifications, follow this with a set of functional specifications for the software, then try to figure out a logical user interface and command structure. The Star project started the other way around: the paramount concern was to define a conceptual model of how the user would relate to the system. Hardware and software followed from this."[4]

Alto served as a valuable prototype for Star. Over a thousand Altos were eventually built, and Alto users have had several thousand work-years of experience with them over a period of eight years, making Alto perhaps the largest proto-

typing effort in history. There were dozens of experimental programs written for the Alto by members of the Xerox Palo Alto Research Center. Without the creative ideas of the authors of those systems, Star in its present form would have been impossible. On the other hand, it was a real challenge to bring some order to the different user interfaces on the Alto. In addition, we ourselves programmed various aspects of the Star design on Alto, but every bit (sic) of it was throwaway code. Alto, with its bit-mapped display screen, was powerful enough to implement and test our ideas on visual interaction.

REFERENCES

1. Smith, D. C., E. F. Harslem, C. H. Irby, R. B. Kimball, and W. L. Verplank. "Designing the Star User Interface." *Byte,* April 1982.
2. Metcalfe, R. M., and D. R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM,* 19 (1976), pp. 395–404.
3. Intel, Digital Equipment, and Xerox Corporations. "The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (version 1.0)." Palo Alto: Xerox Office Products Division, 1980.
4. Seybold, J. W. "Xerox's 'Star.'" *The Seybold Report.* Media, Pennsylvania: Seybold Publications, 10 (1981), 16.
5. Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. "Alto: A Personal Computer." In D. Siewiorek, C. G. Bell, and A. Newell (eds.), *Computer Structures: Principles and Examples.* New York: McGraw-Hill, 1982.
6. Ingalls, D. H. "The Smalltalk Graphics Kernel." *Byte,* 6 (1981), pp. 168–194.
7. English, W. K., D. C. Engelbart, and M. L. Berman. "Display-Selection Techniques for Text Manipulation." *IEEE Transactions on Human Factors in Electronics,* HFE-8 (1967), pp. 21–31.
8. Fitts, P. M. "The Information Capacity of the Human Motor System in Controlling Amplitude of Movement." *Journal of Experimental Psychology,* 47 (1954), pp. 381–391.
9. Card, S., W. K. English, and B. Burr. "Evaluation of Mouse, Rate-Controlled Isometric Joystick, Step Keys, and Text Keys for Text Selection on a CRT." *Ergonomics,* 21 (1978), pp. 601–613.
10. Oppen, D. C., and Y. K. Dalal. "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment." Palo Alto: Xerox Office Products Division, OPD-T8103, 1981.
11. Huggins, W. H., and D. Entwisle. *Iconic Communication.* Baltimore and London: The Johns Hopkins University Press, 1974.
12. Smith, D. C. *Pygmalion, A Computer Program to Model and Stimulate Creative Thought.* Basel and Stuttgart: Birkhäuser Verlag, 1977.
13. Bolt, R. *Spatial Data-Management.* Cambridge, Massachusetts: Massachusetts Institute of Technology Architecture Machine Group, 1979.
14. Sutherland, I. "Sketchpad, A Man-Machine Graphical Communication System." *AFIPS, Proceedings of the Fall Joint Computer Conference* (Vol. 23), 1963, pp. 329–346.
15. Sutherland, W. "On-Line Graphical Specifications of Computer Procedures." Cambridge, Massachusetts: Massachusetts Institute of Technology, 1966.
16. Christensen, C. "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language." In M. Klerer and J. Reinfelds (eds.), *Interactive Systems for Experimental and Applied Mathematics.* New York: Academic Press, 1968.
17. Kay, A. C. *The Reactive Engine.* Salt Lake City: University of Utah, 1969.
18. Kay, A. C., and the Learning Research Group. "Personal Dynamic Media." Xerox Palo Alto Research Center Technical Report SSL-76-1, 1976. (A condensed version is in *IEEE Computer,* March 1977, pp. 31–41.)
19. Newman, W. M. "Officetalk-Zero: A User's Manual." Xerox Palo Alto Research Center Internal Report, 1977.
20. Dahl, O. J., and K. Nygaard. "SIMULA–An Algol-Based Simulation Language." *Communications of the ACM,* 9 (1966), pp. 671–678.
21. Lampson, B. "Bravo Manual." In *Alto User's Handbook,* Xerox Palo Alto Research Center, 1976 and 1978. (Much of the design and all of the implementation of Bravo was done by Charles Simonyi and the skilled programmers in his "software factory.")
22. Baudelaire, P., and M. Stone. "Techniques for Interactive Raster Graphics." *Proceedings of the 1980 Siggraph Conference,* 14 (1980), 3.
23. Tesler, L. "The Smalltalk Environment." *Byte,* 6 (1981), pp. 90–147.
24. Wertenbaker, L. *The World of Picasso.* New York: Time-Life Books, 1967.
25. Bush, V. "As We May Think." *Atlantic Monthly,* July 1945.
26. Engelbart, D. C. "Augmenting Human Intellect: A Conceptual Framework." Technical Report AFOSR-3223, SRI International, Menlo Park, Calif., 1962.
27. Engelbart, D. C., and W. K. English. "A Research Center for Augmenting Human Intellect." *AFIPS Proceedings of the Fall Joint Computer Conference* (Vol. 33), 1968, pp. 395–410.

MFS: a modular text formatting system

by JAMES D. MOONEY
West Virginia University
Morgantown, WV

ABSTRACT

This paper presents the design goals and architecture of the Modular Formatting System, currently being developed at West Virginia University. MFS applies to formatters the principles of separation of function used in many successful program systems. A small central kernel forms the basis for a family of formatting systems, tailored to specific applications and environments.

MFS is intended to support a wide spectrum of applications drawn from the experience of commercial composition, word processing, and research systems. It does not rely on any specialized terminal or input source characteristics. Output devices from line printers through high resolution typesetters are handled in a uniform manner. Users can exercise detailed control over document appearance, or work exclusively with styles predefined through macros.

INTRODUCTION

Computer-aided composition of documents is an application which has developed rapidly since the early report preparation systems such as RUNOFF[1]; programmed photocomposers from such companies as Star Parts, Photon, Compugraphic, Harris, and Mergenthaler; and mainframe formatting systems such as PAGE-1.[2] Each of these lines has produced many more advanced descendants. TROFF[3] is a powerful formatter in the RUNOFF family. Commercial systems now automate almost all the composition tasks of complete books or newspapers. Research systems such as TEX[4] and SCRIBE[5] have arisen which provide ambitious solutions to complex composition problems. The development of "word-processing" systems has made a new mode of interactive document preparation available in many environments, such as business offices, where it would not have been considered before.

As the notion of using a computer to prepare documents becomes more accepted and ubiquitous, many such systems are entering new environments where their full potential is only gradually discovered. It is characteristic that users will, if possible, apply this potential to tasks not thought of when the system was first acquired. New categories of documents may be processed; new, more sophisticated output devices may be obtained, and a variety of more (or less) powerful terminals may be added as text entry and editing becomes a more democratic process.

Expansion of the use of document preparation systems brings many persons into contact with the system who will use it only if use is extremely easy and natural, and few special codes or procedures need to be learned. At the same time, familiarity brings increased demands; and many users, once content with a typewritten product, develop the discernment of professional typographers. While taking full advantage of the automatic processing available from a powerful formatter, they may also want hairline control of the document's final appearance, and of the formatter's behavior in various specific situations.

These considerations demonstrate the need for a document processing system which, over its lifetime, can deal with a considerable diversity of input and editing mechanisms, target output devices, applications, and user demands for flexibility or convenience. Some existing systems are quite flexible along some of these dimensions, but no system known to the author appears to be sufficiently general in all ways. The Modular Formatting System (MFS) is an attempt to address this need.

DESIGN ISSUES AND GOALS

The overall objective of MFS is to synthesize the essential capabilities of previous formatting systems in a concise and modular system which can be adapted to a great variety of applications and environments. MFS establishes four specific goals of generality: input source independence, target device independence, support for a wide spectrum of applications, and user-selected levels of flexibility or convenience. Each of these goals will be discussed below.

Input source independence

Many formatting systems are limited to use with specific terminal types. This is true especially of word processors, which are most often supplied by the terminal vendor. In contrast, many user environments will progress from a few simple terminals to many diverse input sources as their system usage evolves.

The likely variations in input source characteristics include coding format and degree of interactivity. The coding conventions of any particular terminal can be mapped into a standard form such as ASCII by an input module matched to the terminal. If the original input includes graphic information such as specific fonts, special characters, or spacing, this information also must be converted to a suitable command stream by the input module.

Like other software systems, the early document processing programs were all designed to process complete marked-up documents in batch mode. The rise of screen-oriented editors with some formatting capabilities (word processors) parallels the rise of the interactive user interface. In this environment, users can modify their documents incrementally, and see immediate results from their formatting instructions.

On the other hand, interactive computing has not made popular such tools as incremental (interpretive) compilers, except for very simple languages and programs. Most compiling is still performed on complete programs without interaction. A similar pattern exists for document formatting. Editing programs insert and delete text, and perhaps carry out simple line filling incrementally. More elaborate formatting, often requiring knowledge of a wide context, is invoked by distinct commands on well-defined units of text.

A further impediment to interactive document creation occurs since there is most often a mismatch between the abilities of the terminal and the target output device to represent text. Some applications target very simple printers, and some terminals have high-resolution, bit-mapped displays, but in most cases the terminal can only approximate the final output appearance. This approximately processed text, while useful as a proof reference, may be more difficult to work with than the original, marked-up document.

For these reasons, MFS can be viewed essentially as operating on a batch of input, which may be digested in its entirety before output is produced. It should not, however, ignore the

possibility that it was invoked interactively. Decisions may arise in processing, such as questionable line breaks or hyphenation, meaningless commands, or impossible typographic situations. Where appropriate, the relevant modules of MFS should be able to query the user for a solution to these problems.

Target device independence

Only a few existing formatters (e.g. SCRIBE) are comfortable with a full spectrum of output devices. TEX relies on a variety of special characters and use of its own fonts, and is not easily mapped down to simpler devices. TROFF drives a single sort of typesetter, and has a separate version, NROFF, for simple printers. Word processors know little of multiple fonts and precise spacing control. The limitations of vendor-supplied systems are evident.

A major goal of MFS is to be able to drive a wide class of output devices, making full use of their typographic capabilities, and to defer the choice of device as much as possible while processing.

The simplest devices are those of the "line-printer" class with a single character font and fixed character and line spacing. Advanced printers, phototypesetters, etc. may provide many fonts, precision spacing, and other variables. For much routine text processing the simplest printers are adequate, and they are heavily used. But advanced devices are appearing side by side with line-printers in a growing number of environments. The choice of device for each job is then a matter of taste and economics.

A user who works with a variety of target devices will prefer to view them through a common formatter, controlled by a uniform coding mechanism.

In most processing, the user will have in mind a specific target device and configuration when the document is prepared. Indeed, character selection and many creative format decisions must usually be made in view of the capabilities of the output device. However, in some applications a document's life cycle will go beyond immediate printing on a single device. It may be printed from time to time on several devices; it may be distributed to various sites to be printed in a variety of environments; it may be printed on an unexpected device if the intended target is unavailable. In these situations it is generally acceptable to reprocess the source document through the formatter with the actual target specified; but it may not be acceptable to require manual re-editing of the document.

These considerations impose the following requirements on MFS:

1. It should be able to access all foreseeable capabilities of the various devices for character selection, transformation, positioning, etc.
2. It should allow the user to prepare documents for a *virtual target* not necessarily bound to any single real device.
3. It should, like SCRIBE, try to provide a reasonable interpretation if the specified characters or processing functions are not available on the specified device.

Further, it should not be necessary (as in TEX) to specify characters in a way which depends on their grouping within the output device. Users should be free to work with *virtual fonts,* character groupings formed for the convenience of the application.

Wide spectrum of applications

The primitive functions available through various formatters vary widely, directly affecting the class of application problems which can be addressed. Many recent formatters offer valuable solutions to complex problems, e.g. math formula layout, compiling indexes and bibliographies, etc. At the same time, capabilities which have proven invaluable in commercial composition have not been carried through to newer systems. Word processors may offer interesting interfaces to data processing functions, yet never address many typographic needs.

All of the possible capabilities serve ultimately to define mappings from input strings (text and command information) to output strings (character specifications and positions). However, many of these mappings are highly context dependent. A principal goal of MFS is to identify a set of primitive functions which support as large a consistent set of capabilities as possible, and to provide these functions in the MFS kernel.

The following is a partial list of capabilities which have been proven useful by various formatting systems. Preserving these capabilities has significant implications for a formatter architecture. Syntactic issues, which have less impact, are considered in the next subsection.

Line breaking and justification

For some typographers, the best justification algorithm is a very personal matter. We may consider a single line or entire paragraphs in making decisions. Excess space may be distributed among word breaks and between letters using various strategies. There should be provision for overriding "quad spaces" before, after, or within any text, and for filling these spaces with characters which may need to meet some criteria for alignment.

Hyphenation

Good hyphenation is highly subjective, and the algorithm must be variable. Technical and geographic words may have to be added to the dictionary for specific jobs. Different languages require entirely different approaches, occasionally within the same job. This need is recognized in most commercial systems, while research systems have treated even simple hyphenation as an afterthought.

Multiple environments

Ability to save and switch the prevailing collection of formatting parameters is necessary for clean merging of intermixed text such as footnotes. This facility exists in most systems only in a limited or specialized form.

Deferred input sequences

It is often necessary to specify a sequence of commands or text to be deferred until some typographic condition occurs, e.g., after 6 inches of text. This ability, found to some degree in most commercial systems, supports picture cuts, running heads, floating tables, and other important structures. The deferred input mechanism is found in limited form in TROFF, and missing from TEX and SCRIBE.

Deferred output

The ability to save text *after* processing, for insertion into the output at a later point, is the software analog of "cut and paste." This function, important for footnotes, floating tables, etc., is well developed in the "diversions" of TROFF and the "boxes" of TEX.

Text measurement

The ability to preprocess text to determine its dimensions or other characteristics before output is a special case of Deferred Output. These measurements may be needed for problems such as footnote fitting, large initial capitals, vertical justification, etc. Many systems provide only the ability to determine the width of a short string.

Numeric variables

Various capabilities are made possible by use of numeric variables (normally integer) which can be set or tested and manipulated with simple arithmetic. Examples are numbering of pages and other text items, or setting conditions which will be tested.

Readable state variables

As a special case of numeric variables, it is valuable to be able to determine and test parameters of the current typographic context, such as current coordinates on the output page.

Special displays

Various applications have requirements to allow manipulation of specialized display material in a convenient and appropriate manner. Examples are mathematical formulas as addressed by TEX, and layout of advertising copy which is a major concern of commercial systems.

Multilingual support

A comprehensive formatter must address the specialized needs of specific languages. Problems not arising in English text may include setting right to left (Arabic) or top to bottom (Chinese); very large alphabets; positioning accents and building up characters from parts; or hyphenation which modifies the resulting partial words.

Transformation systems

Many formatters include specialized subsystems which (eventually) generate output text after digesting various kinds of input information. Examples include the bibliographies and indexes of SCRIBE and the numerical calculations of some word processors, as well as number-to-string conversions and assorted manipulations of the date and time. Some of these functions require knowledge of the typographic context, and cannot be viewed as preprocessors.

Conditionals

In general a conditional is a transformation system yielding a logical value instead of an output string. This value may be used to select between two different well-defined input sequences.

Graphic material

Some output devices are able to generate graphic images as well as text characters. A formatter is not likely to contain a picture definition facility; but it should be able to accept picture descriptions in which the primitives known to the output system, whether dots, vectors, or higher subpictures, are presented as special characters. The resulting graphics may be mixed with text; and interaction with systems such as PIC[6], as preprocessors or transformation systems, should be a possibility.

User-selected flexibility and convenience

The user's wishes for an ideal view of the system include various tradeoffs. Flexibility, convenience, error tolerance, economy of input, and readability of marked-up text are all desirable characteristics which cannot be maximized at the same time. Markup may be desired to be independent of the output device, or to take advantage of features of a particular output device. To meet these needs the user interface should be easily customized.

A part of this interface is the domain of the input or editing system rather than the formatter; and advanced input systems may hide much markup and generate it automatically in any required format. However, the formatter needs to be concerned with the flexibility of the interface presented to very simple input systems.

The basic approach to meeting these needs is to provide a flexible, low-level command language and a powerful macro facility. The command language alone provides all possible direct control of formatting details, while the macro facility makes possible selected levels of abstraction culminating in a minimum-markup, non-procedural language such as GML.[7] Both TROFF and TEX provide good examples of powerful

command and macro systems, but neither meets all of the following desirable requirements:

1. The complete set of input characters which will have special meaning, such as command prefixes, should be user-selectable. It should be possible to keep the number of such characters extremely small.
2. All names for commands, macros, variables, etc. should be user-selectable. The range of possible names should run from single characters to long, descriptive names.
3. Command and macro arguments, or the range of text affected, may consist of a single character or many lines of input. Any range should be selectable by suitable bracketing. Commands should be acceptable anywhere regardless of input line endings.
4. The user or application should be able to dynamically define virtual fonts, complete mappings from input text characters into the set of presumably available output characters. This mapping should include both single input characters and strings (named characters, ligatures, etc.). It should hide any differences in the actual access mechanism for the selected character set.
5. The significance of spaces and line endings in the input text should be consistent and intuitive in any context.

In addition, to support a modular organization of the documents themselves there should be a nestable inclusion mechanism for the component files of the input text. This mechanism should make use of string variables to avoid embedding system-dependent file names in the document itself.

SYSTEM ARCHITECTURE

The MFS architecture is designed as a small, central kernel interacting with a collection of processing modules. Like the kernel of an operating system, the MFS kernel is intended to provide only essential, primitive support functions while making few restrictions on possible processing mechanisms. This organization is shown in Figure 1.

The division of the system into modules provides a clear separation of functions, isolating capabilities that may be independently varied. The hyphenation system may be replaced without impact on the rest of the system. Similarly, MFS may be transported to a different computing system with changes only in isolated modules.

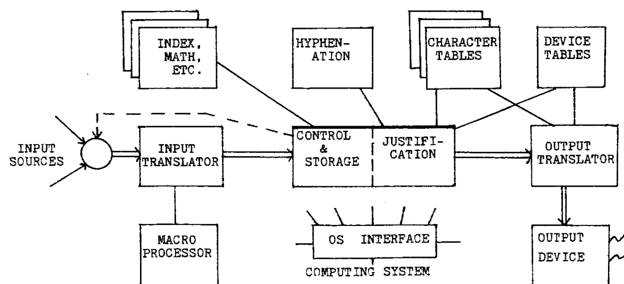


Figure 1—Modular formatting system architecture

The system kernel is responsible for control, sequencing, and storage. Input, initially from the standard source, is accepted by the input translator and passed to the kernel as a sequence of primitive commands. This input may include macro text to be deferred until a given condition occurs. The kernel will store the text and flag the appropriate internal variables. When a change to the required condition occurs, the saved macro will be rerouted through the input translator.

The received text is then "justified" in the context of the current environment. The kernel maintains a collection of named control records representing the environments, which can be switched on request. The information in these records includes format control parameters, current state descriptors, and user-controlled variables. Some information also remains in a global environment which is not switched.

The justification module constructs lines, determining line endings and processing various spacing parameters. Several algorithms of varying sophistication may be selected. The extent to which a single algorithm can be parameterized, and the degree to which this module can be unbound from the kernel, are subjects of current investigation.

Transformation systems are handled as distinct modules using a standard interface. They are viewed as collections of special commands which can accept arguments and insert text into the processing stream. Mechanisms such as indexing can thus be "piggybacked" onto MFS even if they were not planned in the original design.

The operating system interface module provides the only link to the underlying computing system, enhancing portability. Input and output routines, file name translation, date and time, and current invocation environment and options are all processed through this module.

The MFS output is a sequence of character selection and positioning codes in a device-independent format. However, processing at any time is done with the expectation of a particular output device and configuration. Characters are accessed with the understanding that they will be available on the actual output. In addition to character tables, a device table is used to determine significant characteristics of the output device. The minimum spacing increments are provided, for example; if these are coarse, it may be preferable during justification to round values at every stage of calculation. The ability of the output mechanism to back up, horizontally or vertically, may affect the processing strategy.

In a given environment, current output may be routed directly to the output stream or saved in a buffer. The deferred outputs may then be placed in the output stream at a later stage, perhaps with offset to a different position.

The output file, perhaps at a later time, is processed by an output translator which assembles coding for the particular output device, and generates the physical copy.

CONCLUSION

We have presented an architecture for a text formatting system, MFS, which is currently being developed at West Virginia University. The MFS design is more flexible than known current systems. It covers a wide range of applications although the program modules may be compact. Any of a wide

range of output devices can be driven with suitable translators and tables. Device characteristics are not built into the formatter.

A system with this architecture would enhance the interchangeability of document files and would be especially useful in environments where diverse applications may exist and a variety of output devices may be available over a period of time.

In addition, due to its modularity, MFS will serve as a test bed in which different hyphenation algorithms, command languages, etc., may be tried and compared. With other aspects of the formatter held constant, the effects of changes in a particular subsystem can be more easily studied.

REFERENCES

1. Saltzer, J. E. "Runoff." In P. A. Crisman (ed.), *The Compatible Time-Sharing System.* Cambridge, Massachusetts: MIT Press, 1965.
2. Pierson, J. *Computer Composition using PAGE-1.* New York: Wiley-Interscience, 1972.
3. Ossanna, J. F. *NROFF/TROFF User's Manual.* Comp. Sci. Tech. Rep. No. 54, Bell Laboratories, 1977.
4. Knuth, D. E. *TEX and METAFONT: New Directions in Typesetting.* Bedford, Massachusetts: Digital Press, 1979.
5. Reid, B. K., and J. H. Walker. *SCRIBE Introductory User Manual.* Pittsburgh, Pennsylvania: UNILOGIC, 1980.
6. Kernighan, B. "PIC—a Language for Typesetting Graphics." *ACM SIGPLAN Notices,* 16 (1981), pp. 92–98.
7. Goldfarb, C. F. "A Generalized Approach to Document Markup." *ACM SIGPLAN Notices,* 16 (1981), pp. 68–73.

MANAGEMENT ISSUES/
DECISION SCIENCE SUPPORT SYSTEMS

Complex business systems: a strategy for success

by NAOMI LEE BLOOM
American Management Systems, Incorporated
Arlington, Virginia

ABSTRACT

This paper deals with the development of complex business systems from two perspectives: (1) What role must be played by the project manager and other members of the project team to ensure a cost-effective and timely solution to the right business problems; and (2) What project organization and management techniques can be used to facilitate communications and decision-making among the project team, users, data center personnel, and other project participants.

INTRODUCTION

Complex systems do more than automate existing procedures. Such systems introduce new approaches to decision-making, force changes in job descriptions and in the organization itself, combine many separate (often organizationally distinct) tasks into a single black-box process, and challenge the business's rules-of-thumb with accessible, manipulatable data. For example, even a very large payroll system generally replicates once manual payroll calculations. However, a human resource management system might identify prospective management trainees by a complex weighting of their education and experience rather than by the personality traits so often used by human selectors.

When complex systems are installed, the organization is bombarded with change. Some changes—new forms, new reports, and a new chart of accounts—can be easily digested after proper training. But other changes must be anticipated—those such as widely different inventory levels produced by economic order quantity/reorder point systems or greatly changed hiring patterns produced by affirmative-action-based applicant tracking systems. Digestion of these types of changes requires corporate actions with a long lead time—e.g., building a new warehouse (or leasing excess space).

Although complex systems are often cost-justified on the basis of cost displacements, e.g., elimination of x payroll clerks, these displacements rarely occur. These systems are more properly viewed as opportunities for improved revenue streams, an enhanced quality of work life, greater productivity by the existing staff, and better decision-making. A primary benefit of an order-entry system that drives production scheduling is that customers more frequently have their requirements met on time. In today's competitive business climate, complex systems can mean the difference between survival and failure.

Large-scale (translate: complex) business systems projects have not been singled out in our industry for their overwhelming successes. Cost and schedule overruns, organizational disruptions, user alienation—all these and many more problems have plagued our efforts to develop complex management information and decision support systems. When such a project succeeds, and I've been fortunate enough to have been associated with several such successes, it's worth documenting who did what to whom (and when, and how) and why it worked.

This paper deals with complex systems from two perspectives:

- What roles must be played by the project manager and other members of the project team to ensure that the system successfully solves the right business problem in the most cost-effective and timely manner? (The roles of users and operations are discussed by Cox[1] and Jackson.[2]
- What project organization and management techniques can be used to facilitate communications and decision-making among the project team, users, data center personnel, and other project participants?

SYSTEM LIFE CYCLE

Each system progresses through a similar series of major steps, or phases. Once the content of each phase and their interrelationships are understood, the roles of project participants can be defined for each phase and a framework can be created for the effective planning and control of information systems projects.

To simplify integration of our three papers on complex business systems projects, we have adopted as a common terminology the AMS system life cycle (see Figure 1), which is divided into five phases:*

1. *Concept Definition* (Figure 2)—The model of the system that will solve the stated business problems is developed.
2. *System Design* (Figure 3)—The specifications from which the system will be built are prepared.
3. *System Development* (Figure 4)—The system design is transformed into programs and procedures, and it is demonstrated that the system meets the design specifications, working successfully in a controlled environment.
4. *System Implementation* (Figure 5)—The system is brought into operation in the production environment.
5. *System Support* (Figure 6)—Ongoing support is provided for production operations and system modifications, and enhancements are made over time.

In this approach, a complete picture of the entire system is developed from the very beginning—its business purpose, scope, structure, technical environment, costs, and benefits. As the development proceeds, the picture depicted in the concept definition becomes clearer and more focused as each piece is defined in increasing detail. But each of these detailed views ties in harmoniously with the overall framework envisioned in the concept. Thus, later development tasks are less likely to diverge from the main theme of the system. Further, this approach emphasizes both the business and technical aspects of a system from the outset of the life cycle.

In the *concept definition* phase the basic framework or model of a system is developed that will meet the user's business

*The life cycle phases and project management techniques presented in this paper were developed by and are proprietary to American Management System, Inc. (1981), 1777 North Kent Street, Arlington, VA 22209

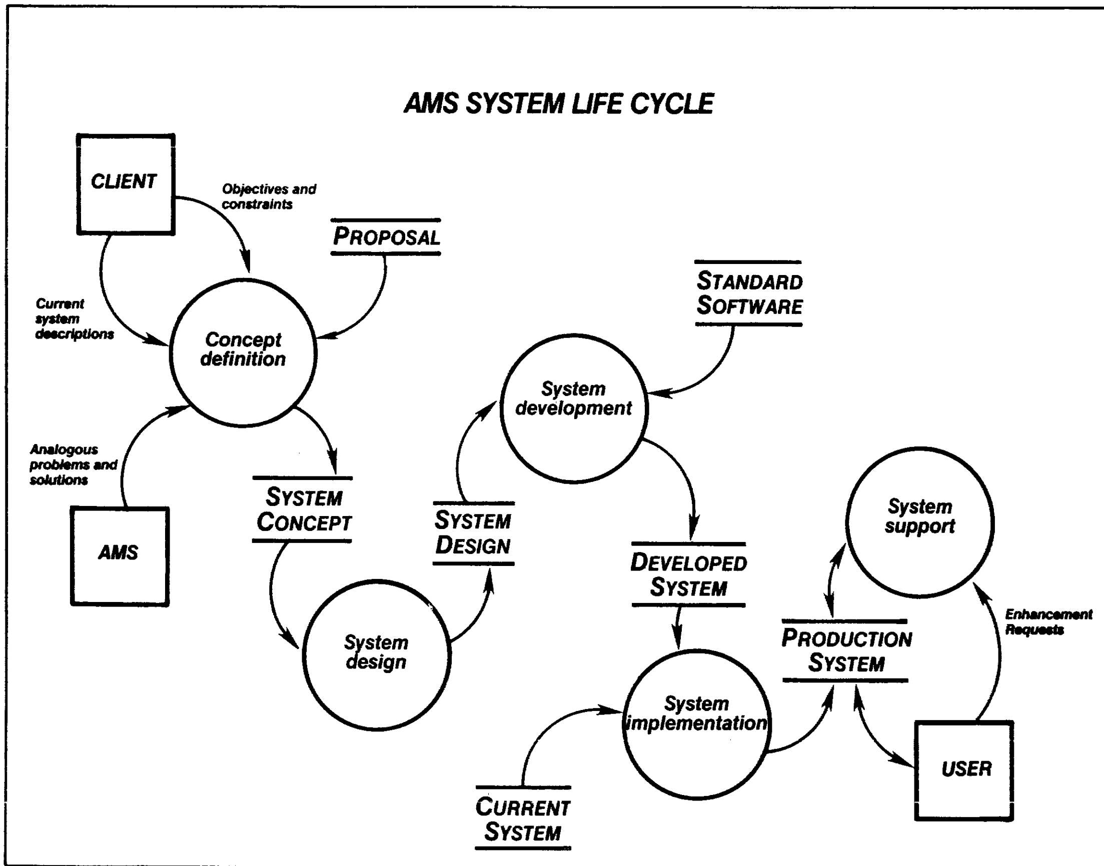


Figure 1—AMS system life cycle

objectives. The model synthesizes the elements of the new system—its aesthetics, scope, functional capabilities, organizational and budgetary constraints, and technology like an architectural model of a building. The system concept document provides an overview of the total system and shows how its various elements fit into a unified, workable solution to the user's needs.

The goals of the system concept are to evaluate the need for a systems solution to a management or business problem; provide the context in which informed decisions can be made on the numerous policy and procedural issues that must be worked out before the new system can be developed; and provide a road map to guide the activities of subsequent development phases.

In the *system design* phase the system model developed during concept definition is used to produce specifications for the system. This blueprint includes not just program designs, but all the components of and considerations affecting the new system, such as hardware and software configurations, user and operations procedures, implementation plans, and a detailed work plan for the subsequent phases of the system development effort.

In the *system development* phase the specifications developed in the system design are used to build a system that performs all the specified functions. The complete system is demonstrated to work in a controlled environment. At the completion of this phase, all developed programs, jobstreams, databases, and user and operations procedures are thoroughly tested by the project team. The computer and telecommunications facilities are installed, and the system is ready for implementation in the user's operating environment.

In the *system implementation* phase the new system is installed for the first time in the user's operating environment. At the completion of this phase, user personnel will have been trained to operate the new system, and the system will have been turned over to the production operations staff.

In the *systems support* phase the project team may provide supplemental expertise and resources to assist in operating the system and in evaluating and implementing modifications and enhancements. During this phase, ongoing efforts are directed at ensuring that the system meets performance objectives, software problems are repaired, and necessary enhancements are made to adapt the system to changing user requirements.

PROJECT MANAGEMENT PHILOSOPHY

Large and complex projects require strong and steady management for successful completion. Project management is an ongoing process that spans the system life cycle; it is a difficult process that involves the creation and management of a com-

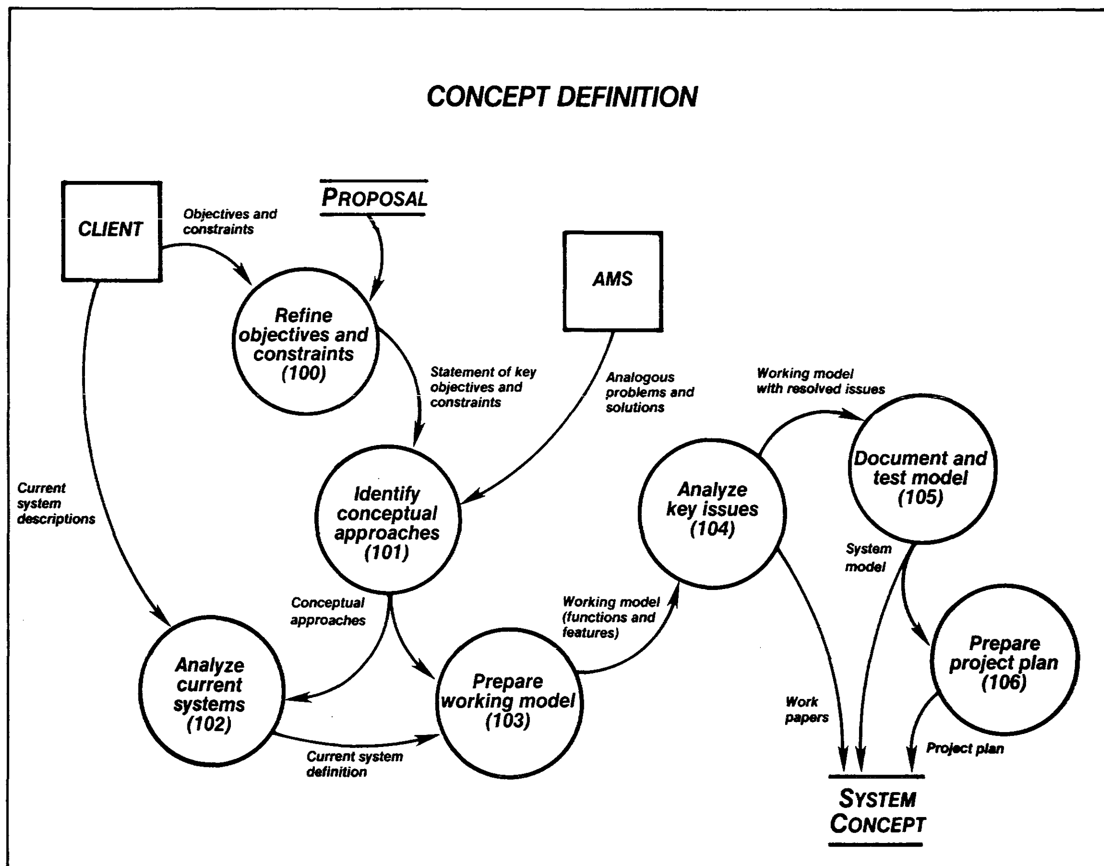


Figure 2—Concept definition

pletely new organization to produce something that has never before been built. This section describes the keys to our management style:

- Effective project management is substantive management, not mere administration. The project manager directs the development of the business and technical solution to the user's problems. The project manager must have a professional level of knowledge in all areas of system development and should be an expert in some of these areas. The manager's involvement with the substance of the project is the best assurance of a high-quality product.
- There must be a forum for discussing the numerous design issues that arise on any project and for resolving them so that decisions are clearly understood and carried out. We have found that a weekly team leaders' meeting, run by the project manager, is an effective mechanism for uncovering and resolving issues. These meetings foster close communication between the teams, reducing interface problems.
- Our approach to status reporting also supports our philosophy of active, substantive leadership. We use simple forms that show task and deliverable status in a graphic manner and measure completion in unambiguous terms of complete or incomplete. If work in one area should fall behind schedule, we take corrective action to facilitate completion of the problem task, to resolve issues holding up progress, and to shift resources to areas not affected by the late deliverable. We find our reports to be more useful for project control than many computerized tools based on detailed PERT charting or critical path management, although the latter are useful in initial planning. The system development process is flexible by nature. Many sequence dependencies that would be assumed in a PERT chart are soft; a task can often be started before its precedents are completed. We take advantage of this flexibility to work around problems.
- Finally, close coordination with the user is essential to our approach. The project manager plays a key role in the user design, reviews deliverables with the user, follows-up on user commitments, and keeps the user aware of the project's status.

PROJECT ORGANIZATION AND ROLES

While project organizations vary greatly in size and in team responsibilities, our large projects tend toward a standard structure as follows:

- *Project Staff*—The business analysts, technical analysts and programmer analysts who make up the project staff

are the backbone of the project. It is the staff's hard work, ability and enthusiasm that ultimately determine the success of the project. On medium and large projects, the staff is organized into teams of 3 to 10 professionals, each with a team leader. Teams are the basic working unit. A large project might have one team devoted to the programming of each major subsystem; a team devoted to testing; another team devoted to user documentation, procedures, and training; and a team working to prepare the production processing environment.

- *Project Manager*—The project manager takes day-to-day responsibility for all aspects of the project. The project manager takes the lead in working with users, in making design and major technical decisions, and in managing the quality and timeliness of project work. In the development and implementation phases, project management is usually a full-time job; in earlier phases, the manager may have other tasks, depending on the size of the team. The project manager is also responsible for planning, status reporting, and other administrative work. On larger projects, a project administrator may be assigned to free the manager from the details.
- *Project Supervision and Review*—At AMS, the project supervisor is a senior line manager, usually a vice president, who has overall project responsibility. The project supervisor reviews deliverables, provides input on substantive issues, takes the lead role in contract negotiations, and works closely with key users on management issues. All projects are formally reviewed on a regular basis by AMS corporate managers. Project reviews enable top management to communicate past experience to the project management, to exchange ideas on new approaches, to provide inputs at key decision points, and to detect and help respond to potential problems.

Based upon my own experiences, selecting a thoroughly competent project manager is one of the two most critical factors in helping to ensure a successful project; top management support and commitment is the other. To be effective, the project manager must have:

1. Substantive, expert knowledge of the project's business (translate: user) objectives;
2. Considerable delegated authority (or access to users or sponsors with authority) to commit user, data processing and external resources to the project;
3. Enough business experience to assess for the user when, how and in what ways the organization's current policies and procedures will change as a result of the project;
4. Sufficient technical (translate: computer) expertise to

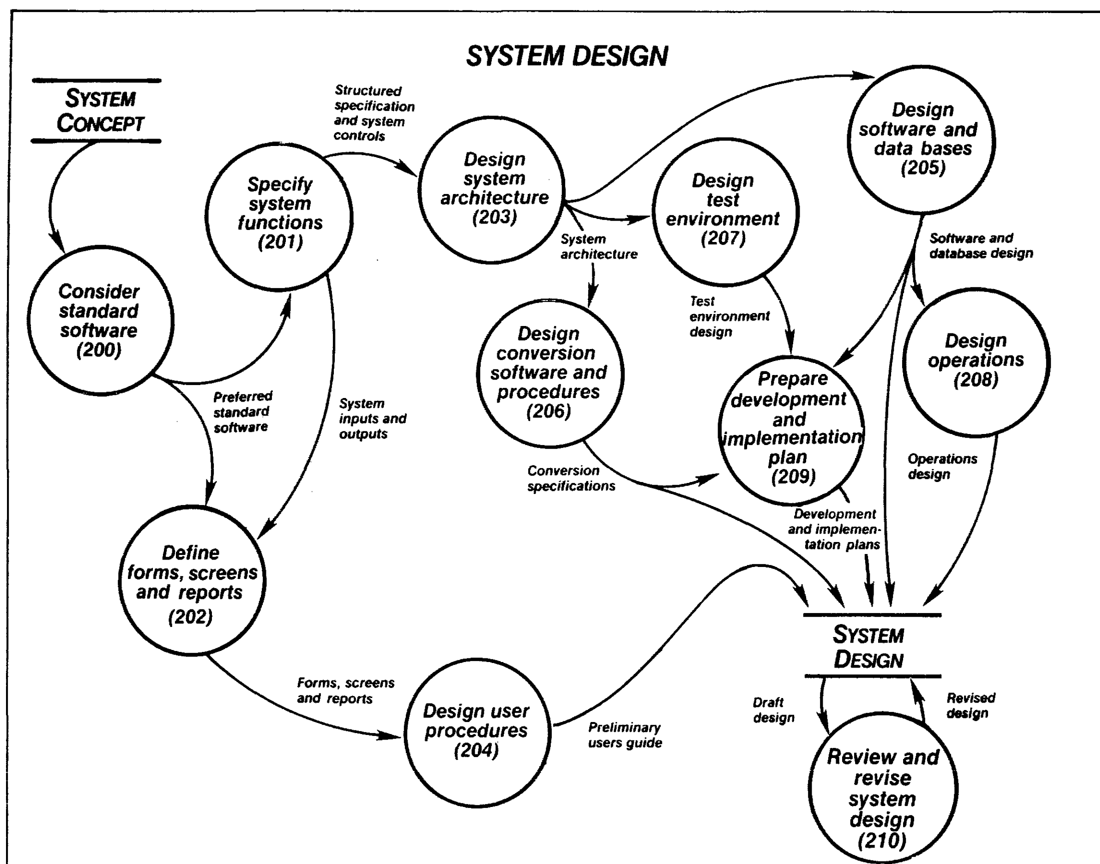


Figure 3—System design

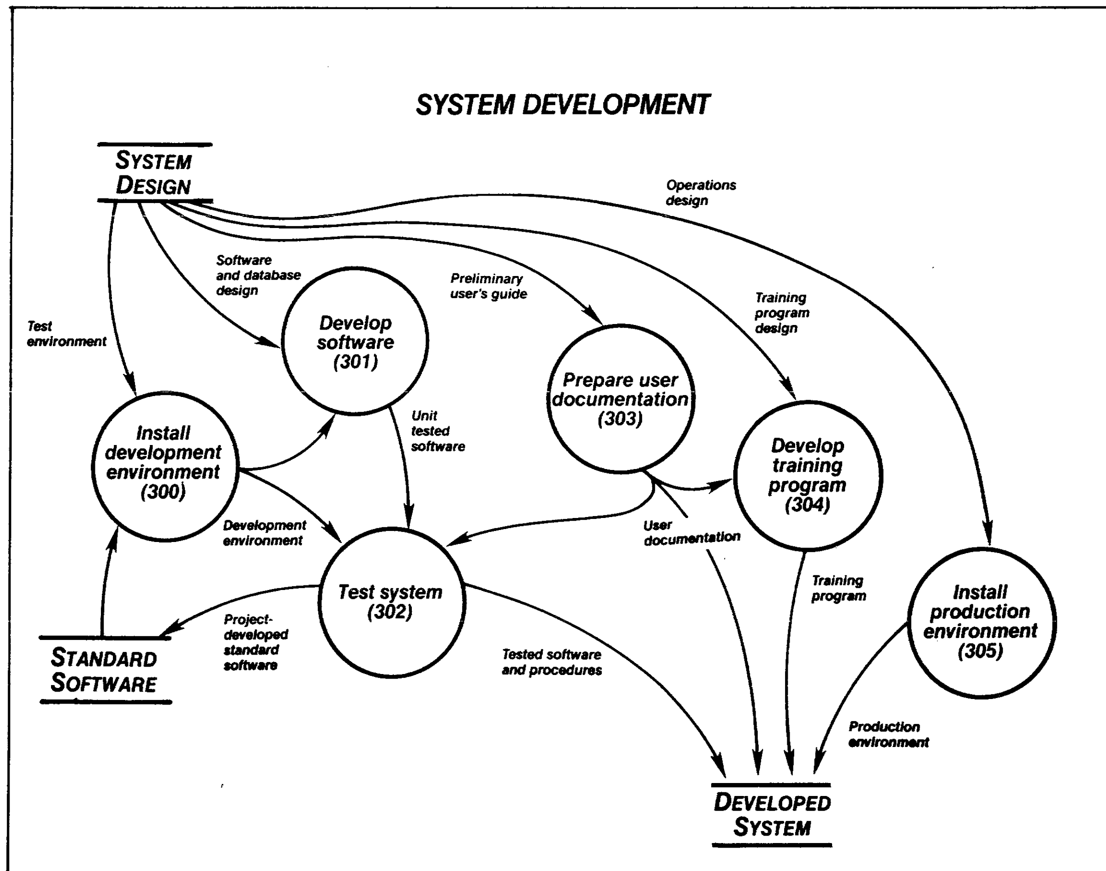


Figure 4—System development

manage a software development effort and to plan for, define and monitor changes to the operating environment;
5. Excellent people management skills, for the project will involve the active participation of staff ranging from entry-level clerical personnel to senior management;
6. The ability to move the project forward in the face of conflicting user requirements, insurmountable technical "glitches," contractual headaches, and all the other problems that will arise; and
7. The maturity to accept responsibility for problems while according the project team credit for successes.

OTHER ROLES

Successful projects are those which satisfy the properly nurtured business expectations of their sponsors (translate: users) and can be reliably operated by their inheritors (translate: data center personnel). In order to help ensure success, the project team must therefore include user and data center perspectives. The papers by Cox[1] and Jackson[2] address the roles of users and data center personnel throughout the system life cycle.

KEEPING IT ALL MOVING FORWARD

While there are no sure-fire management techniques that will compensate for a "just average" project team, even a really excellent team must operate in an orderly, well-documented manner. Effective project management requires a standardized approach to the basic management functions of *organizing, planning,* and *controlling* the project. This section describes approaches and techniques developed by AMS for carrying out each of these basic functions.

Organizing the project team is not a one-time effort. The project organization must be adapted to the changing availability of specific personnel, including user and data center personnel. Real people have skills and experience levels that never exactly match a theoretical organization; therefore, the actual project organization will be molded around the resources that are assigned. In addition, each phase of the system life cycle requires a different mix of skills, and, on long projects, there is usually some planned and some unforeseen turnover. Maintaining a viable project organization in the face of all these changes is a major task for the project manager.

Planning also occurs continuously throughout the system life cycle. Our planning approach includes three tasks:

1. Near the conclusion of each phase, develop a plan for the next phase which includes the tasks, milestones, and

budget. At the same time, develop or adjust the high level plan, which contains the approximate costs and schedule for the remainder of the project;
2. At the start of each phase, or at the start of each task area in larger projects, develop a detailed plan of task assignments and intermediate milestones for internal progress monitoring. Typically, these more detailed plans are done by the team leaders, helping to ensure their commitment to meet goals to which they and the project manager have agreed.
3. Revise plans and forecasts as work proceeds. It is essential to maintain an up-to-date plan and realistic forecasts of actual completion dates at all times. Changes in the system features, the project staff, or their assignments must be reflected in the plan. The forecast for the completion of milestones and for the project as a whole must be revised to reflect both changes in the plan and actual team performance. Team leaders usually update their task level forecasts on a weekly basis.

The manager's day-to-day involvement with the team's work is the fundamental means of *controlling* quality and progress. Nevertheless, a set of orderly processes is necessary for coordination and control. The project manager performs the following control tasks:

- Monitor and report project status. We use a standard set of project management reports which we have found to be highly effective in monitoring overall project status. A *Project Task Plan* shows the status of individual tasks. A *Deliverables Schedule* shows deliverables by week and pinpoints any that are late. A *Milestone Chart* provides a high-level visual display of the schedule, of any late deliverables, and of upcoming deliverables that may be late because of a late start or slow progress. Finally, a *Planned/Actual/Projected Staff Utilization Report* tracks actual hours against plan by task area. We use this report to maintain a current forecast in each area.
- Conduct team leader meetings. Every week, a team leader meeting is held to report on progress or problems, review deliverables, assign action items, and resolve issues. The meetings always have a published agenda and minutes. On small projects, a team meeting is generally substituted.
- Conduct client/user status meetings. We review the project status with the key client/user on a regularly scheduled basis. These meetings serve the purpose of informing the user of the status of deliverables and resolving any management issues. Meetings are attended by the user manager or managers, the project manager, frequently the project supervisor, and team leaders or members as appropriate. These meetings normally have an agenda, handouts, including status reports, and minutes.

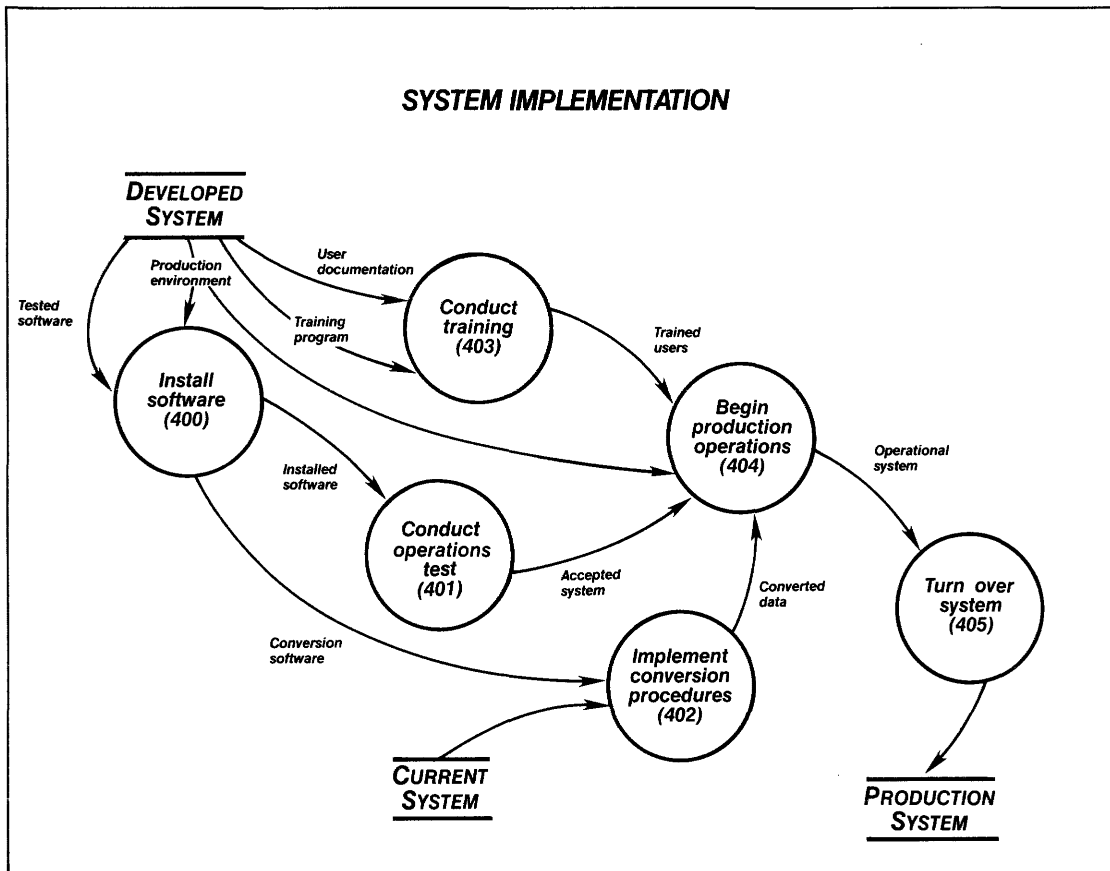


Figure 5—System implementation

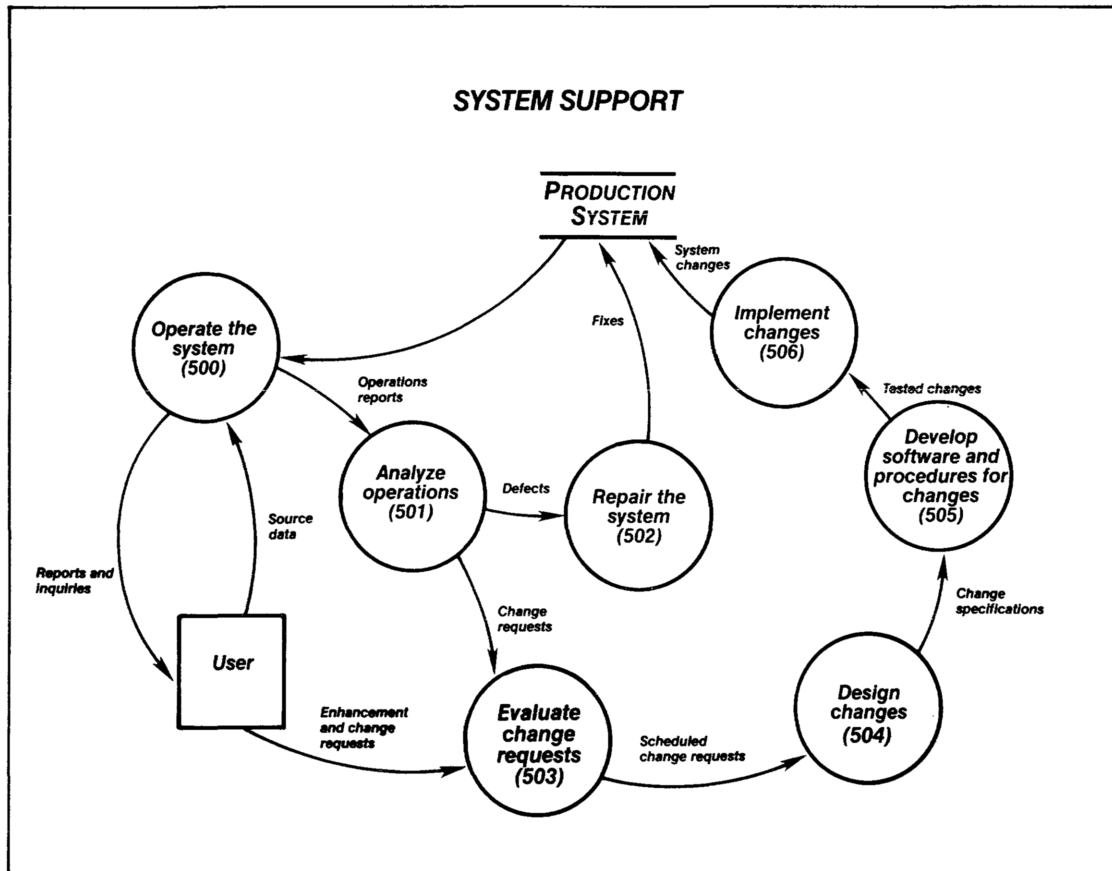


Figure 6—System support

- Conduct walkthroughs. Deliverables for all phases are reviewed, or walked through, internally and with the user. Internal walkthroughs are held as needed, and are typically attended by the team member's immediate manager, one or two other team members, and the project manager. These walkthroughs prevent any piece of work from proceeding too far without review. User walkthroughs are the key to ensuring that the system meets the user's business needs. Comprehensive user walkthroughs are major task areas in the concept and design phases. They are usually attended by the project manager, the team leader, the team member primarily responsible for the deliverable being reviewed, and the users whose responsibilities are related to the deliverable.
- Conduct project reviews. As mentioned above, project reviews are held regularly to ensure the input of top-level AMS managers into the quality, substance, and progress of the project.

CONCLUSION

There is no magic formula for ensuring a successful outcome for a complex business systems project. However, I believe that the probability of success can be substantially enhanced by doing the following:

1. Rigorously defining the responsibilities of all participants during each phase of the project in terms of specific, tangible deliverables;
2. Reducing miscommunication and/or documentation deluge through well-structured, formally recorded weekly team leader meetings;
3. Managing the expectations of users and developers alike by explicit resolution of design issues and constraints; and
4. Selecting a capable project manager with strong, substantive knowledge of the project's business objectives.

REFERENCES

1. Cox, David A. "The Role of Data Center Personnel in the Development of a Large-Scale Business System." *AFIPS, Proceedings of the National Computer Conference* (Vol. 51), 1982, pp.
2. Jackson, James E. "The Role of the User at Standard Oil Company (Indiana) in the Development of Large-Scale Business Systems." *AFIPS, Proceedings of the National Computer Conference* (Vol. 51), 1982.

The role of the user at Standard Oil Company (Indiana) in the development of large-scale business systems

by JAMES E. JACKSON
Standard Oil Company (Indiana)
Chicago, Illinois

ABSTRACT

Over the past 15 years Standard Oil (Indiana) has been involved in the development and implementation of a number of major computer-based business systems. Concurrent with this, Standard management has organized the users of these systems into a structure that provides for their effective participation in the development process. This structure, which consists of user management, user representatives, and the end user, recognizes that each of these groups has specific roles to play during the system development project life cycle. This paper will examine these user groups and their respective roles. The emphasis will be on identifying the key areas of user involvement that are necessary for the successful development and implementation of large-scale business systems.

INTRODUCTION

At Standard Oil (Indiana), management recognizes that for user organizations to achieve the greatest benefit from large-scale business systems it is essential that the user assume an active role in their development, implementation, and operation. This role requires that user organizations clearly grasp their role versus that of Standard's Corporate Information Services Department, which has overall technical and project leadership responsibility. Additionally, management recognizes that the user who is to effectively execute his/her role must be organized to participate efficiently. This philosophy is intended to ensure that, when large systems are implemented, they address the real requirements of operating personnel.

A fundamental concept in user participation in system development is that during the course of the systems effort there is a recurring series of activities requiring user involvement. This paper will discuss these activities and the associated role played by the user. Before doing this, it will first address the structure of the user organization as it relates to major systems development.

THE USER

User participation in systems development is structured around three groups—user management; departmental, or user representatives; and the end user. This organization exists to ensure that the right business systems will be built to address operating needs, that systems will be built in a cost-effective manner, and that major systems can be developed and implemented in a way that minimizes disruption of end user operations. The following is a discussion of the roles played by these groups to achieve these objectives.

User management is defined as senior management at the departmental level. Each operating subsidiary maintains an information systems development steering committee chaired by a fairly high-level user. (In the case of Amoco Oil, for instance, this person is the vice-president of planning and administration.) This steering committee is responsible for defining the aims, philosophy, and business goals of the organizations represented. Underlying this philosophy is the recognition that effective management of an organization requires good information systems. A key role of user management, then, is to identify how well existing business systems are supporting existing as well as future operating needs.

User representatives, typically members of steering committees, represent their specific departments on major systems efforts throughout the system life cycle. Their responsibilities include representing their department in determining and evaluating the requirements and constraints of the new system. They are particularly valuable in placing relative values on the requirements and constraints and prioritizing the various projects.

User representatives have the authority to draw upon members of end user organizations in the development process. However, this end user involvement is carefully planned and staged by user representatives so that there is a minimum of disruption to the end user's operation.

User representatives bring management and the end user into the system life cycle at the appropriate times. Basically, this is done to obtain their concurrence that the true business problem is being addressed and that the economics of the project are still valid.

End users are departments involved in the inputs, processing, and/or outputs of the production system. At Amoco Oil, which is a subsidiary company, typical end users are the Marketing, Manufacturing, and Transportation Departments.

USER INVOLVEMENT IN THE SYSTEM DEVELOPMENT LIFE CYCLE

Up to this point I have outlined the structure of the user organization as it relates to major system development. Next I will examine the roles of the three user groups in more detail. This will be done within the five phases of the system project life cycle: concept definition, system design, system development, system implementation, and system support. To guide this discussion, I have identified key points requiring the involvement of one or more of the user groups. (See Table I, "Points of User Involvement.") The following is a discussion of user responsibilities for these various points.

Project Request

At Standard, project requests for a major systems effort originate from a steering committee made up of user management and departmental user representatives. These requests are in the form of a feasibility study. In initiating a feasibility study, the user states the proposed objectives of the project, the economic justification for the project, and which departmental user representatives as well as end users will need to participate in the study. The feasibility study initiates the first phase of the system life cycle, concept definition.

Concept Definition

In concept definition a model of a system that will meet the user's current and future business objectives is developed. Included in this model are the elements of the new system, which are its scope, functional capabilities, organization

TABLE I—Points of user involvement

	User management	User representative	End user
Project request	X	X	
Concept definition phase			
Refine objectives and constraints for model		X	X
Review model for approval	X	X	X
System design phase			
Prepare forms and reports		X	X
Prepare user procedures		X	
Design conversion software and procedures		X	X
Prepare detailed project plan	X	X	X
System development phase			
Test the system		X	
Prepare user documentation		X	
Develop training plan		X	X
System implementation phase			
Conduct acceptance testing		X	X
Implement conversion procedures		X	X
Conduct training		X	X
Begin production operations			X
System turnover	X	X	X
System support phase			
Operate the system			X
Analyze operations		X	
Implement changes		X	X

and budgetary constraints, and technology. At Standard this is formally referred to as the project definition and authorization.

User involvement in concept definition includes refinement of system objectives and constraints, identification of gross system benefits, and review of the system model for possible approval.

The user representative, with the assistance of the end user, identifies his/her department's *requirements and constraints* in terms of the inputs, processing, and outputs that will be imposed on the new system model. In this effort the user—not the Information Services department—must be the final authority on the requirements.

Once the model has been developed, the user representative presents the model to the end user to obtain concurrence that the proposed system model satisfies end user requirements. The user representative then presents the model to the steering committee for approval to proceed into the system design phase.

The System Design Phase

In this phase user involvement focuses on a number of activities that will result in the development of detailed specifications from which the system will be built. Essentially, these activities are to refine the model developed in the concept phase.

Specifically, the user representative has responsibilities in the development of forms and reports, the preparation of user procedures, the design of conversion software for data codes, and the preparation of a detailed project plan to control the system development phase. Much of this involvement is to ensure that the evolving detail design is consistent with user requirements. Here the user must be watchful that the design of the Information Services department does not incorporate features that, though interesting, are not really needed. On the other hand, the user should not back away from the stated requirements merely because Information Services says they would be too difficult to implement. The user should also be careful that the design proposed by Information Services is sensitive to the user's dynamic environment and is flexible enough to allow the user to function without constantly having to make programming changes.

With respect to specific responsibilities, both the user representative and the end user are involved in the preparation, review, and approval of all forms and reports.

In the *preparation of user procedures,* the user representative identifies at what points the end user will have interaction with the new system. In addition, end user responsibilities at these various points are identified. From this identification, the user representative ensures that provision is made for appropriate user manuals.

In the design of *conversion software and procedures,* the user representative reviews and approves the conversion methodology for system files and databases prior to the cutover to a production system. This is reviewed with the end user to ensure that he/she concurs with the conversion strategy and has adequate resources when the conversion is made.

The *detailed plan* identifies the cost and timing attaching to the system development phase. The user must require that this plan, developed under the direction of Information Services, be comprehensive and in sufficient detail to allow for meaningful monitoring of system development progress. The user representative reviews this plan with user management and the end user to secure their approval to proceed into the system development phase.

System Development

The purpose of the system development phase is to realize in practice the system documented in the system design phase. Key user representative activities in this phase relate to system testing, preparing end user documentation, and developing a training plan. Generally, end user involvement in this phase is minimal. However, where the new system represents a radical change, the end user should be exposed to key systems concepts through training. This early exposure to the new system can significantly lessen the "cultural shock" the end user might experience in the implementation phase. At the end of this phase the new system will be installed in a production environment and will be available for end user acceptance testing.

In the *system test* the user representative participates as a member of the system test team, which has the responsibility for executing and evaluating the system test and, when completed, declaring the system ready for acceptance testing.

In the *preparation of user documentation,* the user represen-

tative reviews user procedures for technical correctness. As part of this he/she makes an effort to make these procedures as user-friendly as possible—i.e., remove the computerese.

In the *development of a training plan* the user representative works with the end user to identify a limited number of people who will be involved in acceptance testing. The strategy here is to have end users involved in acceptance testing and conduct the final training for their groups.

System Implementation

The *system implementation phase* brings the system into operation for the first time in the end users' real environment. Going into this phase, the new system has been tested to the satisfaction of the user representative. Key areas of user involvement in this phase are acceptance testing, conversion procedures, training, start of production operations, and system turnover. At the conclusion of this phase the new system will have been implemented in the end user department(s).

In *acceptance testing* the end user has an opportunity to assure that the new system will coincide directly with the design and expectations of his department(s). Since one of the primary purposes of acceptance testing is to allow the end user to "fine tune" user procedures, the user should require that Information Services have rough drafts of all user manuals at the start of acceptance testing.

The primary role of the end user in *conversion* is to verify the accuracy and completeness of created databases. The departmental user representative(s) will support the end user as needed in this effort, but responsibility for it rests with the end user.

The end users who were involved in acceptance testing will *train* appropriate personnel in their respective organizations. Experience has shown that the commitment to and acceptance of a new system is greater when the end users are trained by personnel from within their department who are knowledgeable in the new system.

There are a number of methods that can be used in *beginning production operations*—e.g., straight turnover and phased turnover. The method used depends on a variety of factors. However, the success of whichever method is selected will depend on how effectively the end user has participated in acceptance testing, conversion, and training. These activities are geared toward giving the end user operational experience before production operations begin.

Once production operations have stabilized to the point that early results indicate that the new system is running smoothly, the system can be *turned over* to the end user. Before this turnover the departmental user representative reviews the system with management and the end user to review results of initial production operations and to ensure that end user staffing requirements are sufficient to support the operation of the new system.

Life Cycle System Support

The concern of the user organization for the new system does not end at the date of turnover. One can be sure, even if the system developed is near perfect, that within a short time after turnover the first questions will arise. The *system support phase* gives recognition to this circumstance. In this phase the user has responsibilities in the operation of the system, in analyzing operations, and in implementing changes.

In the *operation* of the system, end user responsibilities include data preparation, entry and error correction, system administration report distribution, and production troubleshooting. The impact of production troubleshooting will vary directly with the flexibility of the system design. The user can save much inconvenience and expense by reviewing the detailed design of Information Services from this perspective in the system design phase.

The end user is responsible for *analyzing operations* by analyzing system performance, error rates, procedural problems, etc. Where appropriate, the end user, working through the departmental user representative, will need to initiate system tuning and maintenance requests.

When maintenance requests have been completed, the end user, supported by the user representative, is responsible for *implementing changes*. This involvement includes acceptance testing, training end user personnel, and conversion of databases. These activities are similar to user involvement in the implementation phase, but on a smaller scale.

SUMMARY

This paper has examined the role of the Standard Oil user in large-scale business systems development. The intent has been to identify the key areas of user involvement that are necessary for the successful development and implementation of such systems. These key areas have been discussed within the life cycle of the system.

In addressing this involvement, particular attention has been given to the structure of the user organization—management, the user representative, and the end user.

The role of data center personnel in the development of a large-scale business system

by DAVID A. COX
Selective Service System
Washington, D.C.

ABSTRACT

The purpose of this paper is to identify the role that data center personnel should play in the early phases of the development cycle and to highlight special areas of concern in the later phases of the development cycles. In order to provide a framework which will encompass the majority of situations, I have chosen to describe the role of data center personnel in the context of the systems life cycle of a large-scale, complex business system. It is hoped that the result of this paper will be to define a role for operations personnel that is as visible and influential in the early stages of the systems development life cycle as it is during the implementation stage.

INTRODUCTION

Historically, data center personnel are asked to provide hardware, system software, and personnel support for new systems even though they have not had the opportunity to participate in the early stages of the system development cycle. The lack of participation by data center personnel in the early stages of development can have a significant impact on the implementation stage of the development cycle and can cause project completion dates to slip, costs to escalate, and the quality of the final product to be reduced.

The purpose of this paper is to identify the role that data center personnel should play in the early phases of the development cycle and to highlight special areas of concern in the later phases of the development cycles. In order to provide a framework which will encompass the majority of situations, I have chosen to describe the role of data center personnel in the context of the systems life cycle of a large-scale, complex business system. It is hoped that the result of this paper will be to define a role for operations personnel which is as visable and influential in the early stages of the systems development life cycle as it is during the implementation stage.

It is important to note that the system development process, while straightforward by definition, is in practice a very complex iterative process. The best analogy of this process is to compare the solving of a business problem to that of peeling an onion. As each layer of the problem is "peeled" away and examined, it reveals more problems which need to be solved. As each layer of the problem is removed, examined, and solved, the best approach to solving the business problem becomes increasingly apparent.

The continual "peeling" away of the problem means that the same type of analysis must be performed several different times, each at a different level of detail. It is this continual refinement in the analysis process that when properly executed, results in the successful installation of a system that truly solves the business problem.

DEFINITIONS

For purposes of the system development process, the term *data center personnel* should include the following:

1. Hardware/system software specialist—a person who understands the interrelationships of the hardware/operating system currently in place and those available in the market place today; and, their ability to support new hardware needs.
2. Telecommunications specialist—an individual who can perform the network analysis necessary to determine the optimal telecommunications design and develop a workable design within resource and technology constraints.
3. Procedural specialist—an individual who understands and can identify the processes required to handle exceptions, correct errors, and schedule production efforts.
4. Procurement specialist—an individual who can identify the procurement cycle, lead times and costs for each new system component, and lead the procurement effort.

For purposes of this paper the system life cycle consists of the following five major phases:

1. *Concept definition*—the model of the system that will solve the stated business problems is developed.
2. *System design*—the specifications from which the system will be built are prepared.
3. *System development*—the System Design is transformed into programs and procedures, and it is demonstrated that the system meets the design specifications by working successfully in a controlled environment.
4. *System implementation*—the system is brought into operation in the production environment.
5. *System support*—ongoing support is provided for production operations and system modifications, and enhancements are made over time.

Concept Definition

In the *Concept definition* phase, the basic system framework (its aesthetics, scope, functional capabilities, organization and budgetary constraints, and technology) is developed. Like an architectural model of a building, the system concept document provides an overview of the total system and shows how its various elements fit into a unified, workable solution to the business needs of the organization.

The goals of the system concept are to evaluate the need for a systems solution to a business problem, provide the context in which informed decisions can be made on the numerous policy and procedural issues that must be worked out before the new system can be developed, and provide a "road map" to guide the activities of subsequent development phases.

During this phase there are many activities taking place that as independent activities do not directly affect data center personnel. However, when viewed in the aggregate, these activities have a major impact on the operations area of the data center. The primary mission of data center personnel during this phase is to determine the feasability of the system concept from a technical standpoint given the real world constraints of available technology (hardware, telecommunication, and software) and the impact of the new system on the support and operation of existing data systems. This role is critical to the long-term viability of the project, since the

system concept described by the user/project manager very rarely exists in a pure form that is either readily available in the existing system or is a transparent addition to existing facilities. Rather, the solution must be molded to fit the technology available to the data center either through adaptation of existing systems or through procurement of new systems. Therefore, the data center personnel must know as much about the goals of the system as the design team if they are to be able to find alternate architectures that can be made to fit the requirements of the new business system.

Selecting a hardware and system software architecture which can meet specific requirements with only a conceptual system design in place is a difficult, yet necessary chore. The key issue is to determine if the new application can fit into existing hardware, or whether system upgrades or new computers must be procured to handle the new workload.

The data center team must concentrate on specific tasks that must be performed in parallel with the system concept tasks being performed by the business and system analysts. These tasks are described more fully below.

1. *Establish policy guidelines for service levels, system life, and operational requirements.*
   In defining the service orientation of the new system, the operation team needs to know if the system is to be user-oriented or production-oriented. A production-oriented system will use as much of the capacity of a machine as possible through prior planning of production workloads, whereas a user-oriented system must be geared to provide a high level of response and service even under unplanned peakload conditions. This difference in service requirements will have a dramatic effect on the sizing of the eventual system, since a user-oriented system will require more resources than a production-oriented system. The level of service must be expressed in quantitative terms, such as transactions per minute in a batch system or response time in seconds in an online system.
   Once the service orientation has been fully defined, the system life and general requirements for an operational window (i.e., 1st year, 1 shift per day; 2nd year, 2 shifts per day; 3rd year, 24 hours per day operation) must be defined. Once these items are defined, the team will have the basic parameters necessary to evaluate the impact of the current and planned production on the current hardware/system software facilities.
2. *Identify the present workload.*
   Unless the decision has been made to use a new computer system exclusively for the new application, the processing characteristics of the current workload must be fully understood. Quantification of this workload will include physical requirements (i.e., disk space, telecommunication support, etc.) and processing requirements by application type. The analysis should include requirements over the system life of the application, peak load requirements and the service level requirement for each application (i.e., user vs. production).
3. *Forecast future workloads.*
   Forecasting the workload of the new system is an iterative process and may require the development of several alternative system models and the preparation of workload estimates for each of these production models. At this stage, the models developed will be described at a high level, with general descriptions of cost and capacity requirements. Examples of alternative models may include: (a) a centralized database system with online terminals used for updating and file query and all edits done on the host computer; (b) a centralized database system with remote intelligent terminals accessing the central database on a dial-up basis and most editing done on the terminal; and (c) a distributed database system with remote terminals hooked to distributed processors and high-speed data links to the central computer.

General workload estimates must be developed for each of these alternative models. These estimates should be detailed enough to identify the following:

1. Volume of transactions per node in the system model
2. Volume of mass storage required
3. Network patterns and approximate cost
4. Service level required
5. Production window requirements
6. Phased growth of transactions over time

By applying workload levels to each system model, each node (i.e., terminal, CPU, storage, etc.) in the system architecture can be sized and cost with a nominal level of precision ($\pm 20\%$).

This system workload data will be analyzed and presented to the user/project manager in a report which describes the alternative architectures by their costs, benefits, disadvantages, procurement lead times, and impact on existing systems. The project team must then select the basic system architecture to be used as the basis for subsequent design efforts. The system architecture decision must be made early on in the planning process, since the procurement cycle (specification development, evaluation of bids, delivery time, initial installation, hardware test, system software installation, system software test) for a new system may take anywhere from 6 to 36 months, depending on system complexity and manufacturer lead times. The information regarding lead times for procurement must be incorporated into the system concept so that the concept that is approved by top management can be delivered on time.

System Design

In the *system design* phase, the system model developed during the concept definition phase is used to produce specifications for the system. This "blueprint" includes not just program designs but all the components of and considerations affecting the new system, such as hardware and software configuration, user and operations procedures, implementation plans, and a detailed work plan for the subsequent phases of the system development effort.

Specific Activities include the following:

1. Specify system functions
2. Design system architecture

3. Define forms, reports, and screens
4. Design software and databases
5. Design conversion software and procedures
6. Design test environment
7. Develop detailed operational concept
8. Prepare detailed project plan
9. Review and revise design
10. Prepare procurement documents for hardware and system software

There are several key activities where the data center personnel must take the lead. These include developing detailed estimates of the system workload, preparing technical specifications for the hardware and systems software required to meet the application needs, identifying the telecommunications network to be used for the system, and preparing the test environment for system development. In order to complete these types of activities, the operations personnel must perform the same type of analysis that was required during the concept definition phase. The primary effort will be to evaluate alternative equipment/system software configurations that will support the system architecture approved in the concept definition phase.

In addition, the workload estimates will be more detailed and will result in estimates that have a much greater level of precision than those developed during the concept definition phase. The results of this analysis will be used as the basis of the procurement documents that must be developed for the acquisition of the hardware and systems software.

The level of specificity required for procurement documents will vary depending upon the nature of the procurement. If the procurement is to be fully competitive (i.e., any vendor may qualify—IBM, Univac, Honeywell, etc.), the specifications that are prepared must be functional and not vendor-specific. However, if the solicitation is designed to augment existing equipment or be compatible with existing equipment, the specification can, and should be, much more detailed and be system-specific.

In the fully competitive procurement, the analysis that provides system-specific specifications must be performed after the vendor is selected. Because this activity is in the critical path, having to wait for the procurement decision can delay implementation times. Once the system-specific specifications are developed (either before or after the procurement), the balance of the operations activities in the system design phase can be planned by the data center personnel. These activities include developing a test environment and planning the installation of system software for the new system. In the case of augmentation to an existing system, it may also require planning for the conversion of existing systems to new system software and modifying production schedules to accommodate the development efforts of the new system.

System Development

In the *System development* phase, the specifications developed in the system design phase are used to build a system that performs all of the specified functions, and the completed system is demonstrated to work in a controlled environment.

The activities performed during this phase by data center personnel are classic operations activities. The specific tasks will be driven by the implementation plan but, in general, include the generation of new executive and systems software modules, preparing new operating procedures for test and evaluation, and providing special support for the development team. At the completion of this phase, all developed programs, jobstreams, databases, and user and operations procedures are thoroughly tested by the project team. The computer and telecommunications facilities are installed, and the system is ready for implementation in the user's operational environment.

System Implementation

In the *System implementation* phase, the new system is installed for the first time in the user's operational environment. At the completion of this phase, all personnel will have been trained to operate the new system.

In the case of large network terminal-based systems, the time involved in the physical installation of equipment can be very lengthy, since there are many pieces of equipment to be installed in many different locations. This task is made even more complex because the installation must be coordinated with the vendor, the telephone company, the user, the building managers for electrical connections, and the training team which must train all of the users on the terminals as they are installed.

In addition to the tasks involved in training and equipping users, the new workloads caused by adding new users will have an effect on the data center operation. Invariably, this activity will cause some imbalance in the system that was previously unplanned and that will require modifications to the installation schedule. These variances in schedules and unexpected problems must be fully coordinated with the project team in order to ensure that the implementation effort is continuing on schedule.

The orderly transition to the new system is the most important part of the development cycle from the users viewpoint, and every effort must be made to ensure success. Very often at this stage of the project, the development team is tempted to start working on a new project and ignore the final phase of implementation. As a result, the data center support team is called on to provide an increased level of support to solve start-up problems. It is critical that the data center team have members of the development staff available to catalog and help solve problems as they arise during this stage, since the users can best identify deficiencies which need to be corrected in subsequent release of the system software.

System Support

In the *System support* phase, ongoing efforts are directed at ensuring that the system meets performance objectives, soft-

ware problems are repaired, and necessary enhancements are made to adapt the system to changing user requirements.

During this phase, which lasts for the system life, a key responsibility of the operations area is to monitor system usage and performance. The statistical data gathered serve two purposes. First, the results should be compared to the performance requirements defined in the system concept and design phase to determine if the original goals are being met. Second, the data should be accumulated to determine the operating characteristics of the system so that in the future, when other "new systems" are proposed, the operating data are available as input into the concept definition phase of the next system.

SUMMARY

If a major automated business system is to be successfully installed, it is imperative that all members of the development team play an active role in all phases of the development cycle. Only through rigorous compliance to the team concept can all members of the development team share the same vision of a system that starts as a "gleam in the eye" of a user and ends up as a complex, highly sophisticated data processing system.

By being involved in the early stages of the development cycle, the data center team can provide the hardware and system software support necessary to turn the user's idea into a working reality.

What life? What cycle?

by NICHOLAS ZVEGINTZOV
Staten Island, New York

ABSTRACT

The traditional system life cycle model does not portray the life of a system, nor is it a cycle. An alternative model is described that portrays the modification cycle of the system and the detailed activities of making a change. Implications are drawn for maintenance, development, and the education of software engineers.

THE NOT-LIFE NOT-CYCLE

During the 1970s the phrase *system life cycle* came to be widely used to describe the stages of growth of an applications system. For a while the phrase was almost a fad or a buzz word. As Glass and Noiseux say in their *Software Maintenance Guidebook:*[2]

> At a recent computing conference, discussion of the so-called computing life cycle became a standing joke. Every presenter of every paper showed a viewfoil or a slide containing his or her graphic version of the concept. Toward the end of the day, one wag referred to his as the "obligatory software-life-cycle chart"!

These charts had the basic form of Figure 1, in which a system comes into being by being elaborated or made concrete in a sequence of phases and finally is installed and enters an operation and maintenance phase. The exact number and names of the phases sometimes varied, but the basic structure remained the same. The arrows were generally downward to indicate the management constraint that each phase must be frozen before the next is started, although sometimes feedback arrows were added to indicate that this was not always possible to enforce (Figure 2).

A Finnish version[3] even offers a more elaborate pattern of feedback (Figure 3).

Popular though this model is, there are two objections to calling it a life cycle:

1. It does not portray the system's life.
2. It is not a cycle.

First, it portrays only the creation, development, or youth of a system, and does not include its adulthood—the productive phase of its life. It is as vague about the operation and maintenance phase as a teenager is about life after marriage.

Second, it is a linear path or progression from goals to operation and maintenance; and it does not, as a cycle must, in some sense return to its own beginning. In borrowing the term *life cycle* from biology, the originators of this model failed to borrow its central concept, the tracing of the organism from its embryonic origin to the adult state in which it originates and nurtures the embryo of a new individual.

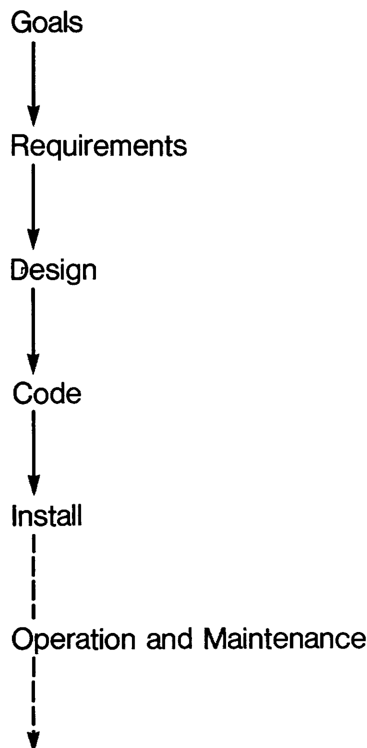


Figure 1—"Life cycle"

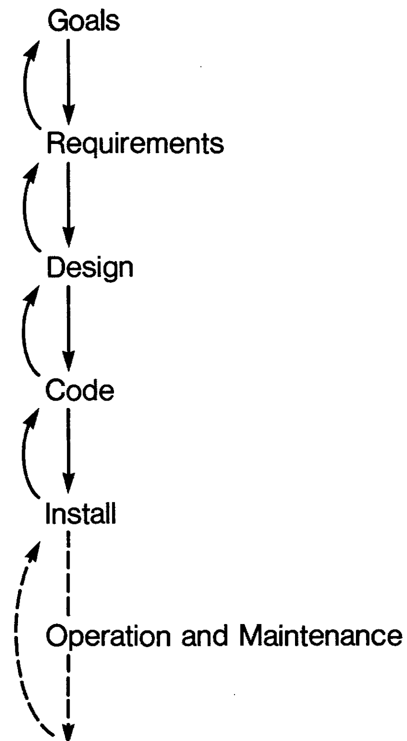


Figure 2—"Life cycle" with feedback

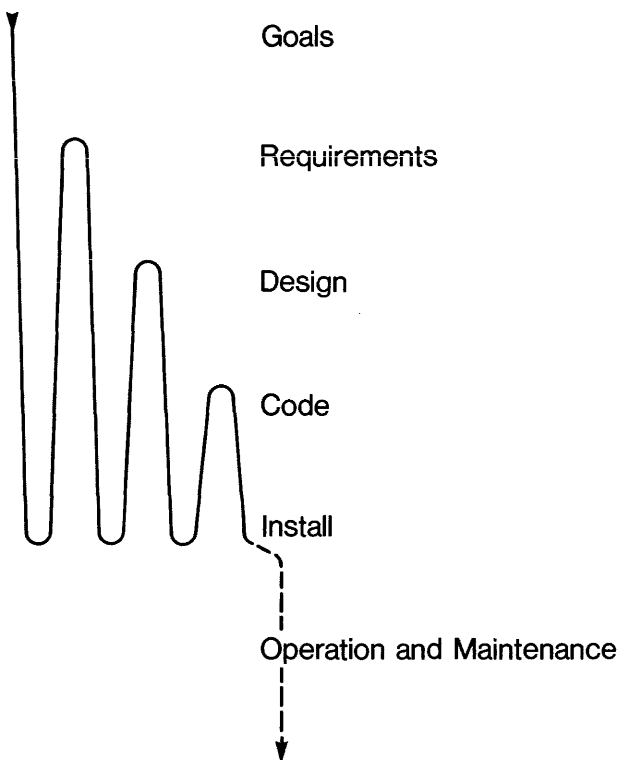


Figure 3—Finnish "life cycle"

Thus this model is misnamed if it is regarded as a model of the system life cycle. It is, in fact, a model of the development path of a system—in fact, as I shall show in this paper, of *part* of the development path. Nevertheless, it has had great popularity, since, even misnamed and partial as it is, it helps to illuminate and analyze (and hence make controllable) a significant and expensive portion of the system's life. Can we supplement it by finding an equally illuminating model of the rest of the life of the system?

THE MODIFICATION CYCLE

The first step is to use the clue given in the name of the last phase of the development model—operation and maintenance. There are two terms here, representing two activities. The system does its job (operation), and it undergoes modification (maintenance). A relatively small part of this modification consists of repair; the rest consists of changed or enhanced function. (Readers who doubt this about software systems are referred to the Lientz and Swanson and U.S. General Accounting Office questionnaire studies.[4,7])

The general categories of such changes, for organizations as well as for computer systems, are as follows:

1. New, changed, or deleted functions
2. Adaptation to environmental changes (legal, financial, political)
3. Consolidation, reorganization, routinization
4. Turnover of staff, equipment, resources

A simple model of the cyclic incorporation of such changes is portrayed in Figure 4. During the system's operation, its constituents (owners, managers, operators, users) *assess its performance*. On the basis of this assessment, they generate *requests for change* related to their own interests. These, after political tradeoffs and the commitment of resources, become *modifications* of the system. These modifications affect the system's performance, which the constituents assess, and the cycle continues.

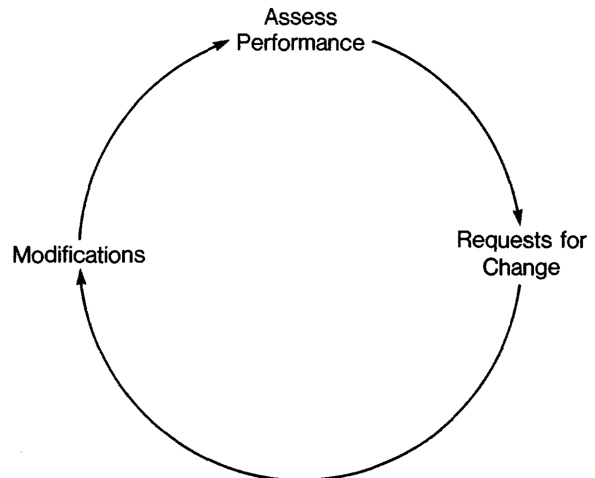


Figure 4—The change cycle

This model *is* cyclic and *does* deal with the adult life of the system. The developmental model can be grafted onto it to form what I call the *starting gate model,* because development appears as a one-time initiation, after which the system continues infinitely around the cyclic track (Figure 5).

This model, though better than the last, is still inadequate as a portrayal of the life cycle. In particular, the cyclic part of the model appears impoverished and undifferentiated compared to the richness of structure shown on the developmental starting gate. Have the developmental stages of goals, requirements, design, and code anything to tell us about the modification of existing systems? Certainly. Although they do not appear as developmental stages, they are the analytic framework we need for comprehending (and therefore controlling) an existing system. With this hint, we can proceed to fill in the detail of the modification cycle.

LEVELS OF DESCRIPTION

Why is development performed in stages? Simply because the gap between goals and code is too great to be crossed in a single intellectual leap. Therefore the process is converted into a chain of stages chosen so that the gap between each stage and its successor *can* be crossed.

Clearly, the same problem exists in understanding existing systems. The gap between high-level performance and low-level implementation is too wide to be crossed in one leap. The leaps must be narrowed by interposing various intermediate layers to aid understanding; these are the *levels of description.*

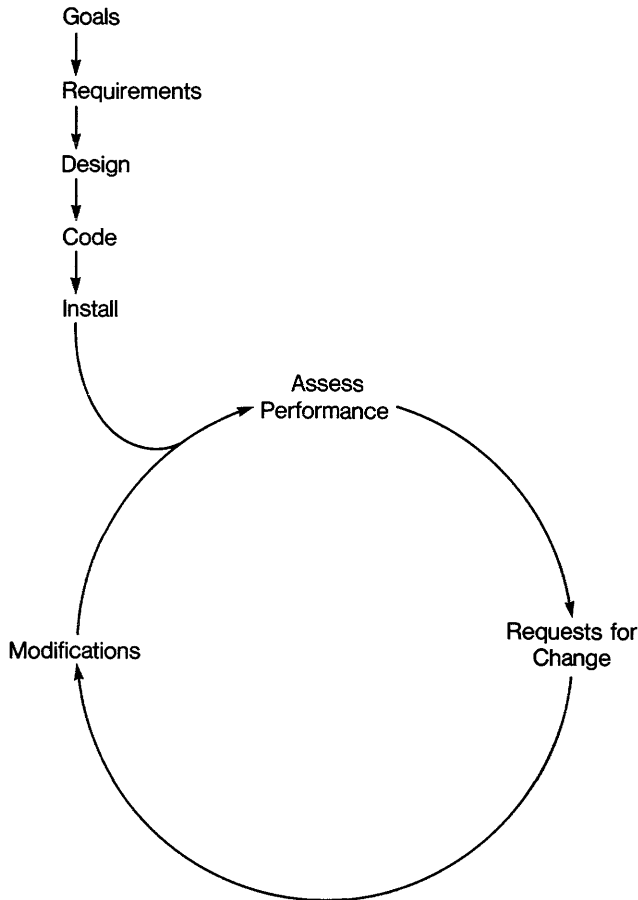


Figure 5—The starting gate model

The levels of description are alternative and simultaneously correct descriptions of an existing system from different perspectives. In general, a higher-level description answers the "Why?" question for a lower, a lower answers the "How?" question for a higher, and the levels of description roughly correspond to levels of management. Each level has a characteristic vocabulary appropriate to the people who deal at that level, and each level contains motivating information not directly derivable from any other level. Thus the levels contain information-hiding decisions in the sense used by Parnas,[5,6] and they correspond to the "knowledge domains" found by Ruven Brooks in documentation.[1]

In Figure 6 the levels of description are shown under an existing system; the stages of development are shown under a desired system; and labels are given to equivalent levels, stages, and management roles. (This particular example is of a system of a scale large enough to match the major operations of the organization, and therefore its goals reach as high as the chief executive officer; but there are many systems with more modest goals reaching less high in the hierarchy.)

1. PERFORMANCE/GOALS/CEO: "We (need to) have an inventory control system for our warehouses and distributors."
2. CAPABILITIES/REQUIREMENTS/V-P: "I (need to) control the regional warehouses at.... "

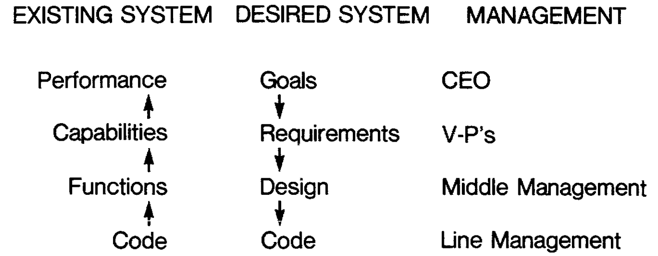


Figure 6—Levels, stages, management roles

3. FUNCTIONS/DESIGN/MIDDLE MANAGEMENT: "I (need to) have staff, procedures, and equipment to handle picking, shipping, reordering, charging.... "
4. CODE/CODE/LINE MANAGEMENT: "I (need to) have terminals and display/update software for my fork-lift operators...."

On both the Existing and the Desired sides the levels (stages) describe (define) the same system, but there are two differences in the way that they relate to each other.

First, the levels of description exist simultaneously—they represent the state of the system as viewed today at different levels of management. By contrast, a desired system goes through the stages of development in sequence. This difference contrasts the actuality of an existing system with the futurity of a desired system.

Second, the identity of the levels of description is imposed from below, whereas the identity of the stages of development is imposed from above. The managers of an existing system must understand it, warts and all. If a higher-level description does not accurately abstract a lower-level one, it must be revised. By contrast, a desired system is described in terms of intention. If a later stage does not accurately implement an earlier one, it is reworked until it does. This is why the arrows are upward on the Existing side and downward on the Desired side.

The levels of description have brought into the analysis of an existing system the richness of detail found in the development model. How do these levels interact with the change process to give a model of the modification process?

HOW TO MAKE A CHANGE

The stages of making a change are as follows:

1. Understand the request.
2. Transform the request to a change; the change is the goal.
3. Specify the change: choose cut-line and patch.
4. Develop the patch.
5. Test.
6. Install.

The first stage in making a change is to *understand the request.* A request is a description of a new system, phrased in

terms of the existing system and in the vocabulary of a level of description:

"Computerize inventory!"

or

"Get ready for the new Denver warehouse."

or

"Put I/O devices on the forklifts."

Understanding a request (Figure 7) requires (a) understanding the system via its levels of description, and (b) running the request down the levels of description until it finds a place where it makes a difference.

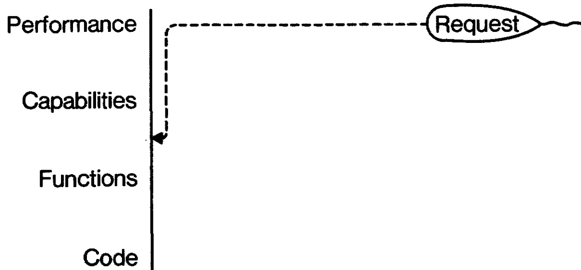


Figure 7—Understand the request

One may assume that the request "Computerize inventory!" affects the system even at the level at which the CEO views it. However, the request "Get ready for the new Denver warehouse" would cause only minor changes in a high-level document that observes that "All regional warehouses are computerized.... " Going lower still, the request "Put I/O devices on the forklifts" does not affect the CEO or the V-Ps, but only the regional managers and below. Thus the process of understanding a request is a process of working down the levels of description to find the highest level affected.

TRANSFORM THE REQUEST TO A CHANGE

The second stage in making a change is to *transform the request to a change* (Figure 8). The action here is to apply the request to the description of the existing system and derive an alternative description of a desired system, i.e.:

1. Given the existing inventory system, what would a computerized one be?
2. Given the existing computerized system, what would one be that includes the new Denver warehouse?
3. Given the existing in-warehouse system, what would one be that includes I/O devices on the forklifts?

At the point at which the request becomes a change, the system begins to bifurcate: above it there is no distinction between existing and desired, below it there is one. Closing this bifurcation is the goal of the change process. (Note that this goal is usually at a level much lower than the goals of the overall system.)

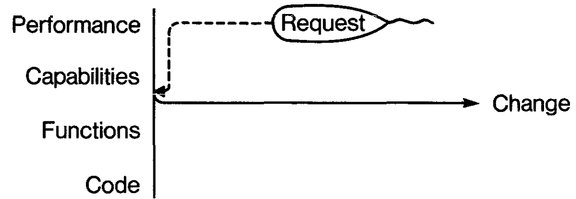


Figure 8—Transform the request to a change

SPECIFY THE CHANGE: CUT-LINE AND PATCH

The third stage in making a change is to *specify the change;* this involves choosing the *cut-line* and the *patch* (Figure 9). The cut-line is existing code or procedures which must be changed; the patch is new code or procedures. In the following simple example, the cut-line is the sentence from IF to PERFORM SECTION-ELSE, and the patch is SECTION-Y.

EXISTING CODE
```
IF INPUT="X"
   THEN PERFORM SECTION-X


 ELSE PERFORM SECTION-ELSE.
...
SECTION-X. (TEXT)

SECTION-ELSE. (TEXT)
```

CHANGED CODE
```
IF INPUT="X"
   THEN PERFORM SECTION-X
 ELSE IF INPUT="Y"
   THEN PERFORM SECTION-Y
 ELSE PERFORM SECTION-ELSE.
...
SECTION-X. (TEXT)
SECTION-Y. (TEXT)
SECTION-ELSE. (TEXT)
```

Choosing the cut-line is the major intellectual challenge in making a change. The cut-line has less code than the patch, but it has greater complexity, since it derives complexity from its intimate interaction with the complexity of the main system. The designer has two aims in choosing the cut-line:

1. Minimize the impact of the cut-line on the existing system.
2. Isolate the patch from sources of variability in the existing system.

It is important to minimize the impact of the cut-line, because it directly affects the fabric of the existing system. In changing a system, as in surgery, the major challenge is not causing the desired alteration in the organism, but avoiding undesired alterations.

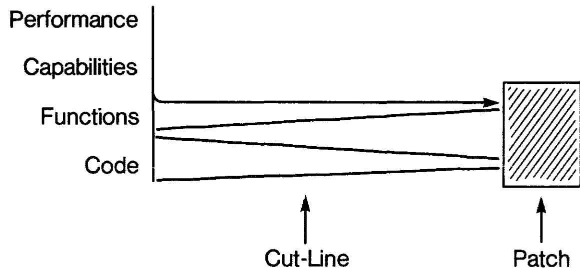


Figure 9—Specify the change

It is important to isolate the patch from sources of variability in the existing system because the patch is a piece of free-standing code, possibly of considerable size, and its development may be a development project of considerable scope. To give this project the best chance of succeeding, it is desirable to isolate it as far as possible from other ongoing or incidental changes of the host code. Thus the correct choice of the cut-line is what enables the specifications of the patch to be frozen.

FINAL PHASES OF THE CHANGE

In fact, *develop the patch* is the fourth stage of making a change (Figure 10). This is where the classic life cycle, i.e. the development path, is an appropriate model. The project has a frozen goal: code to implement the new requested function. It also has frozen specifications, namely, to fit with the chosen cut-line. The design, code, and unit test of the patch therefore proceeds according to the development path model.

The fifth stage of making a change is to *test* it. This is done by inserting the patch at the cut-line and testing the new system in parallel with the old. The main tests are these:

1. Test for the enhanced functions of the patch ("Do we have what was requested?")
2. Test for degraded functions at the cut-line ("Have we lost what we had before?")
3. Regression test

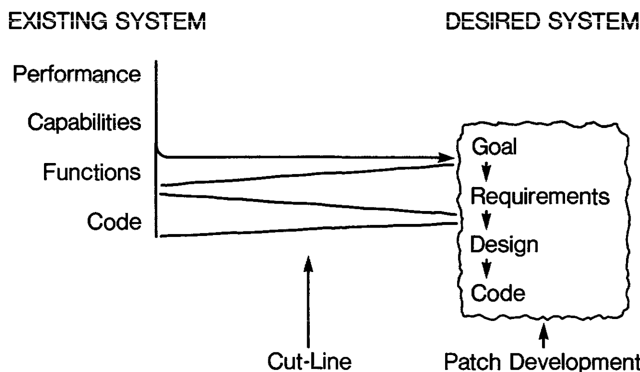


Figure 10—Develop the patch

The sixth and final stage of making a change is to *install* it—insert the tested patch at the tested cut-line and remove the old system and the entire scaffold of the change process. Then the changed system becomes the existing system, and the modification cycle begins anew (Figure 4).

The stages of the change cycle are portrayed together in Figure 11, which may be regarded as a detailed expansion of Figure 4. We propose that together they supply a more adequate model of the system life cycle.

CONCLUSIONS

The purpose of a model is to disassemble a complex process into its component parts so as to aid in the assignment of

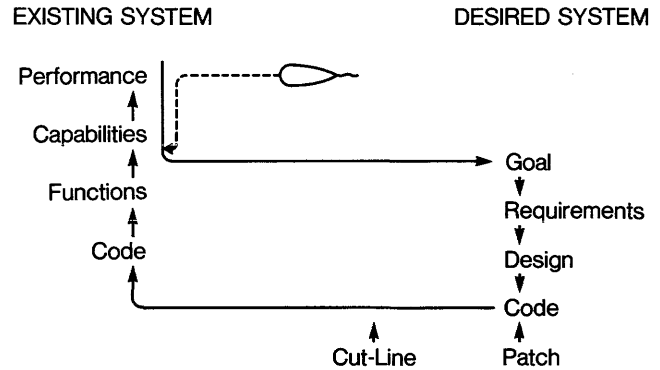


Figure 11—Details of the change cycle

resources, the training of novices, the specialization of roles, and so on. The model given in this paper has already proved fruitful in explaining on a theoretical basis some of the things that maintenance managers do—for instance, the philosophy of the big patch, or the quick and dirty fix. But, beyond that, it has implications for development and for software engineering education.

In the model the development path from goals to installation—the entire classic life cycle as presented at the start of the paper—is seen as a model of patch development, which is just a part of the change cycle, which in turn is subordinate to the modification cycle of the entire system. A similar subordination is seen in Van Horn's recent model of "evolutionary software development,"[8] but it is not reflected in any of the curricula or textbooks for software engineers. Yet such a subordination has profound implications for the staffing, estimating, or teaching of software development.

Most development projects are not, in fact, the creation of something entirely new—they are creations of a replacement for a relatively small section of a relatively larger system. They are, in fact, patch developments. But, under this model, patch development entails also the analysis of the larger system, the transformation of a request into a change, the specification of the change via the choice of cut-line and patch, and a controlled turnover. In this context, patch development is rather a small part of the problem. In fact, the choice of the cut-line is seen as the major design challenge, since it provides the frozen goal that is the idealized prerequisite for the classic development model.

Yet patch development is the only model currently being taught to software engineers; the other aspects of system modification are only learned from on-the-job experience or apprenticeship. If the analysis in this paper is correct, it gives theoretical support to the view widely held by managers that software engineers do not come out of school well prepared for the realities of their job.

ACKNOWLEDGMENTS

I would like to thank my colleagues J. Cris Miller and Michael Lyons, and our students in the course Software Maintenance: Tools, Techniques, and Management Strategies, for pushing and pulling me through the painful process of clarifying this model.

REFERENCES

1. Brooks, R. "Using a Behavioral Theory of Program Comprehension in Software Engineering." *IEEE Computer Society, 3rd International Conference on Software Engineering,* Atlanta, 1978, pp. 196–201. Silver Spring, MD: IEEE Computer Society, 1978.
2. Glass, R. L., and R. A. Noiseux, *Software Maintenance Guidebook.* Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
3. Kerola, P., and P. Freeman. "A Comparison of Lifecycle Models." *IEEE Computer Society, 5th International Conference on Software Engineering,* San Diego, 1981, pp. 90–99. Silver Spring, MD: IEEE Computer Society, 1981.
4. Lientz, B. P., and E. B. Swanson. *Software Maintenance Management—A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Reading, Massachusetts: Addison-Wesley, 1980.
5. Parnas, D. L. "On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM,* 15 (1972), pp. 1053–1058.
6. Parnas, D. L. "On the Design and Development of Program Families." *IEEE Transactions in Software Engineering,* SE-2, (1976), pp. 1–9.
7. U.S. General Accounting Office. *Federal Agencies' Maintenance of Computer Programs: Expensive and Undermanaged.* U.S. General Accounting Office, AFMD-81-25, February 26, 1981.
8. Van Horn, E.C. "Software Must Evolve." In H. Freeman and P. M. Lewis, II (eds), *Software Engineering.* New York: Academic Press, 1980.

LANGUAGE AND DATABASE PROCESSING

Data model processing

by MATTHEW B. KOLL, W. TERRY HARDGRAVE, and SANDRA B. SALAZAR
*National Bureau of Standards**
Washington, D.C.

ABSTRACT

The Data Model Processor (DMP) is an interactive tool for defining and evaluating data models. It is based on Positional Set Notation, a formalism for uniform representation of data modeling objects. The DMP allows the user to enter a set-theoretic description of a data model's structures and a definition of the model's primitive operations based on positional set operations. Based on the data model definition, the DMP then emulates a database management system (DBMS) implementing that data model. It allows the user to play various roles associated with a DBMS, such as database definer and end user.

This paper gives an overview of the DMP and discusses its foundations, namely, Positional Set Notation and a Positional Set Processor. It traces an example showing how the DMP has been used to model the relational data model. (Hierarchical and network models have also been implemented on the DMP.) Future applications of the DMP are considered.

*This paper is a contribution of the National Bureau of Standards and is not subject to copyright.

INTRODUCTION

The study of "data models" is an important aspect of database management technology. A data model is defined here as a collection of data structures plus a collection of primitive operations used for database management. Each database management system (DBMS) may be viewed as an implementation of some underlying data model. While three data models are most widely discussed, many others have been and continue to be proposed.[1]

More rigorous definition of and comparisons among data models are needed. Better selection and use of DBMS's could result from improved understanding of the various data models' strengths and weaknesses, and from detailed specification of their structures and operations. A simple, general vehicle for formal analysis of data models could be a valuable tool. Such a tool may aid in database conversion and translation as well.

The Institute for Computer Science and Technology (ICST), within the National Bureau of Standards, is charged with establishing federal standards for database management systems. To support this effort, ICST is interested in developing a more structured approach to the analysis of data models. The Abstract Database Models project at ICST has developed and implemented the Data Model Processor (DMP), a software package for formal specification and analysis of data models and their implementations.

The first prototype DMP was recently completed and has been used to study the behavior of a relational, a tree-structured, and a network-structured data model. A significant feature of the DMP is that it not only provides for a common definition language for various data models, but that it allows each data model definition to be implemented by emulating a DBMS embodying that data model (as shown in Figure 1).

This paper describes the DMP and how it may be used. First we give an overview of the DMP. Then we discuss its foundations, namely, Positional Set Notation[2] and the Positional Set Processor.[3] Positional Set Notation (PSN) is a set-theoretic notation that enables uniform representation of the data structures for various data models. The Positional Set processor (PSP) is a sophisticated tool for manipulating (e.g., storing, retrieving, combining) positional sets (p-sets). Next, we show how the DMP can be used to define and emulate DMBS's implementing the relational data model. Finally, we discuss planned and potential applications of the DMP.

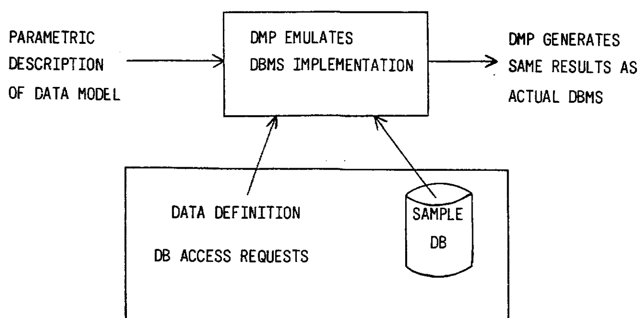


Figure 1—Overview of DMP

THE DATA MODEL PROCESSOR

Overview

The Data Model Processor (DMP) is an interactive tool for defining, testing and evaluating data models. It has been implemented in the C programming language under UNIX and runs on either a PDP-11/45 or LSI-11/23. The DMP accepts formal definitions of the structures and operations of a data model from the user. It also allows the user to define and manipulate various features of an implementation of that data model (i.e., a DBMS). As described in Figure 1, after the DMP has been given the specifications for defining and implementing a DBMS for some data model, it then emulates that DBMS.

The DMP recognizes the following different human roles involved in the life cycle of a DBMS:

1. Data Model Definer (DMD)
2. DBMS Implementer (DI)
3. Database Definer (DBD)
4. Access Definer (AD)
5. Database Populator (DBP)
6. Query/Access Language Definer (QLD)
7. Data Transformation Definer (DTD)
8. Data Manipulator (DM)

The DMP first presents the user with the master menu of roles. The user may play any role, provided that the prerequisite roles have been fulfilled. The dependencies between roles are shown in Figure 2. Dotted lines indicate that the lower role may, but does not always, depend on the higher role.

User Roles in the DMP

The Data Model Definer (DMD) is the most important role in the DMP. The activities of the other roles are structured by

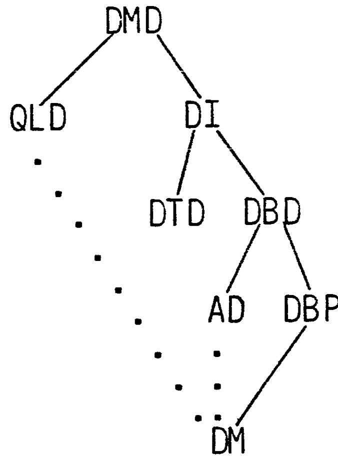


Figure 2—Dependencies among roles

information provided by the DMD. For example, the database definition and population phases consist entirely of filling in structures previously outlined by the DMD.

The DMD role is divided into the following four sections:

1. Declaring the basic CONCEPTS of the data model
2. Outlining the P-SETS used to represent the concepts
3. Identifying the SOURCES of values to populate the sets
4. Defining the primitive OPERATIONS allowed on the p-sets

In order to define a data model, the DMD must use PSN to define the structures (i.e., p-sets) that will be manipulated by the data model. The DMD outlines p-sets using the TEMPLATE command, discussed below (see Figure 3). These p-sets represent the basic concepts of the model (e.g., relations, trees). These structures are usually partitioned into two classes: data definition structures and occurrence structures. However, the DMD has substantial flexibility in these definitions. Note that the DMD is the only role requiring knowledge of PSN.

In addition to the stored structures, the DMD must define the primitive operations to manipulate these structures. These operations become available to the Data Manipulator (DM), Query Language Definer (QLD) and Access Definer (AD). Currently, operations are written in the C programming language. As seen in the example below, the operations consist mainly of calls to PSP operations.

The DBMS Implementor (DI) completes those aspects of the data model that were not completely specified by the DMD. The term "implementation" has a substantially different meaning in the DMP context than in its traditional context. Implementation, in its traditional sense, is automatic because structures are completely specified in PSN, and most or all operations are completely specified by the DMD. Except for defining the elementary sets (e.g., integers, character-strings), the DI is allowed to define only the sets and procedures which have been explicitly designated to the DI by the DMD. Such designated sets and procedures would

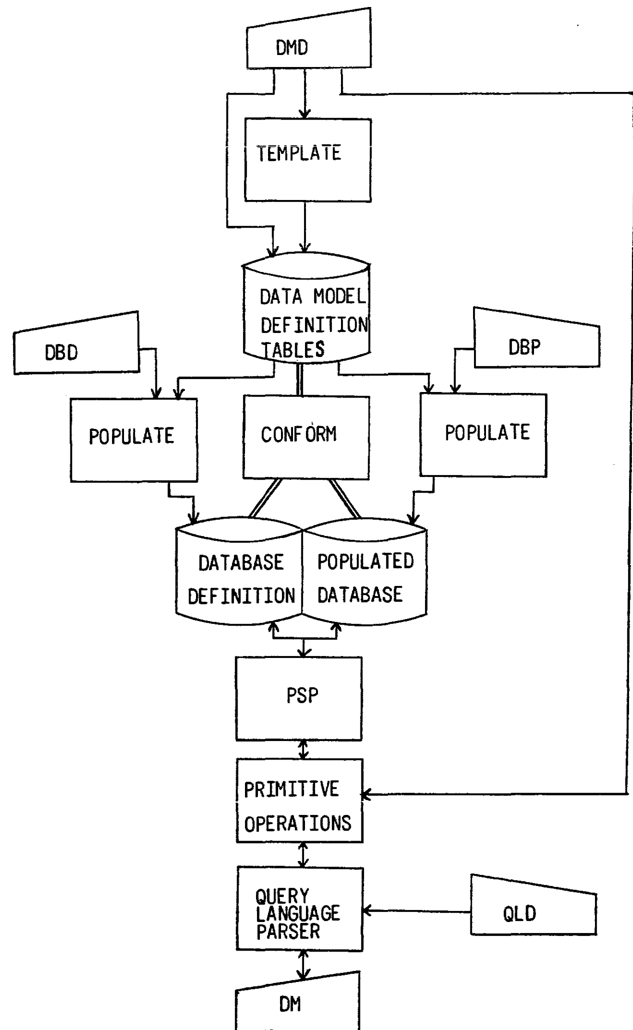


Figure 3—Architecture of DMP

be those needed for implementation but not considered part of the data model definition (e.g., valid names).

The Database Definer (DBD) enters the data definition for some databases. For each concept designated in the SOURCE section to be populated by the DBD, the DMP prompts the DBD to provide values to populate the p-set(s) representing that concept. The DMD provides the "outline" for the p-set with the TEMPLATE statement; the DBD provides the values to fill in the p-set.

The Access Definer (AD) proceeds in the same manner as the DBD. The AD populates p-sets which were outlined in the P-SETS section and designated (in the SOURCE section) for population by the AD. He assigns users to user classes, and defines a perspective (e.g., view, subschema) of the database for each class. Access control mechanisms and granularity vary with the data model and implementation.

The Database Populator (DBP) proceeds in the same manner as the DBD and AD. Traversing the appropriate template, the DMP prompts the DBP to populate each p-set designated for DBP population in the SOURCE section. These filled-in p-sets (usually the actual database) are stored

as files and are manipulable by the DMP primitive operations (through the PSP).

The Query/Access Language Definer (QLD) enters tables to define the syntax and semantics of a query or access language. These tables are inputs to the Query Processor (QP), an augmented macro processor.[4] QP accepts the specification of the syntax of the language in a tabular form that is reminiscent of SNOBOL-like programming languages. This technique is different in one simple but significant way: a mechanism is provided to pass information from the syntactic grammar to the primitive operations specified in semantic tables. This allows the study of a single syntax that has multiple semantic interpretations.

The Data Transformation Definer (DTD) is the only role that has not yet been implemented. We expect that role to be similar to the QLD. The DTD will enter tables and/or programs based on PSP operations in order to define mappings between different implementations or data models.

The Data Manipulator (DM) may use the primitive operations or a language defined by a QLD to manipulate the data in a populated database. The nature of the session with the DM depends to a large extent on the language of the data model that has been defined. The DMP asks the DM to enter his user-id, the model, implementation and database, and the language he will use so it can link to the appropriate parser. Commands issued by the DM are then passed to that parser (probably QP), which then executes the designated primitive operation(s), which, in turn, execute PSP commands, returning the response (edited if necessary) back to the DM.

Foundation

The DMP uses PSN for representing the various data models' data structures and the PSP for manipulating those structures. We provide here a brief description of PSN; more detail is available elsewhere.[2] The essence of PSN is the recursive definition of the p-set:

$$s = [xi@pi\ldots]$$

where the xi are either atoms (i.e., numbers or character-strings), or p-sets;
and the pi (Position IDentifiers—PIDs) are either atoms, or # (the null PID).
[] = the null p-set.

The *xi* are the elements of the p-set; the *pi* are the positions of their membership. The *pi* occurring in a p-set need not be unique. A pair, xi@pi, is called a duplex. The duplexes within a p-set are unordered; a p-set may be thought of as a set of ordered pairs.

There is a mapping from p-sets, *s*, to mathematical sets, s′, such that:

$$s = [xi@pi\ldots] \Rightarrow s' = \{\langle x,p\rangle \ldots\}.$$

That is, for each p-set there exists a corresponding set of ordered pairs.

P-sets are used to model data modeling objects. The three-level relationship among data modeling objects, p-sets, and mathematical sets is shown in Table I.

TABLE I—Representing objects in PSN

Data Modeling	PSN	Mathematics
{Jones, 30}	[Jones@#,30@#]	{<Jones,#>, <30,#>}
<Jones, 30>	[Jones@1,30@2]	{<Jones,1>, <30,2>}
Name Age Jones 30	[Jones@Name,30@Age]	{<Jones,Name>, <30,Age>}

The PSP is an access mechanism used to manipulate these objects. The PSP is actually a collection of about 40 operators for manipulating p-sets, each of which may be called independently at the UNIX shell level or as a subroutine. While the PSP provides many of the features of a DBMS, it is NOT a DBMS, lacking such important features as a data definition facility. Unlike most access methods, however, it has a very powerful query facility. Also, while most access methods exist for performance improvement and convenience, the PSP exists to allow precise specification and manipulation of mathematical objects.

PSP operations can be broken into four functional groups. The *Classical Set Operations* include union, intersection, cardinality, etc. When applied to positional sets, they are analogous to the traditional set operations.

The *Positional Set Operations* provide the user with the following basic database functions: retrieving, updating, adding, and deleting. Some of these operations resemble those available in relational query languages. In particular, there is a RANGE command for linking range variables with p-sets and a CREATE command which performs functions analogous to the SELECT, PROJECT and JOIN found in relational algebra. There are additional operations to return the (classical) set of elements or the (classical) set of PIDs for a p-set and to distribute a PID over a classical set (that is, to un-nest a nested set).

Three other operations of this group are noteworthy: (1) TEMPLATE, (2) POPULATE, and (3) CONFORM. As seen in Figure 3, they play key parts in the DMP. TEMPLATE allows the user to specify a template and a set of constraints that defines a class of p-sets. That is, templates are a kind of metalanguage for p-sets. POPULATE traverses a template and prompts the user to enter values at the appropriate PIDs. CONFORM is a predicate that compares a p-set to a template and set of constraints. It returns true if the p-set is structured according to the rules given in the template and if the values within the p-set conform to the specified constraints.

Some data models (e.g., CODASYL) require the manipulation of sequences. To take advantage of the ordering within sequences, the PSP has *Sequence Operations* that allow manipulation of sequences as a special form of positional sets. For example, special insert and delete operations renumber sequences after changing their contents.

The *Utility Operations* (e.g., copy, print) provide additional capabilities needed for use in an interactive environment.

SAMPLE APPLICATION

This section shows annotated excerpts from application of the DMP to the relational data model. The full definition and exercising of an implementation of the relational model, which has been recently completed, is too long to include here. We also omit details of data model processing for hierarchical and network models for lack of space. We hope that the segments shown will help explain how the DMP works.

The relational definition shown is not meant to represent THE definition, but rather, is designed to show one possible definition of the relational model that is generally consistent with common understanding.[5, 6, 7]

Where possible, system output is given in lower case and user input in upper case, with comments enclosed by "/*" and "*/". Wavy lines indicate that a portion of the interaction is omitted. In some cases, additional comments are inserted for clarification.

```
****************** data model processor******************
                        version 1.3

select option: DMD
enter name of model: RELATIONAL
-----------------------------------------------------------

     concept definition section
declare concepts ('$END' to terminate):
  >  REP_DEF DOMAINS WITH DOM-DEF; /*
  >  REPRESENT DOMAIN DEFINITIONS WITH
  >    THE P-SET "DOM_DEF" */
  >  REP_DEF RELATIONS WITH REL-DEF; /*
  >  REP. RELATION DEFS WITH "REL-DEF" */
  >  REP_OCC RELATIONS WITH REL-OCC; /*
  >  REP. RELATION OCCURRENCES WITH
  >    "REL-OCC" */
  >  $END

-----------------------------------------------------------

     p-set definition section
enter p-sets:
```

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

The next section defines templates and constraints for relation definitions and occurrences.

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

```
  >  /* SETTING UP RANGE VARIABLES FOR USE
  >    IN CONSTRAINTS */

  >  RG RD IS REL-DEF /* RANGE OF RD IS REL-
  >    DEF */
  >  RG RDA IS RD.A-D-PAIRS /* RDA RANGES
  >    OVER THE P-SET STORED AT THE PID
  >    "A-D-PAIRS" NESTED WITHIN THE P-SET
  >    CURRENTLY POINTED TO BY RD */
  >  RG RO IS REL-OCC;
  >  TEMPLATE REL-DEF = [[RN@NAME,ADP-
  >    TPL@A-D-PAIRS]@#...]
  >   /* "#..." INDICATES THAT THE PRECEDING
  >    ELEMENT'S STRUCTURE WILL BE
  >    REPEATED */
  >  WHERE
  >    ISA -C RN /* RN IS AN INTEGER */
  >    CD REL-DEF = CD CR WITH (RD.NAME)
  >      /* THE CARDINALITY OF REL-DEF
  >      EQUALS THE CARDINALITY OF REL-DEF
  >      PROJECTED ONTO NAME - I.E., REL
  >      NAMES ARE UNIQUE */
  >    TEMPLATE ADP-TPL = [[AT@ATTR,K@KY-
  >    PRT,D@DOM]@#...]
  >      WHERE
  >    ISA -C AT /* AT IS A CHARACTER-STRING */
  >    CD CR WITH (RD. NAME, RDA.ATTR) =
  >    CD CR WITH (RD.NAME, RDA.ALL)
  >      /* ATTRIBUTE NAMES ARE UNIQUE
  >      WITHIN A RELATION */
  >    ISIN K '{YES,NO}',
  >    ISIN D DOM-NAM; /* DOM-NAM IS A SET OF
  >      DOMAIN-NAMES SPECIFIED ELSE-
  >      WHERE */

  >  TEMPLATE REL-OCC = [[RN@NAME,RR-
  >    TPL@RELATION]@#...]
  >    WHERE
  >      ISA -C RN,
  >      CD REL-OCC = CD CR WITH (RO.NAME),
  >      TEMPLATE RR-TPL = [[V@P(J)...]@#...]
  >        WHERE
  >          ISIN P(J) ATTF RN, /* ATTF RETURNS
  >          THE ATTRS LISTED IN REL-DEF FOR
  >          A GIVEN RELATION */
  >    INCLUDE V (DOMF N P(J)) /* EACH VALUE
  >      IS IN THE DOMAIN PAIRED WITH ITS
  >      ATTR *
```

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

The next section shows the definition of one primitive operation.

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

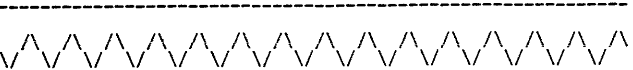
```
  primitive operations definition section
enter operations:
  >  GLOBAL PROCEDURE REMOVE_R
  >  /* REMOVE_R
  >  ** REMOVES THE RELATION REL FROM
  >  ** REL-OCC AND REL-DEF
  >  **    PARAMETERS:
  >  **              REL-   >RELATION NAME
  >  */
  >  # include "primops.h"
  >  REMOVE_R(REL)
  >  char *REL;
  >  {
```

```
>     char buff[512];
>     expsp("RL X1");
>     expsp("RG X1 IS REL-OCC");
>     expsp("RL X2");
>     expsp("RG X2 IS REL-DEF");
>     expsp(stringf(buff,
>     "CR Z WITH "(X1.ALL)" WHERE "(X1.Name
>     ~ = '%s')" " ,REL));
>     expsp( "CP Z INTO REL-OCC");
>     expsp(stringf(buff,
>     "CR Y WITH "(X2.ALL)" WHERE "(X2.Name
>     ~ = '%s')" " ,REL));
>     expsp("CP Y INTO REL-DEF");
>     expsp("DE Z");
>     expsp( "DE Y");
>  }
>  $END
```

The next sections show the population of relation definitions and occurrences. The p-sets produced are shown in tabular form in Figures 4 and 5.

```
select option: DBD
enter name of model: RELATIONAL
enter implementation: ICST
enter database name: TOY
enter RELATIONS:
populating "RELATIONAL/ICST/TOY/REL-DEF"
  [$file, $end, or ⟨cr⟩]:
  # [$e, $f, or ⟨cr⟩]:
    NAME: EMP       /* "NAME" IS SYSTEM PROMPT -
                       "EMP" IS USER RESPONSE */
A-D-PAIRS [$e, $f or ⟨cr⟩]:
      #[$e, $f or ⟨cr⟩]:
        ATTR: NAME
        KY-PRT: YES
        DOM: NAME-DOMAIN
      # [$e, $f, or ⟨cr⟩]:
        ATTR: SALARY
        KY-PRT: NO
        DOM: DOLLARS
      # [$e, $f or ⟨cr⟩]: $E
  # [$e, $f, or ⟨cr⟩]:
    NAME: SALES
    A-D-PAIRS [$e, $f or ⟨cr⟩]:
      # [$e, $f or ⟨cr⟩]:
        ATTR: DEPARTMENT
        KY-PRT: YES
        DOM: DEPT-DOMAIN
      #[$e, $f, or ⟨cr⟩]:
        ATTR: ITEM
        KY-PRT: YES
        DOM: ITEM-DOMAIN
      # ]$e, $f or ⟨cr⟩]: $E
```

```
select option: DBP
enter name of model: RELATIONAL
enter implementation: ICST
enter database name: TOY
enter RELATIONS:
populating "RELATIONAL/ICST/TOY/REL-OCC"
  [$file, $end, or ⟨cr⟩]:
  # [$e, $f, or ⟨cr⟩]:
    NAME: EMP
    RELATION [$e, $f, or ⟨cr⟩]:
      # [$e, $f, or ⟨cr⟩]:
        PID P(1): NAME
        NAME: MORGAN
        PID P(2): SALARY
        SALARY: 24000
        PID P(3): $E
      # [$e, $f, or ⟨cr⟩]:
        PID P(1): NAME
        NAME: LEWIS
        PID P(2): SALARY
        SALARY: 26000
        PID P(3): $E
      # [$e, $f, or ⟨cr⟩]: $E
  # [$e, $f, or ⟨cr⟩]: $E
    end of database population
```



At this point, the DM can manipulate the database directly via the primitive operations.

```
select option: DM
enter user-id: MATT
enter model: RELATIONAL
enter implementation: ICST
enter database: TOY
enter name of query language
  (primops if using primitive operations): PRIMOPS
enter command ($END to terminate):
/* PRINT THE NAME AND SALARY FOR ALL EMPLOYEES MAKING MORE THAN 25000; USE THE
FORMAT "EMP_FMT" (DEFINED ELSEWHERE) */
c> PRINT_R EMP E E.ALL "E.SALARY>25000"
EMP_FMT;
```

EMP	NAME	SALARY
	LEWIS	26000

```
c> $END;
  end of manipulation
***********************************************************
select option: E
  end of data model processor session
```

REL-DEF	NAME	A-D-PAIRS		
		ATTR	KY-PRT	DOM
	EMP	NAME	YES	NAME-DOMAIN
		SALARY	NO	DOLLARS
	SALES	DEPARTMENT	YES	DEPT-DOMAIN
		ITEM	YES	ITEM-DOMAIN

Figure 4—Populated REL-DEF

PLANS

Until very recently, the chief concern of this research had been the construction of the DMP. Now that it is operational, we can turn our attention to applications and experiments. Work will continue on the DMP to incorporate the DTD role, improve the DMP's performance (the current version is quite inefficient), and modify it as necessary for future applications and experiments.

The three data models that have already been implemented on the DMP served as targets during the DMP's development. The relational model used was a synthesis of the features of the major academic and commercial relational DBMS's.[7] The network model definition is fairly consistent with the specifications of ANSC/X3H2.[8] The tree-structured model used is TDMS,[9] a forerunner of System 2000, widely used in the federal government. Being able to model the structures and operations of these data models is a good indication of the DMP's power and generality as a data model modeling tool (or meta-modeler). It may be insightful to attempt to model some other existing and proposed data models.

REL-OCC	NAME	RELATION	
		NAME	SALARY
	EMP	MORGAN	24000
		LEWIS	26000

Figure 5—REL-OCC populated with the EMP relation

Query processing is an area in which we have done some experimentation[4] and plan to do more. The completeness of the set of primitive operations defined for a data model can only really be tested by trying to map query languages onto the operations. Future experiments may include mapping several relational query languages (e.g., SQL, QUEL, QBE) onto one set of primitive operations, implementing different semantic approaches for TDMS,[9] and mapping a relational query language onto CODASYL primitive operations.

Other work on mapping may be conducted within the scope of the DTD role. One idea is to use the DMP to develop database conversion strategies. While large conversions would not actually use the DMP, the DMP could provide a framework for design, formal definition, and preliminary testing of transformations.

Another use of the DMP will be implementing and evaluating data model specifications. Such work would be along the lines of that mentioned above using the ANSC/X3H2 specification of a network-structured data model. In attempting to formally define and implement an abstract specification, anomalies can often be discovered at an early stage.

The DMP may be of some practical value in the process of selecting a DBMS. A potential buyer may be able to use the DMP to help define his requirements and to evaluate prospective systems.

Finally, the DMP has pedagogical value. It allows one to explore a wide variety of data models and query languages in an experimental environment. A library of data model specifications available for teaching and modification on an experimental basis could be maintained.

ACKNOWLEDGMENTS

The products of the Abstract Database Models project, including this paper, represent a group effort. We would like to thank several individuals for their contributions: Steve Norman, Jerry Herstein, Tony Marriott, and Chuck Haas for implementing the DMP; Gary Sockut, Sabrina Saunders and T. C. Ting for their work in applying (and debugging) the DMP; and Joe Naputi, George Skillman, Ed Beller, and Tammy Kirkendall for giving the PSP the strength to carry on.

REFERENCES

1. Kerschberg, L., A. Klug, and D. C. Tsichritzis. "A Taxonomy of Data Models." In P. C. Lockemann and E. J. Neuhold (eds.), *Systems for Large Data Bases.* Amsterdam: North Holland, 1977.
2. Hardgrave, W. T. "Positional Set Notation." To appear in *Advances in Database Management, Vol. 2.* New York: Heyden and Sons, 1982.
3. Hardgrave, W. T. and S. B. Salazar. "The Positional Set Processor: A Tool for Data Modeling." National Bureau of Standards NBSIR 81-2302, Washington, DC, 1981.
4. Hardgrave, W. T., M. B. Koll and S. B. Salazar. "Query Translation and Processing." National Bureau of Standards NBSIR (in progress), Washington, DC, 1982.
5. Date, C. J. *An Introduction to Database Systems.* Reading, MA: Addison-Wesley, 1977.
6. Ullman, J. D. *Principles of Database Systems.* Potomac, MD: Computer Science Press, 1980.
7. Final Report of ANSI/X3/SPARC DBS-SG Relational Task Group. CBEMA/X3/Secretariat, Washington, DC, 1981.
8. Draft Proposed American National Standard for a Data Definition Language for Network Structured Databases. CBEMA/X3/Secretariat (ANSC/X3H2), Washington, DC, 1981.
9. Hardgrave, W. T. "Ambiguity on Processing Boolean Queries on TDMS Tree Structures: A Study of Four Different Philosophies." *IEEE Transactions on Software Engineering,* Se-6 (1980), 4, pp. 357–372.

Automatic database system conversion: schema revision, data translation, and source-to-source program transformation

by BEN SHNEIDERMAN
University of Maryland
College Park, MD
and
GLENN THOMAS
Kent State University
Kent, OH

ABSTRACT

Changing data requirements present database administrators with a difficult problem: the revision of the schema, the translation of the stored database, and the conversion of the numerous application programs. This paper describes an automatic database system conversion facility which provides one approach for coping with this problem. The Pure Definition Language and the Pure Manipulation Language have been designed to facilitate the conversions specified in the Pure Transformation Language. Two conversions and their effect on retrievals are demonstrated.

INTRODUCTION

Contemporary database management systems isolate the users from changing physical implementation strategies, but offer little assistance when logical structures must be modified. Several research projects have been directed at automating part or all of the database system conversion process. A complete strategy for coping with requirement changes would have to aid in the revision of the schema, translation of the stored database, and conversion of the application programs.

Data translation research at the University of Michigan has begun to include work on database conversion (Navathe and Fry, 1976,[19] Swarthwout, Deppe and Fry, 1977[28]), by classifying possible schema transformations for a network structured database and by specifying architectures for a conversion system.

The IBM Research group at San Jose, which developed the EXPRESS system (Shu et al., 1977[23]) for data translation, recognized that this powerful system was a natural basis for developing program conversion aids. Housel's paper (1977)[16] showed how CONVERT operations could be used as a query language as well as for describing schema transformations. He demonstrated a set of rules which enabled schema transformations described in CONVERT to be applied to CONVERT queries.

At the University of Florida, CONVERT transformations were applied to relational schemas and SEQUEL queries (Su and Liu, 1977,[26] Su and Reynolds, 1977,[27] and Su, 1976[24]). Dale and Dale (1976, 1977[13,14]) at the University of Texas studied program preserving transformations for the tree structured data model. Gerritsen and Morgan (1976)[15] dealt with a class of network schema transformations by dynamically translating program statements to match the revised schema. Navathe (1980)[18] examined transformations on schema diagrams which were related to the entity-relationship model and Sakai (1980)[20] suggested some transformations during logical design in the relational model. Shneiderman (1978)[22] provided a framework for research in network schema transformations and Taylor et al. (1979)[29] identified problem areas and offered several directions for research. Jacobs (1980)[17] described automatic conversion in the context of his database logic which provides a formal mathematical foundation for database systems.

AUTOMATIC CONVERSION IN THE PURE DATABASE SYSTEM

The Pure Database System was designed to facilitate schema changes which call for database translation and application program conversion. For example, changing a two-level schema structure, such as division records owning employee records, to a three-level structure, where division records own department records which in turn own employee records, generally requires special purpose translation programs to restructure the database and hand-written revisions to modify application programs. Using Pure Transformation Language (PTL) operators, database administrators can specify transformations which automatically generate a new schema, stored database, and application programs. The execution of the target application programs operating on the target database should produce output identical to that produced by the source application programs operating on the source database.

Our Pure Definition Language (PDL) and Pure Manipulation Language (PML) blended elegant high-level relational ideas with lower-level network concepts to produce a data model conducive to automatic transformation. Our goals were to ensure input/output equivalence where possible, minimize host language interactions, provide integrity assurance for the transformations, offer useful and effective definition and manipulation languages, and construct a convenient set of transformations. Although more efficient transformations are feasible, we felt that generality, modularity, simplicity, provability, and integrity assurance were more important criteria. The effectiveness of the PDL and PML and the utility of the PTL must be verified through field testing or controlled experiments with manual alternatives.

Refinements, extensions, and alternatives are easy to generate, but we felt the need to limit our focus and demonstrate a complete workable system. The PDL and the PML have been implemented using the XPL compiler-compiler to generate UNIVAC DMS-1100 code which is then executed through normal procedures. The PTL processor is more complex since it requires the preparation of a new schema, the generation of programs to restructure the database, and the revision of possibly hundreds of application programs embedded in host language code. Furthermore, before a transformation is carried out, the stored database must be examined to ensure that integrity constraints are satisfied and the application programs must be parsed to verify that the transformation is possible.

Although great care and effort was devoted to constructing a set of transformations at an appropriate level, we recognize alternative approaches. Higher level transformations (Su and Lam, 1979[25]) would capture more "semantic" constructs, but a greater number of transformations might be needed to accommodate database administrator (DBA) needs. Lower level transformations might be easier to implement and prove correct, but would be complicated to use. Feedback from users and experience seems essential to help choose the most convenient approach.

PURE DEFINITIONS AND MANIPULATIONS

Like the CODASYL DBTG approach, the Pure schema has a collection of record types and set types. There is a singular record type, SYSTEM, which is the starting point for all searches. Each record type may be the owner and member of several set types but the schema graph must be acyclic. Within a set instance the record instances are in ascending order by the set keys. The simple schema shown in Figure 1a might be defined by the following Pure Definition statements:

```
SCHEMA NAME IS students.
RECORD SECTION.
    RECORD NAME IS stu.
    FIELDS ARE.
        sno         PIC 9(6).
        sname       PIC X(25).
    END RECORD.

    RECORD NAME IS crs.
    FIELDS ARE.
        cno         PIC 9(4).
        title       PIC X(60).
        grade       PIC X(3).
    END RECORD.

END RECORD SECTION.

SET SECTION.
    SET NAME IS sys-stu.
    OWNER ISE SYSTEM.
    MEMBER IS stu.
    SET KEY IS (sno).
    END SET.

    SET NAME IS stu-crs.
    OWNER IS stu.
    MEMBER IS crs.
    SET KEY IS (cno).
    END SET.

END SET SECTION.
END SCHEMA.
```

In this schema student records (stu) are in ascending order by student number (sno) and contain the student name (sname). Each student record owns a set of course records (crs) which are kept in ascending order by course number (cno) and contain a course title and grade.

The current design for the Pure Manipulation Language assumes embedding in a host language such as COBOL. The FIND statement specifies a search through the database and the creation of a train (an ordered collection of record identifiers, each of which uniquely specifies a database record) satisfying an access path expression. For example, to find the course records in which a student named 'JOE' received a grade of 'A' we might write:

```
FIND (crs: SYSTEM, sys-stu, stu(sname = 'JOE'), stu-crs,
          crs (grade = 'A')).
```

The target record type, crs, must be somewhere along the path expression which follows the colon. The path expression starts at the SYSTEM record and follows sets and records through the database. Records may have boolean expressions to qualify field names within a record and the path expression may traverse sets in forward and reverse order.

The GET statement retrieves a single record of a train specified by its numeric position in the train, and places the record in a buffer area associated with the record type. By embedding the GET statement in a loop all the records in a train can be retrieved. The STORE statement inserts a record in the database and ensures that the record will properly participate in all sets in which it is an owner or member. The DELETE statement can be used to delete a single record or a train of records from the database. Deletion can only be made if the database structure is preserved and if all records would be reachable after the deletion. Three forms of the MODIFY statement have been included: replacement of non-key field values in a record, alternation of key fields which effect set order only, and alteration of key fields which effect set membership. Improperly or incompletely specified modifications are not carried out.

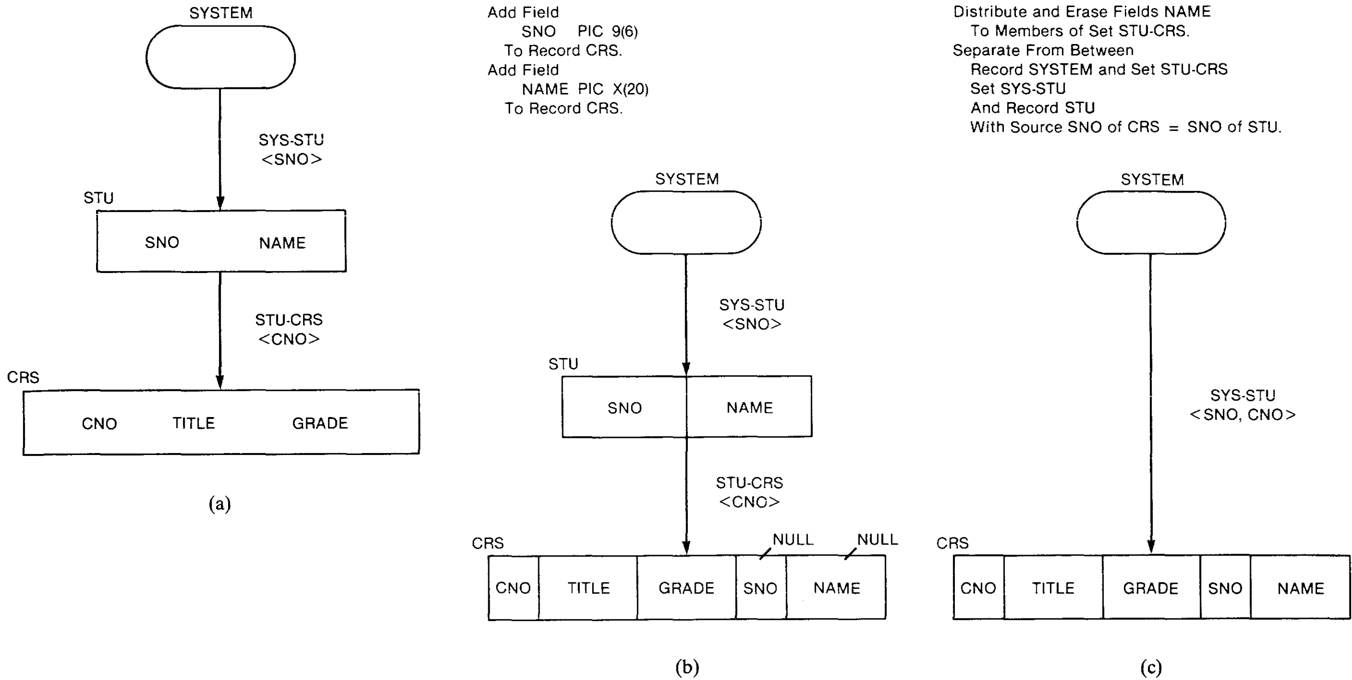
In summary, the Pure System blends the appeal of schema traversal using path expressions with the high level relational operations on collections of records. The network concepts of set ordering and explicit linkage have been combined with the relational notions of keys and tuple uniqueness.

PURE TRANSFORMATIONS

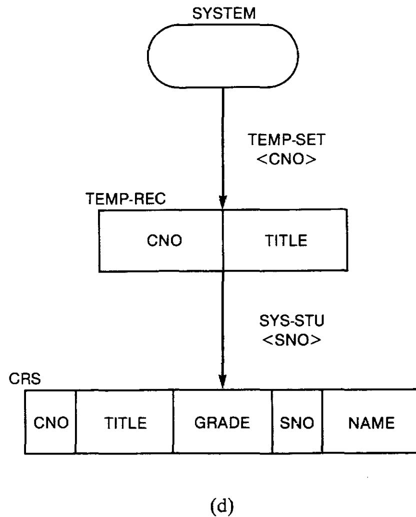
The 18 Pure transformations presented in Table I permit conversion of a two-level schema to a three-level schema, factoring of common fields from member to owner records, distribution of fields from owner to member records, manipulation of set key fields, introduction (and elimination) of sets, records and fields, and name changes. We first offer a general three dimensional categorization for transformations before informally presenting the Pure Transformation Language.

The schema of Figure 1a allows queries of the form: "What grade did student X receive in course Y?" A possible transformation would be to remove the field grade from the record type crs. This transformation is not information preserving because the new (target) schema does not contain all the information derivable from the old (source) schema. More generally a transformation is *information preserving* if all the information derivable from the source schema is derivable from the target schema.

The second categorization dimension of *data dependence* may be illustrated by a user requirements change. Assume, DEPT records may own EMPLOYEE records which contain a field MGR identifying the employee's manager. A corporate policy change may require that all employees within a department be managed by the same individual. In this case, it is reasonable to move the MGR field to the DEPT record type. The FACTOR and ERASE transformation copies a field value from the members of a set to their common owner. Before the transformation may be allowed, each occurrence of the set type linking DEPT and EMPLOYEE occurrences must be examined to determine whether or not the source

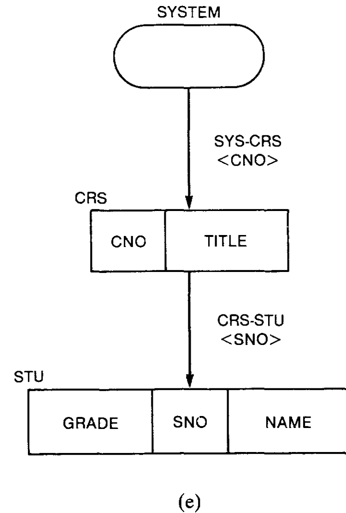



```
Permute Key of Set SYS-STU
   From (SNO, CNO) to (CNO, SNO).
Introduce Between Record SYSTEM and Set SYS-STU
Record
   Record NAME is TEMP-REC.
   Fields are.
      CNO  PIC X(4).
      TITLE PIC X(30).
   End Record.
And Set
   Set NAME is TEMP-SET.
   Owner Record is SYSTEM.
   Member Record is TEMP-REC.
   Set Key is (CNO).
   End Set.
With Source CNO of TEMP-REC = CNO of CRS.
Factor and Erase Fields TITLE
   From Members of Set SYS-STU.
```



(d)

```
Remove Fields CNO, TITLE From Record CRS.
Change NAME of Record
   From CRS to STU.
Change NAME of Set
   From SYS-STU to CRS-STU
Change Name of Record
   From TEMP-REC to CRS.
Change NAME of Set
   From TEMP-SET to SYS-CRS.
```



(e)

Figure 1—Inversion of the student and course relationship

TABLE I—Categorization of pure transformations

	Information preserving		Not information preserving	
	Data independent	Data dependent	Data independent	Data dependent
Program independent	Change name Add field Permute Distribute Fields Distribute and erase fields Introduce set record Introduce between	Distribute set key Distribute and erase set key fields Introduce where		Detach Factor fields Factor and erase fields
Program dependent	Append		Remove fields Separate set Separate set record Separate from between	

stored database satisfies this corporate policy. Should any occurrence of the set type linking DEPT and EMPLOYEE have two or more member record occurrences with different values for the field MGR, then the transformation fails. Transformations are *data dependent* if the source stored database must be examined to determine whether or not current data values satisfy changing user requirements. Otherwise, a transformation is *data independent.*

The final categorization dimension is *program independence.* Should the field sname be removed from the Figure 1a record type stu, any source program referencing this field will have to be examined to determine whether it can be modified to run under the target schema or dropped from the set of programs accessing the stored database. While the conversion system can isolate such program dependencies, the database administrator is responsible for deciding on the correct action to be taken. Logical schema change. A transformation is *program dependent* if the source programs must be examined to determine whether or not they can be modified to run under the target database system. Otherwise a transformation is *program independent.*

CHANGE NAME, ADD FIELD, and REMOVE FIELDS allow the database administrator to change the name of any source set, record or field, add a new field to an existing record type, or remove a field from the definition of a record type. Of these, only REMOVE FIELDS requires program examination. The sequence:

ADD FIELD f TO RECORD r.
REMOVE FIELDS f FROM RECORD r.

yields a target database system that is identical to the source database system. Hence, REMOVE FIELDS is the inverse of ADD FIELD. However, the reverse is not true because REMOVE FIELDS destroys data values in the source database.

The transformations APPEND, PERMUTE and DETACH allow the database administrator to redefine the key of an existing set and logically reorder member record occurrences. APPEND adds a field as the least significant component of a set key. While information preserving and data independent, APPEND requires DBA interaction to modify storage paths involving the member record type. DETACH removes the least significant set key field for some set. Each occurrence of the affected set must be examined to determine whether or not the resulting set key will uniquely identify the members. If it will not, DBA interaction is required to obtain the necessary uniqueness. APPEND and DETACH are inverses for each other when allowed. As illustrated by Figures 1c and 1d, PERMUTE redefines the left-to-right order of concatenation of set key fields. In addition to being information preserving, data independent, and program independent, PERMUTE is the only Pure transformation that is its own inverse.

The six DISTRIBUTE and FACTOR transformations allow for the copying of field values from owner to member records (DISTRIBUTE) or vice versa (FACTOR). For FACTOR, this requires examining the source stored database to determine whether or not all members of each set occurrence share a common value for the field(s) being copied. When ERASE is specified, the field is set to the 'null' value after it has been copied. FACTOR and DISTRIBUTE are mutual inverses.

INTRODUCE SET RECORD adds a record type and a set type owning this record type to the schema. This transformation has no effect of the stored database or the set of programs as no instances of the new record type exist. SEPARATE SET RECORD is the inverse for INTRODUCE SET RECORD. When specified, all occurrences of the named record and set types are removed from the source database system. Because data values are lost, this is not an information preserving transformation. Thus, it has no inverse.

INTRODUCE WHERE allows the definition of a new set type between two existing record types. The WHERE clause specifies a selection criterion to associate every member record occurrence with exactly one owner record occurrence. These are then made members of a set occurrence of the new set type whose owner is selected by the WHERE clause criterion. SEPARATE SET is the inverse of INTRODUCE WHERE. Because this eliminates a path from the schema and destroys the information contained in this set type, this is not an information preserving transformation.

The final pair INTRODUCE BETWEEN and SEPARATE

FROM BETWEEN allow for transformations of the form illustrated by the schemata of Figures 1c and 1d where a new record and set type are introduced between an existing record and set type. A succession of INTRODUCE BETWEEN's may be employed to create a hierarchy within the schema while SEPARATE FROM BETWEEN may be employed to remove this hierarchy.

PURE TRANSFORMATION EXAMPLE

Figures 1 and 2 provide two examples of potential applications of an automatic database conversion system. The starting database in Figures 1a and 2a shows a collection of student records organized in ascending order by student number (sno). Each student record owns a collection of course records (crs) which are organized in ascending order by course number (cno). Figures 1b through 1e show successive transformation steps to convert the source schema into a target schema where courses own students. This conversion was called inversion by Navathe and Fry (1976).[19]

The example query shown earlier, Find the course records in which a student named 'Joe' received a grade of 'A', applies to the schema in Figure 1a:

```
FIND (crs: SYSTEM, sys-stu, stu(sname = 'JOE'), stu-crs,
          crs(grade = 'A')).
```

No program transformation is required for the schema in Figure 1b. The DISTRIBUTE AND ERASE in Figure 1c requires the introduction of a boolean path expression involving the EXISTS predicate:

```
FIND (crs: SYSTEM, sys-stu,
          stu(EXISTS (stu-crs, crs(sname = 'JOE'))),
          stu-crs, crs(grade = 'A')).
```

The SEPARATE FROM BETWEEN transformation in Figure 1c allows a more compact path expression:

```
FIND (crs: SYSTEM, sys-stu, crs (sname = 'JOE' and grade
          = 'A')).
```

The INTRODUCE BETWEEN transformation in Figure 1d forces a longer path expression:

```
FIND (crs: SYSTEM, temp-set, temp-rec, sys-stu,
          crs(sname = 'JOE' AND grade = 'A')).
```

Finally, the name changes in Figure 1e induce a simple transformation to the desired target query for the new schema:

```
FIND (crs: SYSTEM, sys-crs, crs, crs-stu,
          stu(grade = 'A' AND sname = 'JOE')).
```

Figures 2b through 2d show successive transformation steps to create a many to many relationship between student and course instances. Here again the paths expressions in FIND statements can be rewritten in an orderly way so that the same retrievals can be performed on the target database. Of course, transformations to STORE, DELETE, and MODIFY statements can present somewhat greater difficulty, but we feel that where a transformation is possible, our design supports it.

Figures 1 and 2 show the effects of the transformation on a schema diagram, but the system is designed to take the actual code for the source schema and generate a target schema, to take the stored database and translate it to match the target schema, and to take the numerous application programs and convert them to run on the target schema. Of course, if information is deleted during a transformation, some of the application programs may not operate in the same way as they did before. The database administrator must decide if the results of such a conversion are acceptable. Whether the eighteen transformations we offer are convenient and provide enough power to be useful in commercial applications remains an open question.

CURRENT RESEARCH DIRECTIONS

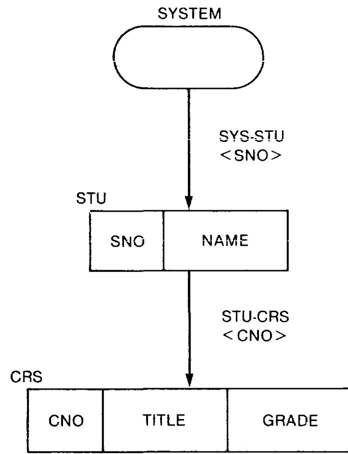
Our fundamental goal has always been to create a research system which demonstrates the feasibility of automatic database system conversion. We do not seek Pure Database System users, but rather hope that this work will inspire other designers to provide automatic database conversion facilities in their system architecture.

We are currently trying to apply the ideas in the Pure Database System to conversion in other data models and to conversion across data models. Shneiderman and Thomas (1982)[12] describe 15 transformations for the relational model of data and suggest an architecture for an automatic conversion system. Schema to sub-schema mappings can be defined with transformation operations (Thomas and Shneiderman, 1980).[4] We are also pursuing a formalization of these concepts so as to verify the correctness of a transformation, assess the range of our set of transformations, and uncover additional useful transformations.

This work is relevant to standardization efforts currently in progress because we beleive that the ease of conversion should be a consideration for all database definition and manipulation languages. Secondly, a standards planning effort would be useful to coordinate and unify the diverse proposals for transformation facilities.

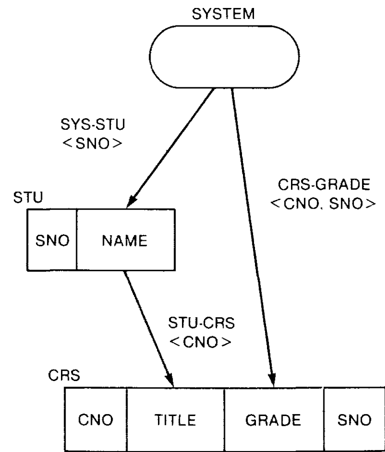
ACKNOWLEDGMENTS

Carl Fosler and Robyn Birckhead carried out the implementation of the PDL and PML. Nancy Sevitsky prepared a user's manual, Wayne Fuller assisted with the documentation, and Bonnie Zager provided administrative support. The Computer Science Center, of the University of Maryland provided some of the computer resources for this project. National Science Foundation grant MCS-77-22509 provided partial support. We appreciate the comments of our colleagues Michael L. Brodie, Barry Jacobs, Edgar H. Sibley and the members of the Database Program Conversion Task Group of the CODASYL Systems Committee, especially Stanley Su and Jim Fry.



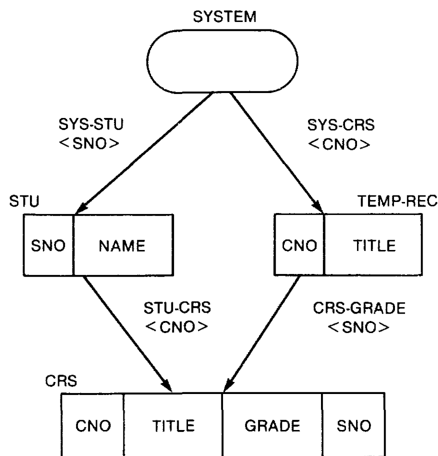
(a)

```
Add Field
  SNO PIC 9(6).
  To Record CRS.
Distribute Fields SNO
  To Members of Set STU-CRS.
Introduce Set
  Set NAME is CRS-GRADE.
  Owner Record is SYSTEM.
  Member Record is CRS.
  Set Key is (CNO, SNO).
  End Set.
  Storage Path is SYSTEM, CRS-GRADE,
    CRS (CNO = CNO-ID and SNO = SNO-ID).
```



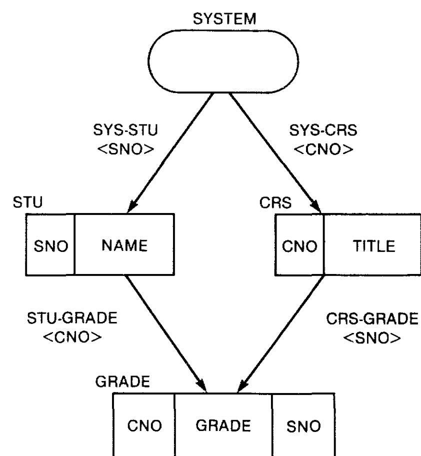
(b)

```
Introduce Between Record SYSTEM and Set
  CRS-GRADE Record
    Record NAME is TEMP-REC.
    Fields Are.
      CNO  PIC  X(4).
      TITLE  PIC  X(30).
    End Record.
And Set
      Set NAME is SYS-CRS.
      Owner Record is SYSTEM.
      Member Record is TEMP-REC.
      Set Key is (CNO).
      End Set.
With Source CNO of TEMP-REC = CNO of CRS.
```



(c)

```
Factor and Erase Fields TITLE
  From Members of Set CRS-GRADE.
Remove Fields TITLE From Record CRS.
Change NAME of Record
  From CRS to GRADE.
Change NAME of Record
  From TEMP-REC to CRS.
Change NAME of Set
  From STU-CRS to STU-GRADE.
```



(d)

Figure 2—Transformation from a one-to-many to a many-to-many relationship

PURE SYSTEM REPORTS

1. Shneiderman, B., and G. Thomas. *Automatic Database System Conversion I: Data Definition and Manipulation Facilities.* Computer Science Technical Report Series TR-82, University of Maryland, College Park, Md., 20742 (1979), 39 pages. (Submitted for publication.)
2. Thomas, G., and B. Shneiderman. *Automatic Database System Conversion II: A Transformation Language.* Computer Science Technical Report Series TR-281, University of Maryland, College Park, Md. 20742 (1979), 46 pages. (Submitted for publication.)
3. Shneiderman, B., and G. Thomas. "Path Expressions for Complex Queries and Automatic Database Program Conversion." *Proceedings of the 6th Very Large Data Bases Conference.* Montreal (1980), pp. 33–44.
4. Thomas, G., and B. Shneiderman. "Automatic Database System Conversion: A Transformation Language Approach to Sub-Schema Implementation." *Proceedings of the IEEE COMPSAC '80 Conference,* Chicago, (1980).
5. Shneiderman, B., and G. Thomas, *Pure Database System Report: A Transformation Language Approach to Automatic Schema, Stored Data and Program Conversion.* Computer Science Technical Report Series TR-880, University of Maryland, College Park, Md. 20742 (1980), 91 pages. (Submitted for publication.)

The Pure Language components defined by this reports are separately described in:

6. Shneiderman, B., and G. Thomas. *Pure Definition Language Manual.* University of Maryland, College Park, Md. 20742 (1980), 10 pages.
7. Shneiderman, B., and G. Thomas. *Pure Manipulation Language Manual.* University of Maryland, College Park, Md. 20742 (1980), 30 pages.
8. Shneiderman, B., and G. Thomas. *Pure Transformation Language Manual,* University of Maryland, College Park, Md. 20742 (1980) 50 pages.

Other Pure System reports are:

9. Fosler, C. *Pure System XPL—DMS/1100 Implementation Documentation.* Computer Science Technical Report Series TR-872, University of Maryland, College Park, Md. 20742 (1980), 38 pages.
10. Fosler, C. *Pure PDL and PML Runstream and Examples.* University of Maryland, College Park, Md. 20742 (1980), 32 pages.
11. Sevitsky, N. *Pure User's Manual.* University of Maryland, College Park, Md. 20742 (1980), 52 pages.
12. Shneiderman, B., and G. Thomas. "An Architecture for Automatic Relational Database System Conversion." *ACM Transactions on Database Systems* (June 1982.)

REFERENCES

13. Dale, A. and N. Dale. "Main Schema—External Schema Interaction in Hierarchically Organized Data Bases." *Proc. ACM SIGMOD Conference, 1977* pp. 102–110.
14. Dale, A., and N. Dale. "Schema and Occurrence Structure Transformations in Hierarchical Systems." *Proc. ACM SIGMOD conference* (1978).
15. Gerritsen, R. and H. L. Morgan. "Dynamic Restructuring of Databases with Generation Data Structures." *ACM National Conference 1976,* pp. 281–286.
16. Housel, B. "A Unified Approach to Program and Data Conversion." *Proc. 3rd Very Large Data Bases Conference,* Tokyo (1977).
17. Jacobs, B. "Applications of Database Logic to Automatic Program Conversion." Submitted for publication.
18. Navathe, S. B. "Schema Analysis for Database Restructuring." *ACM Transactions on Database Systems, 5* (1980), pp. 157–184.
19. Navathe, S. B., and J. P. Fry. "Restructuring for Large Databases: Three Levels of Abstraction. *ACM Transactions on Database Systems* 1 (1976), pp. 138–156.
20. Sakai, H. "Entity-Relationship Approach to the Conceptual Schema Design." *Proceedings of the ACM SIGMOD Conference, 1980,* pp. 1–8.
21. Shu, N., B. Housel, R. W. Taylor, S. Ghosh, and V. Lum "EXPRESS: A Data Extraction, Processing, and Restructuring System." *ACM Transactions on Database Systems* 2, (1977) pp. 134–174.
22. Shneiderman, B. "A Framework for Automatic Conversion of Network Database Programs Under Schema Transformations." *Third Jerusalem Conference on Information Technology* (J. Moneta, ed.) Amsterdam: North-Holland, 1978.
23. Shu, N. C., B. C. Housel, and V. Y. Lum. "CONVERT: A High Level Translation Definition Language for Data Conversion." *Communications of the ACM,* 18 (1975), pp. 557–567.
24. Su, S. Y. W. "Application Program Conversion Due to Database Changes." *Proc. 2nd International Conference Very Large Data Bases,* Brussels, Belgium (September 1976). Amsterdam: North-Holland, 1976, pp. 143–158.
25. Su, S. Y. W., and H. Lam. "Transformation of Data Traversals and Operation in Application Programs to Account for Semantic Changes in Databases." Department of Computer and Information Sciences, University of Florida, Gainesville, Florida, 1979.
26. Su, S. Y. W., and B. J. Liu. "A Methodology of Application Program Analysis and Conversion Based on Database Semantics." *Proceedings of the ACM SIGMOD Conference, 1977,* pp. 75–87.
27. Su., S.Y. W., and M. J. Reynolds, "Conversion of High-Level Sublanguage Queries to Account for Database Changes." *AFIPS, Proceedings of the National Computer Conference* (Vol. 47), 1978, pp. 857–875.
28. Swarthwout, D. E., M. E. Deppe, and J. P. Fry. "Operational Software for Restructuring Network Databases." *AFIPS, Proceedings of the National Computer Conference* (Vol. 46), 1977, pp. 499–508.
29. Taylor, R. W., J. P. Fry, B. Shneiderman, D. C. P., Smith, and S. Y. W. Su, "Database Program Conversion: A Framework for Research." *Proceedings of the 5th Very Large Database Conference.* Available from ACM, New York, 1979.

Fair timestamp allocation in distributed systems

by SAID K. RAHIMI
Honeywell Corporate Computer Sciences Center
Bloomington, Minnesota
and
WILLIAM R. FRANTA
University of Minnesota
Minneapolis, Minnesota

ABSTRACT

Many researchers have addressed the problem of uniquely identifying updates in a distributed database system in the literature.[1,5,6,7,11] Primitive identification schemes that generate globally unique update IDs have also been suggested. These IDs are usually used as priority among updates as well. When used as such, these schemes do not distribute priority evenly across the nodes. This paper presents a numbering scheme that generates unique update IDs and, if used as a priority scheme, is fair.

1. INTRODUCTION

Many authors have addressed the problem of global identification of updates in a distributed database system.[1,5,6,7,10,11] To solve the problem, these researchers have suggested primitive ID generation schemes that globally identify all updates. Almost all these suggestions assume an ID to be a combination of two parameters: a local physical-clock parameter to provide for local identification and node numbers to provide for global identification. Thomas' algorithm for concurrent update problem of distributed database systems[11] assumes that an update ID number is a combination of a node number and readings of a physical clock at that node at the time of update generation. (Our model of a distributed system consists of a set of cooperating nodes connected by a communication facility.) Physical clocks, kept at every node of the system, tend to require resynchronization periodically. If physical clocks are skewed with respect to one another or run at different rates, certain anomalies may occur.[11] To solve the synchronization problem, Lamport[5] has suggested a rather expensive mechanism to resynchronize drifted clocks.

Away from physical-clock problems, these schemes are able to generate globally unique identification numbers for updates. The ID numbers generated are also used as priority numbers among updates.[7, 11] A priority scheme as such does not distribute priority evenly across the nodes. The reason is the fixed node number assignment that biases the priority among updates from different nodes.

Section 2 explains the update numbering schemes and their problems. To solve some of the problems, Section 3 presents the MOD numbering scheme. This scheme is solely based on the use of logical clocks and therefore does not have the problems associated with physical clocks. If ID numbers generated by this scheme are used as priority, one can be sure that this priority scheme is fair. The MOD numbering scheme achieves fairness by dynamically changing the node numbers. The problem associated with varying node numbers and a solution to this problem are also presented.

2. UPDATE NUMBERING SCHEMES/PROBLEMS

An update ID number is generated and assigned to an update by the initiating node at the time of update generation. It is assumed that each node has a logical clock (instead of a physical clock in similar schemes). A logical clock at a node simply counts the number of updates generated at that node. This means that a logical clock at a node is incremented by 1 for every update generated at that node. Another update at a node cannot be generated before the clock at that node is incremented (it is assumed that all logical clocks are set to 0 at the system initiation time). Using the logical-clock readings (LCR) at every node, therefore, solves the problem of locally identifying the updates and orders the updates by their generation. This, on the other hand, does not provide for global identification of updates, because LCRs at different nodes may be the same.

To ensure a global identification, node numbers are used as the second part of update IDs. It is assumed that the N nodes of the system are uniquely numbered 0 to $N-1$. If NN is node number, the tuple (LCR, NN) is a unique ID throughout the system. Two IDs, $ID_i = (LCR_i, NN_i)$ and $ID_j = (LCR_j NN_j)$ are said to be different ($ID_i \neq ID_j$) if and only if $LCR_i \neq LCR_j$ or $NN_i \neq NN_j$.

It is easy to show that for any two different updates i and j with $ID_i = (LCR_i, NN_i)$ and $ID_j = (LCR_j, NN_j)$, $ID_i \neq ID_j$. To see this, suppose updates i and j are generated at the same node; i.e., $NN_i = NN_j$. According to the above discussion, LCR has to be incremented after it is read for one update, and hence $LCR_i \neq LCR_j$. If the updates are from two different nodes, then $NN_i \neq NN_j$, which implies that $ID_i \neq ID_j$.

ID numbers are used in two different ways: for identification and for priority purposes.

The uniqueness property of update IDs, generated this way, gives us confidence in using these tuples as identification of updates. When used as priority, however, this scheme raises some questions. Note that priority here is concerned with ordering conflicting updates and does not have anything to do with user-defined or external priority. For two updates i and j it is usual to say

update i is of equal priority to update j if $ID_i \doteq ID_j$,
update i is of higher priority than update j if $ID_i < ID_j$,
and
update i is of lower priority than update j if $ID_i > ID_j$.

Since there are two different elements (LCR and NN) constituting each update ID, there are two possible ways of defining relations $=$, $>$, and $<$ for two updates i and j:
First,

$ID_i = (LCR_i, NN_i)$
and
$ID_j = (LCR_j, NN_j)$

which means that

$ID_i = ID_j$ if and only if $LCR_i = LCR_j$ *and* $NN_i = NN_j$,
$ID_i > ID_j$ if and only if $(LCR_i > LCR_j)$ *or* $(LCR_i = LCR_j$ *and* $NN_i > NN_j)$
$ID_i < ID_j$ if and only if $(LCR_i < LCR_j)$ *or* $(LCR_i = LCR_j$ *and* $NN_i < NN_j)$,

as used in Rosenkrantz et al.,[9] Thomas,[11] and Traiger et al.[12]

A priority scheme is said to be fair if it distributes priority evenly among the updates from different nodes. A scheme that gives high priority to updates from one node all the time is not fair.

As far as priority is concerned, the scheme given above is fair if different nodes are generating updates at a close rate or if LCRs are not skewed. To see this, suppose that a node is generating updates at a much higher rate than the other nodes. Soon the LCR at this node becomes much greater than LCRs at the other nodes. Therefore, updates generated at this node get the lowest priority among the updates generated in the system. Some authors have suggested means of controlling this situation by proposing synchronizing LCRs,[5,12] which tends to be expensive.

Second,

$$ID_i = (NN_i, LCR_i) \text{ and } ID_j = (NN_j, LCR_j)$$

which means that

$ID_i = ID_j$ if and only if $NN_i = NN_j$ *and* $LCR_i = LCR_j$,
$ID_i > ID_j$ if and only if $(NN_i > NN_j)$ *or* $(NN_i = NN_j$ *and* $LCR_i > LCR_j)$,
$ID_i < ID_j$ if and only if $(NN_i < NN_j)$ *or* $(NN_i = NN_j$ *and* $LCR_i < LCR_j)$.

This scheme solves the problem of skewed clocks but has another potential drawback. Since in this scheme dominance is given to node number NN, all updates generated from the node numbered $N-1$ have lower priority than updates from the node numbered $N-2$, updates generated at Node $N-2$ have lower priority than updates from Node $N-1, \ldots,$ and updates from Node 1 have lower priority than updates from Node 0. According to the above definition, this scheme is not fair either. To solve the fairness problem of this scheme, we suggest the MOD numbering scheme.

THE MOD NUMBERING SCHEME

As before, the MOD numbering scheme assumes that the nodes of the system are numbered 0 to $N-1$ at system initiation time. Since the problems mentioned above stem from fixed node numbers, the MOD scheme suggests that the node numbers be changed periodically and dynamically, as follows:

$$\text{New NN} = (\text{old NN} + 1) \text{ MOD } N$$

which means that Node 0 becomes 1 and Node 1 becomes 2, ..., and node number $N-1$ becomes 0. Changing the node numbers this way solves the problem of having a biased priority scheme but creates the problem of having two or more updates with the same ID numbers. For example, assume that the LCR at Node 2 is 4 and the LCR and Node 3 is 5. This means that Node 3 has already generated an update numbered (3,4). Now assume that Node 2 changes its node number to 3. The very next update generated at this node will also be numbered (3,4).

This problem can be solved by using a node sequence number, SN, as a third part of the update ID numbers. Using SN concatenated with NN, update ID numbers become

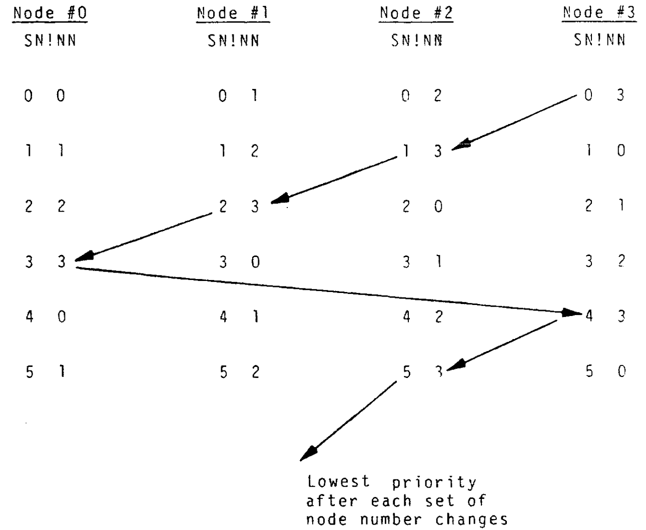


Figure 1—SN!NN for a 4-node system

$$ID = (SN!NN, LCR)$$

where ! denotes concatenation.

SN is set originally to 0 at each node and is incremented each time the node changes its node number.

Figure 1 shows SN!NN for a system of four nodes for the first five node number changes. The dominant factor in this scheme is SN!NN; i.e., for two updates i and j with $ID_i = (SN_i!NN_i, LCR_i)$ and $ID_j = (SN_j!NN_j, LCR_j)$,

$ID_i = ID_j$ if and only if $SN_i!NN_i = SN_j!NN_j$ *and* $LCR_i = LCR_j$,
$ID_i > ID_j$ if and only if $(SN_i!NN_i > SN_j!NN_j)$
or
$(SN_i!NN_i = SN_j!NN_j$ and $LCR_i > LCR_j)$,
$ID_i < ID_j$ if and only if $(SN_i!NN_i < SN_j!NN_j)$
or
$(SN_i!NN_i = SN_j!NN_j$ *and* $LCR_i < LCR_j)$.

To show that IDs generated this way are unique, it is sufficient to show that SN!NN for any given node is unique over the system. To do this, we have to show that for any two nodes i and j at any time either $SN_i \neq SN_j$ or $NN_i \neq NN_j$.

Suppose $SN_i!NN_i = SN_j!NN_j$ for two different nodes i and j. This means that $SN_i = SN_j$ and $NN_i = NN_j$. Let us assume that $SN_i = SN_j = s$, which is the number of times that these nodes have changed their numbers (see definition of SN). If the node number of node i at the system initiation time is ni and the node number of node j at the system initiation time is nj, then

$$NN_i = (ni + s) \text{ MOD } N$$

and

$$NN_j = (nj + s) \text{ MOD } N$$

If NN_i is to be equal to NN_j, then

$$(ni + s) \text{ MOD } N = (nj + s) \text{ MOD } N$$

The only way that this equality can hold is that if

$$ni + s = nj + s + KN \qquad \text{for } K \geq 0$$

or if

$$ni = nj + KN$$

Since $0 \leq ni < N$ and $0 \leq nj < N$ (see initial numbering of the nodes), the only value that K can have is 0, and therefore $ni = nj$, which contradicts the fact that all nodes are *uniquely* numbered at the system initiation time. Hence $ni \neq nj$, which means $NN_i \neq NN_j$ or $SN_i!NN_i \neq SN_j!NN_j$.

Note that besides being unique, SN!NN, generated as above, evenly distributes priority among the nodes of the system. In Figure 1, Node 3 (at the first row) has the highest SN!NN, whereas after the first node number change its SN!NN drops to the lowest (at the second row). Node 2, which had the second highest SN!NN at the beginning, will have the highest SN!NN after the first change (second row). Figure 1 shows how the highest SN!NN or lowest priority is passed from one node to another in a round-robin fashion.

There are two ways of initiating the node number changes. The first scheme calls for a timer at each node. A node changes its node number, according to the above scheme, when its interval timer expires. At this time the timer is reset and the sequence number is also incremented. The problem with interval timers is similar to the problem with physical clocks. To avoid this problem the second scheme can be used. In this scheme every node changes its node number after it generates M (a predefined integer number of) updates. The problem with this scheme is that lightly loaded nodes change their numbers more slowly than heavily loaded nodes. The tradeoffs between the two schemes must be investigated with regard to a specific application.

After a node changes its number, the LCR at that node can

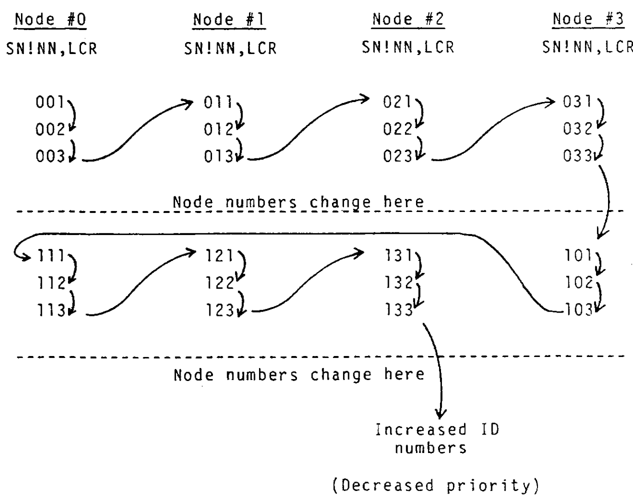


Figure 2—SN!NN,LCR for a 4-node system with M = 3 and LCR restart

	Node #0 SN!NN,LCR		Node #1 SN!NN,LCR	Node #2 SN!NN,LCR
	001		011	021
	002		012	022
	111		121	101
	112		122	102
A →		improper SN_reset		
	221	021	201	211
	222	022	202	212
B →		proper SN_reset		
	301	001	311	321
	302	002	312	322

Figure 3—Resetting SNs

be reset to 1 without threatening the local ordering of updates from the same node. This is necessary because otherwise LCRs may become undesirably large. Figure 2 shows some of the update ID numbers generated for a system of four nodes when each node changes its NN after generating $M = 3$ updates. This figure shows that updates generated from the same node are numbered in order of their generation. Therefore, even though LCRs are reset for each node number change, the local ordering is still preserved.

SNs similar to LCRs can grow large (although at a slower rate) and therefore require periodic resetting. Resetting SNs has to be done so that the properties of uniqueness, local ordering, and total relative ordering of the MOD numbering scheme are preserved. One has to be careful about when to reset a nodal SN. Since NNs and LCRs change in a circular manner, it is possible to generate two or more updates with the same ID when SNs are reset carelessly.

An example of a three-node system that changes a node's number after the node generates two updates is given in Figure 3. This figure shows that if SN of node number 0 is reset to 0 at Point A, the very next update generated at this node is numbered (0!2,1), which was first assigned to another update by Node 2 (first row, last column of Figure 3). In order to avoid this (and possible message transfer for synchronization), Node 0 can wait and attempt to reset its SN at Point B. There are two important properties associated with Point B. First, at this point, the new node number of every node is the same as its original number assigned at the system initiation time (0 for Node 0, etc). Second, if resetting is done at this point, the only possible conflicts are local conflicts: Node 0 might generate ID = (0!0,1), which was first generated by this node. This property eliminates the need for message transfers and synchronization with other nodes. Therefore, resetting SNs at this point will not cause uniqueness destruction of IDs if each node is only assured that all updates it has generated since its last SN reset are finished and out of the system. This might delay the numbering of updates if all previous updates are not finished. Considering the facts that each update is executed in a finite period and that resetting of SNs

does not occur very often, the delay is not substantial. Note that SNs can be reset independently for each node and do not have to be reset for all nodes at the same time. Note also that resetting a SN at a node means starting IDs from the lowest possible number at that node. As far as the local ordering of updates is concerned, this does not matter, because all previous updates at this node are out of the system when resetting occurs.

A virtual ring among the nodes and a token circulating in this ring, similar to the scheme explained in Lelann,[6] can also be used instead of the numbering scheme presented above. In this scheme a token (or a sequencer [Reed[8]]) is circulating in a prespecified virtual ring among the nodes of the system. The token is given a token round number, TRN, that is set to 0 at system initiation time and is incremented for each complete rotation of the token in the ring. A node will change its node number every time it receives the token. The TRN is attached to update ID numbers instead of SNs; i.e.,

$$ID = (TRN!NN,LCR)$$

This scheme also provides for a unique identification and a fair priority scheme among the updates. One drawback to this scheme is the problem of token loss, which may occur if a node that has the token fails. Loss of the token, even though soluble,[6] can delay the numbering procedure and hence contribute to delay in the execution of the updates. Link failures can cause similar problems. The MOD numbering scheme, on the other hand, does not require communication among the nodes to generate timestamps. This means that link failures do not affect timestamp generation. As far as node failures are concerned, a node can fail without interrupting other nodes' timestamp generation. After a node recovers, it can resume its timestamp generation where it left off. Because of the problem associated with the scheme using circulating tokens, it is preferable to use the MOD numbering scheme for numbering the events (updates) in a distributed system.

In summary, the scheme has four properties:

1. IDs generated using this scheme are unique.
2. For a given node, update IDs increase monotonically, and therefore updates generated from a node preserve the order of their generation (local ordering).
3. As discussed above, priority of nodes changes in a round-robin fashion and is not pre-fixed.
4. Control is local and therefore communication cost is low.

CONCLUSION

A numbering scheme that generates globally unique update IDs has been presented. The scheme dynamically changes the node numbers; this change results in an even distribution of priority across the nodes. The scheme does not require physical clocks and therefore avoids all the problems associated with synchronizing them. The MOD numbering scheme could be employed by update algorithms[1,7,9,11] in place of numbering schemes using fixed node numbers and physical clocks. This is expected to improve the performance of these algorithms.

REFERENCES

1. Bernstein, P. A., J. B. Rothnie, N. Goodman, and C. A. Papadimitriou. "The Concurrency Control Mechanism of SDD-1: A System for Distributed Data Bases (The Fully Redundant Case)." *IEEE Transactions on Software Engineering,* SE-4, (1978).
2. Date, C. J. "An Introduction to Database Systems." Addison-Wesley, Reading, Massachusetts, 1977.
3. Everest, G. C. "Concurrent Update Control and Database Integrity." In J. W. Klimbie and K. L. Koffeman (eds.), *Database Management.* Amsterdam: North-Holland, 1974, pp. 241–268.
4. Eswaran, K. P.; J. N. Gray, R. A. Lorie, and I. L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System." *Communications of the ACM,* 19 (1976).
5. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM,* 21 (1978).
6. Lelann, G. "Algorithms for Distributed Data-Sharing Systems Which Use Tickets." *Proc. Third Berkeley Workshop on Distributed Data Base and Computer Networks*, University of California, Berkeley, CA, August 1978.
7. Rahimi, S. K., and W. R. Franta, "A Posted Update Approach to Concurrency Control in Distributed Data Base Systems." *Proc. 1st Intl. Conf. on Distributed Computing Systems,* IEEE, Oct. 1979.
8. Reed, D. P. "Naming and Synchronization in a Decentralized Computer System." Ph.D. thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, September 1978.
9. Rosenkrantz, D. J., R. E. Stearns, and P. M. Lewis, II. "System Level Concurrency Control for Distributed Database Systems." *ACM Transactions on Database Systems,* 3 (1978).
10. M. Stonebraker. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES." *3rd Berkeley Workshop on Distributed Data Management,* 1978.
11. Thomas, R. H. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases." *ACM Transactions on Database Systems,* 4 (1979).
12. Traiger, I. L., J. N. Gray, C. A. Galtieri, and B. G. Lindsay. "Transactions and Consistency in Distributed Data Base Systems." IBM Report RJ2555 (33155), May 1979.

Data abstraction for Pascal programmers

by VISWANATHAN SANTHANAM and JOHN R. POTOCHNIK
Wichita State University
Wichita, Kansas

ABSTRACT

Lack of data abstraction facilities in Pascal is a serious shortcoming for a language whose principal aims include "teaching programming as a systematic discipline." In this paper, a scheme is presented to implement data abstraction in Pascal using a limited form of the class structure defined in Concurrent Pascal. The class structures are translated to equivalent (sequential) Pascal constructs with the help of a preprocessor whose design is also described.

INTRODUCTION

Use of abstraction techniques to improve program reliability and programmer productivity can be traced back to early computer languages such as FORTRAN. Subroutines and functions of most algorithmic languages represent abstraction of operations. However, it is only in the past decade or so that programming language constructs have been devised to extend the idea of abstraction to data as well as operations in a unified manner. The term *data abstraction* has been applied to implementation-independent characterization and usage of data. Among the languages that provide data abstraction capabilities are MODULA,[1] Concurrent Pascal,[2] and Ada.[3]

Most notable among the recent programming languages that do not support data abstraction is Pascal[6]—a language that is growing in use, especially among small computer users and educators. The reason for this omission is perhaps related to the desire on the part of its architect[4] to keep the language simple. Whatever the reason, the absence of even a rudimentary form of data abstraction capability has to be termed a vacuum in the language. This paper describes a scheme that will fill this vacuum.

Interestingly, most languages that currently support data abstraction also support concurrent programming, leading a casual observer to believe that data abstraction somehow naturally belongs in the realm of concurrent programming. The coincidence is largely due to the fact that the need for orderly sharing of data between concurrent processes naturally leads to centralizing such data and related operations. Though there is no particular need for hiding the implementation details to make this sharing work, it turns out to be relatively easy to add this quality to the notion of centralization. The concurrent data- (or resource-) sharing facilities are termed *monitors*[1,2]; the sequential abstraction facilities are termed *classes*.[2,5]

This work presents a data abstraction facility for (sequential) Pascal users. The Pascal language is extended to include a form of class structure similar to that of Concurrent Pascal. However, unlike Concurrent Pascal and other extensions of Pascal that provide similar capabilities (see Steensgard-Madsen,[5] for example), there is no need in this approach for a special compiler or operating environment to compile or execute a program. We present the design of a "preprocessor" that will translate the extensions to suitable constructs in sequential Pascal. This approach has the advantage of being applicable to existing Pascal systems. The source-to-source translation approach has the additional pedagogical value of illustrating a moderately simple technique to realize data abstraction in languages that do not support such a facility.

EXTENSIONS

The central theme of our approach is to extend sequential Pascal to include classes. A preprocessor will then accept programs in this extended language and translate them to equivalent programs in the standard language. The first extension is the introduction of a new type called *class*. Its syntax and semantics are essentially the same as those in Concurrent Pascal, with minor variations:

```
type CLASSNAME = class [(<formal parameters>)];
  <class block>
```

where the optional formal parameters serve to instantiate (expand) the class for a specific application. For example,

```
type STACK = class(ELEMTYPE, MAXSIZE);
```

would allow us to define a generic class type for a variety of different element types and stack sizes. The preprocessor will treat these parameters like macro parameters, replacing them with the texts of the corresponding actual parameters when the class block is expanded.

The class block is similar to an ordinary subprogram block in Pascal, with a few differences. Any function or procedure within a class block may be designated as an *entry* function or procedure. The syntax for this extension is defined by

```
procedure entry PROCNAME [(<formal parameters>)];
```

and

```
function entry FUNCNAME [(<formal parameters>)]:
  FUNCTYPE;
```

For example,

```
function entry POP: ELEMTYPE;
```

Entry subprograms must be local subprograms within the class and not be nested deeper. The visibility (scope) of such subprograms is the same as that of the class block itself.

The statement part of a class block consists of an initialization sequence with this syntax:

```
begin init
  <statement sequence>
end;
```

Defining a class type allows the programmer to declare class variables of that type. This is done by a *var* statement, as is done for declaring any other variable:

```
var <variable name list> : CLASSNAME [(<actual
  parameters>)];
```

The actual parameters, if any, textually replace the corresponding formal parameters in the class block when the latter

```
program EXAMPLE( INPUT, OUTPUT );

type STACK = class( ELEMTYPE, MAXSIZE );
  type STKTABLE = array[1..MAXSIZE] of ELEMTYPE;
  var PTR: INTEGER;
      STK: STKTABLE;
  function entry EMPTY: BOOLEAN;
  begin
    EMPTY := PTR<1
  end;
  function entry FULL: BOOLEAN;
  begin
    FULL := PTR>MAXSIZE
  end;
  procedure entry PUSH( OBJ: ELEMTYPE );
  begin
    if not FULL then begin
      PTR := SUCC(PTR);
      STK[PTR] := OBJ
      end
  end;
  function entry POP: ELEMTYPE;
  begin
    if not EMPTY then begin
      POP := STK[PTR];
      PTR := PRED(PTR)
      end
  end;
  begin init
    PTR := 0
  end;

var S1, S2: STACK( INTEGER, 100 );
    I: INTEGER;

begin (* MAIN PROGRAM *)
  while (not EOF) and (not S1.FULL) do begin
    READLN( I );
    S1.PUSH( I )
    end;
  while not S1.EMPTY do
    S2.PUSH( S1.POP )
end.
```

Figure 1—An example of data abstraction using class

is instantiated. Thus it is possible to declare several distinct classes from a single type definition. For example,

```
var SI: STACK(INTEGER,100);
    SR: STACK(REAL,200);
```

sets up an integer stack and a real stack of different sizes from the same class type STACK.

Operations on class variables are restricted to those defined by entry subprograms in their class types. Invocations of these operations appear in the form of procedure and function calls qualified by the class variable name. For example, SI.PUSH (−401) invokes the PUSH operation (an entry procedure) within the class type of SI.

To illustrate these extensions and their role in data abstraction, an example of a complete program is shown in Figure 1. The program contains the definition of a class type called STACK. Two parameters, ELEMTYPE and MAXSIZE, will allow different instantiations to be derived from this single generic definition. The class block includes several definitions and declarations that are private entities within the class block. They represent the implementation details of the abstract data type STACK. There are four entry subprograms within the class block. They represent the predefined operations on the STACK that are available to the creators of STACK-type variables. They enclose the implementation details of the respective operations, as any subprogram does. The initialization sequence for the class consists of setting the stack pointer PTR to zero.

The main program includes a declaration for two integer STACKs, each capable of holding up to 100 elements. The statement part of the program reads integers from the input and pushes them on S1 stack until it gets full or the input is exhausted. Then, the elements are transferred from S1 to S2 one at a time.

Though the class construct described in this section is similar in most respects to that of Concurrent Pascal (see Hansen[2] for an extensive discussion on the nature and use of classes), there are some important differences. Both systems allow parameters to be specified in a class definition, but for different purposes. Also, in Concurrent Pascal, class variables have no special restrictions on where they may be defined or how they may be used. In our system, they may not be part of a larger data structure, such as an array or a record, nor may they be passed as parameters to subprograms. These latter restrictions are necessary for circumventing the limitations of Pascal and difficulties of translation.

PREPROCESSOR

The definition of a class type is treated by the preprocessor as a macro-definition. The macro itself is expanded (instantiated) only when a variable of that class type is declared. During the expansion the actual parameters replace the corresponding formal parameters in the class definition. This substitution alone will not yield an acceptable Pascal program. Several other transformations are needed, and those are described in this section.

A class block has a dual function. In most respects it is like any subprogram block enclosing private definitions of constants and types and declarations of variables and subprograms. However, there are some key exceptions to this similarity rule. Entry subprograms are not strictly private to the class block; they are callable from blocks that declare variables of that class type. (These latter blocks are called *host blocks*.) This is in contrast to the usual visibility rules of Pascal. Additionally, variables declared within the class block cannot be treated like ordinary local variables, because they are to be allocated when the host block is entered and retained throughout the lifetime of the host block. In other words, their creation and destruction must coincide with the creation and destruction of variables in the host block. But their visibility is to follow the usual block structure rules.

There seems to be no simple way to accommodate the dual characteristics of a class block in Pascal. Our approach is to strip the class block of its enclosing property but retain the privacy of identifiers defined within it by transforming them to

unique identifiers that do not conflict with those in the host block. The transformed definitions (constants, types) and declarations (variables, subprograms) are added to the respective groups in the host block. Of course, there is no need to transform names that are locally defined within the subprograms in a class block, because their uniqueness is guaranteed by the usual scope rules of block structure.

The above scheme nicely solves the dichotomy of entry subprograms. Type and constant definitions are also handled adequately by this transformation. However, variables and labels of the class block must be handled somewhat differently.

Variables in the class block represent actual data structures required to implement the abstract data type. Each class variable declared in the host block would require its own set of these data structures. For example,

var S1, S2: STACK(INTEGER,20);

declares two distinct variables of the abstract type STACK. If STACK is implemented by using a linear array in the class (as in the example of Figure 1), the expansion of the class definition must result in two distinct arrays. This is achieved in our scheme by introducing into the host block one set of class block variables for each class variable declared in the host block (see Figures 1 and 3 for an example). The name transformation technique will ensure that there are no conflicts in the host block.

The duplication of class block variables could lead to other duplications during the instantiation of a class. For example, a subprogram in the class block may reference a class block variable. This could mean that we need to duplicate such subprograms, making one copy for each class variable declared in the host block. This could be costly if there are several such subprograms and/or they are lengthy. Our approach avoids such duplication by eliminating any direct reference to class block variables within a subprogram. Instead, all data structures of the class block are passed as reference *(var)* parameters to each subprogram. Though it may not be necessary to pass all data structures to all subprograms, this simple scheme avoids the need for scanning the subprograms to determine which data structures are needed. In addition to modifying the subprogram definitions, instances of calls to these subprograms are also modified by adding a corresponding set of actual parameters. In the case of calls originating in the host block, the additions consist of class block variables associated with the specific class variable qualifying the call. For calls within the class block, the additions consist of the same names as those added to all subprogram headers. (See Figures 1 and 3 for examples.)

The initialization sequence of the class block is transformed into a special procedure. This procedure is invoked once for each class variable in the host block by including suitable calls at the top of the statement part of the host block (see Figures 1 and 3 for example). The name for this special procedure is derived from the class name so as to avoid conflicts, and all class block variables are passed by reference to this procedure as well. If there are any labels declared in the original class block, they must refer to statement labels in the initialization sequence. Therefore, the class block label declarations, if any, are added to this new procedure, rather than to the host block, without any transformations.

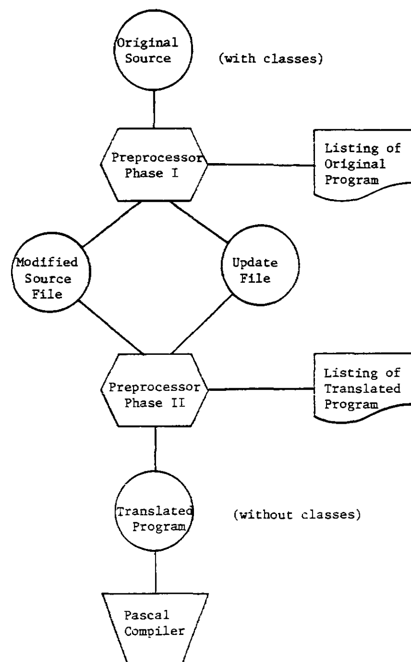


Figure 2—The preprocessor phases

The preceding set of transformations applies to a single list of class variables encountered in the host block. Each such list would cause the class definition to be expanded according to the same rules. An example of this situation is depicted in Figure 4, which is discussed in the next section. At this time, there are no provisions in our approach for avoiding redundant generation of identical subprograms (under different names) when a declaration of the form

var S1: STACK(INTEGER,100);
S2: STACK(INTEGER,50);

is encountered.

As the reader may have noted by now, the transformations required to generate standard Pascal constructs equivalent to the class structure cannot all be achieved in one pass. For example, when a class variable declaration triggers an instantiation of the class type, several new definitions and declarations may be added to the host block. Not all these additions can be placed at one point in the host block. The constant definitions, for example, must be added to others that may be defined in the host block. Since constant definitions must precede variable declarations in Pascal, the preprocessor would have to backtrack to make the additions in the right place. To avoid these potentially costly backtracks, the preprocessor design presented here works in two phases (see Figure 2). The first phase takes the original program (with classes) as input and produces a modified source file and an update file. The modified source file consists of the original source file with two major differences: (a) the class type definitions, if any, have been removed; (b) all identifier transformations and other simpler transformations (such as extension of the parameter lists of class block subprograms and

```
program EXAMPLE( INPUT, OUTPUT );
type STACK#STKTABLE = array[1..100] of INTEGER;
var S1#PTR, S2#PTR: INTEGER;
    S1#STK, S1#STK: STACK#STKTABLE;
    I: INTEGER;
  function STACK#EMPTY( var STACK#PTR: INTEGER;
    var STACK#STK: STACK#STKTABLE ): BOOLEAN;
  begin
    STACK#EMPTY := STACK#PTR<1
  end;
  function STACK#FULL( var STACK#PTR: INTEGER;
    var STACK#STK: STACK#STKTABLE ): BOOLEAN;
  begin
    STACK#FULL := STACK#PTR>100
  end;
  procedure STACK#PUSH( var STACK#PTR: INTEGER;
    var STACK#STK: STACK#STKTABLE; OBJ: INTEGER );
  begin
    if not STACK#FULL(STACK#PTR, STACK#STK) then begin
      STACK#PTR := SUCC(STACK#PTR);
      STACK#STK[STACK#PTR] := OBJ
      end
  end;
  function STACK#POP( var STACK#PTR: INTEGER;
    var STACK#STK: STACK#STKTABLE ): INTEGER;
  begin
    if not STACK#EMPTY(STACK#PTR, STACK#STK) then begin
      STACK#POP := STACK#STK[STACK#PTR];
      STACK#PTR := PRED(STACK#PTR)
      end
  end;
  procedure STACK#INIT( var STACK#PTR: INTEGER;
    var STACK#STK: STACK#STKTABLE );
  begin
    STACK#PTR := 0
  end;
begin
  STACK#INIT( S1#PTR, S1#STK );
  STACK#INIT( S2#PTR, S2#STK );
  while (not EOF) and (not STACK#FULL(S1#PTR, S1#STK)) do begin
    READLN( I );
    STACK#PUSH( S1#PTR, S1#STK, I )
    end;
  while not STACK#EMPTY(S1#PTR, S1#STK) do
    STACK#PUSH( S2#PTR, S2#STK, STACK#POP(S1#PTR, S1#STK) )
end.
```

Figure 3—Translation of program in Figure 1

their invocations) are in place. The update file consists of texts to be inserted at selected points in the host block. Each update is keyed for random access by a special marker. A copy of the marker is placed in the modified source file where the corresponding text is to be inserted. The points within the host block where potential insertions may take place are only a few:

1. Constant definitions
2. Type definitions
3. Variable declarations
4. Subprogram declarations
5. Top of statement part

The design of the output files from the first phase of the preprocessor minimizes the complexity of the second phase by reducing its function to one of scanning the modified source file, sequentially looking for markers, and merging in the corresponding text from the update file. The advantage, of course, is that the merging phase needs to have no knowledge of the syntax of Pascal. The overall design of the preprocessor is depicted in Figure 2.

```
program TWOQS(OUTPUT);
type QUEUE = class(ELEMTYPE,QSIZE);
  type QTABLE = array[1..QSIZE] of ELEMTYPE;
  var Q: QTABLE;
      HEAD, TAIL: INTEGER;
  function entry EMPTY: BOOLEAN;
  begin
    EMPTY := (HEAD=0) and (TAIL=0);
  end;
  function entry FULL: BOOLEAN;
  (* details omitted *)
  procedure entry ENTER( E: ELEMTYPE );
  begin
    if not FULL then begin
      TAIL := SUCC(TAIL mod (QSIZE-1));
      Q[TAIL] := E
      end
  end;
  function entry REMOVE: ELEMTYPE;
  (* details omitted *)
  begin init
    HEAD := 0;
    TAIL := 0
  end;

var QI: QUEUE( INTEGER, 20 );
    QR: QUEUE( REAL, 20 );
begin (* TWOQS *)
  while not QR.EMPTY do
    QI.ENTER( TRUNC(QR.REMOVE) )
end.
```

(a) Program before translation (with class).

```
program TWOQS(OUTPUT);
type Q1#QTABLE = array[1..20] of INTEGER;
     Q2#QTABLE = array[1..20] of REAL;
var QI#Q: Q1#QTABLE;
    QI#HEAD, QI#TAIL: INTEGER;
    QR#Q: Q2#QTABLE;
    QR#HEAD, QR#TAIL: INTEGER;

  (* the following subprograms result from QI instantiation *)
  function Q1#EMPTY( var Q1#Q: Q1#QTABLE;
                     var Q1#HEAD, Q1#TAIL: INTEGER): BOOLEAN;
  begin
    Q1#EMPTY := (Q1#HEAD=0) and (Q1#TAIL=0);
  end;
  function Q1#FULL( var Q1#Q: Q1#QTABLE;
                    var Q1#HEAD, Q1#TAIL: INTEGER): BOOLEAN;
  (* details omitted *)
  procedure Q1#ENTER( var Q1#Q: Q1#QTABLE;
                      var Q1#HEAD, Q1#TAIL: INTEGER, E: INTEGER);
  begin
    if not Q1#FULL(Q1#Q, Q1#HEAD, Q1#TAIL) then begin
      Q1#TAIL := SUCC( Q1#TAIL mod (20-1) );
      Q1#Q Q1#TAIL  := E
      end
  end;
  function Q1#REMOVE( var Q1#Q: Q1#QTABLE;
                      var Q1#HEAD, Q1#TAIL: INTEGER): INTEGER;
  (* details omitted *)
  procedure Q1#INIT( var Q1#Q: Q1#QTABLE;
                     var Q1#HEAD, Q1#TAIL: INTEGER);
  begin
    Q1#HEAD := 0;
    Q1#TAIL := 0
  end;

  (* the following subprograms result from QR instantiation *)
  function Q2#EMPTY( var Q2#Q: Q2#QTABLE;
                     var Q2#HEAD, Q2#TAIL: INTEGER): BOOLEAN;
  begin
    Q2#EMPTY := (Q2#HEAD=0) and (Q2#TAIL=0);
  end;
  function Q2#FULL( var Q2#Q: Q2#QTABLE;
                    var Q2#HEAD, Q2#TAIL: INTEGER): BOOLEAN;
  (* details omitted *)
  procedure Q2#ENTER( var Q2#Q: Q2#QTABLE;
                      var Q2#HEAD, Q2#TAIL: INTEGER, E: REAL);
  begin
    if not Q2#FULL(Q2#Q, Q2#HEAD, Q2#TAIL) then begin
      Q2#TAIL := SUCC( Q2#TAIL mod (20-1) );
      Q2#Q Q2#TAIL  := E
      end
  end;
  function Q2#REMOVE( var Q2#Q: Q2#QTABLE;
                      var Q2#HEAD, Q2#TAIL: INTEGER): REAL;
  (* details omitted *)
  procedure Q2#INIT( var Q2#Q: Q2#QTABLE;
                     var Q2#HEAD, Q2#TAIL: INTEGER);
  begin
    Q2#HEAD := 0;
    Q2#TAIL := 0
  end;

begin (* TWOQS *)
  Q1#INIT( QI#Q, QI#HEAD, QI#TAIL );
  Q2#INIT( QR#Q, QR#HEAD, QR#TAIL );
  while not Q2#EMPTY(QR#Q,QR#HEAD,QR#TAIL) do
    Q1#ENTER(QI#Q,QI#HEAD,QI#TAIL,
             TRUNC( Q2#REMOVE(QR#Q,QR#HEAD,QR#TAIL) ))
end.
```

(b) Translation of program in (a)

Figure 4—An example of two instantiations of the same class

EXAMPLES

A set of three graduated examples is presented in this section to illustrate the use of class structures, their translation, and the translation process itself.

In the first example, the abstract data type STACK of Figure 1 is revisited. The translation of the EXAMPLE program appears in Figure 3. For the sake of better readability, the name transformations are shown as concatenations of component names from which they are derived. For example, STACK#PTR stands for a unique name to be derived from STACK and PTR. In practice, such concatenation may be too simplistic to avoid conflicts. Nevertheless, by placing some restrictions on the use of special characters (such as #), it should be possible to retain a level of readability in the "production" translation comparable to that of the translations shown here.

The new additions to the host block EXAMPLE due to the instantiation of the class STACK by the declaration of S1 and S2 are enclosed in boxes for easy identification. The type name STACK#STKTABLE, for example, comes from the type definition for STKTABLE in STACK class. It is noted that in the type definition for STACK#STKTABLE and throughout the remainder of the class the formal parameters ELEMTYPE and MAXSIZE have been replaced by the actual parameters INTEGER and 100 respectively.

Each local subprogram of the class block has been extended to include the data structures originally defined as variables in the class block. The actual parameters list of external calls to these subprograms has been extended to include the associated variables. For example, S1.PUSH(I) in the original program has been translated to STACK#PUSH(S1#PTR, S1#STK,I). S1#PTR and S1#STK are the variables associated with the abstract data item S1.

Procedure STACK#INIT is a new procedure created from the initialization sequence of the class block. It too has a formal parameter list of original class block variables added to its header. At the top of the statement part of the host block are two new statements, one each to invoke this procedure for S1 and S2.

Figure 4 shows a second example involving two distinct instantiations of the same class. Two important features of the preprocessing step are borne out by this example. Each instantiation (one for QI and another for QR) generates its own set of additions to the host block. It is not sufficient, therefore, to employ just the class name QUEUE to transform class block names. For example, if we generated QUEUE#QTABLE from QTABLE for the first instantia-

```
program DRIVER;
const ISAMFILESIZE = 1000;
type ASSOCMEMORY = class(ASSOCELEMTYPE);
  const MAXHASHINDEX = 97;
  type ASSOCTABLE = array[0..MAXHASHINDEX] of ASSOCELEMTYPE;
       INDEXSET = set of 0..MAXHASHINDEX;
  var ASSOC: ASSOCTABLE;
      EMPTY, DELETED: INDEXSET;
    function entry LOCATION( A:ASSOCELEMTYPE ): INTEGER;
    (* delatils omitted *)
    (* other entries of ASSOCMEMORY class go here *)
  begin init
    DELETED := [];
    EMPTY := [0..MAXHASHINDEX]
  end;

(* type *) ISAMFILE = class(KEYTYPE, DATATYPE, FILESIZE);
  type DATAFILE = file of DATATYPE;
  var KEYS: ASSOCMEMORY(KEYTYPE);
      RECS: DATAFILE;
    procedure entry READ(K:KEYTYPE; var R: DATATYPE; var FOUND: BOOLEAN);
    var RECNO: INTEGER;
    begin
      RECNO :=KEYS.LOCATION(K);
      FOUND := RECNO <> -1;
      if FOUND then begin
        SEEK( RECS, RECNO );
        GET(RECS);
        R := RECS↑
        end
    end;
    (* other entries of ISAMFILE class go here *)
  begin init
    (* details omitted *)
  end;

var F: ISAMFILE(MYKEYTYPE, MYRECTYPE, ISAMFILESIZE);
    K: MYKEYTYPE;
    R: MYRECTYPE;
    FOUND: BOOLEAN;

begin
  READLN(K);
  F.READ( K, R, FOUND );
  if FOUND then DISPLAY(R)
  else WRITELN( 'No such key' )
end.
```

(a) Program before translation (with classes).

```
procedure DRIVER;
const ISAMFILESIZE = 1000;
      IS#AS#MAXHASHINDEX = 97;
type IS#AS#ASSOCTABLE = array[0..IS#AS#MAXHASHINDEX] of MYKEYTYPE;
     IS#AS#INDEXSET = set of 0..IS#AS#MAXHASHINDEX;
     IS#DATAFILE = file of MYRECTYPE;
var F#KEYS#ASSOC: IS#AS#ASSOCTABLE;
    F#KEYS#EMPTY, F#KEYS#DELETED: IS#AS#INDEXSET;
    F#RECS: IS#DATAFILE;
    K: MYKEYTYPE;
    R: MYRECTYPE;
    FOUND: BOOLEAN;

  function IS#AS#LOCATION( var IS#AS#ASSOC: IS#AS#ASSOCTABLE;
    var IS#AS#EMPTY, IS#AS#DELETED: IS#AS#INDEXSET; A: MYKEYTYPE): INTEGER;
  (* details omitted *)
  (* other entries of ASSOCMEMORY class go here *)
  procedure IS#AS#INIT( var IS#AS#ASSOC: IS#AS#ASSOCTABLE;
    var IS#AS#EMPTY, IS#AS#DELETED: IS#AS#INDEXSET );
  begin
    IS#AS#DELETED := [];
    IS#AS#EMPTY := [0..IS#AS#MAXHASHINDEX]
  end;

  procedure IS#READ( var IS#KEYS#ASSOC: IS#AS#ASSOCTABLE; var IS#KEYS#EMPTY,
    IS#KEYS#DELETED: IS#AS#INDEXSET; var IS#RECS: IS#DATAFILE;
    K: MYKEYTYPE; var R: MYRECTYPE; var FOUND: BOOLEAN );
  var RECNO: INTEGER;
  begin
    RECNO := IS#AS#LOCATION( IS#KEYS#ASSOC, IS#KEYS#EMPTY, IS#KEYS#DELETED,
                             K );
    FOUND := RECNO <> -1;
    if FOUND then begin
      SEEK( IS#RECS, RECNO );
      GET( IS#RECS );
      R := IS#RECS↑
      end
  end;
  (* other entries of ISAMFILE class go here *)
  procedure IS#INIT( var IS#KEYS#ASSOC: IS#AS#ASSOCTABLE; var IS#KEYS#EMPTY,
    IS#KEYS#DELETED: IS#AS#INDEXSET; var IS#RECS: IS#DATAFILE );
  begin
    IS#AS#INIT( IS#KEYS#ASSOC, IS#KEYS#EMPTY, IS#KEYS#DELETED )
    (* details omitted *)
  end;

begin
  IS#INIT( F#KEYS#ASSOC, F#KEYS#EMPTY, F#KEYS#DELETED, F#RECS );
  READLN( K );
  IS#READ( F#KEYS#ASSOC, F#KEYS#EMPTY, F#KEYS#DELETED, F#RECS,
    K, R, FOUND );
  if FOUND then DISPLAY(R)
  else WRITELN( 'No such key' )
end.
```

(b) Translation of program in (a).

Figure 5—An example of nested class instantiations

tion, it would conflict with the name to be generated during the second instantiation. In our preprocessor, this situation is handled by using a sequence number (in conjunction with the class name) to generate unique names for each instantiation. Thus, the first instantiation would yield the type name QUEUE1#QTABLE (abbreviated as Q1#QTABLE in Figure 4), and the second instantiation would yield QUEUE2#QTABLE (Q2#QTABLE in the figure) for the original name QTABLE.

The second point illustrated by this example is that amount of translated code increases rapidly when several instantiations are invoked from a single class definition. Not all of this increase can be avoided. For example, the procedures Q1#ENTER and Q2#ENTER, though similar at the surface, deal with distinct types of data elements (INTEGER and REAL). In some cases (e.g., Q1#EMPTY and Q2#EMPTY), however, our scheme does yield multiple copies of virtually the same subprogram. The avoidance of this subtle form of duplication would require more extensive analysis of the original class definition than provided for in our preprocessor.

The third example (Figure 5) shows two classes, ASSOCMEMORY and ISAMFILE, defined in the same block, DRIVER. An instance of the ISAMFILE class is created by the declaration of the variable F in DRIVER. This causes variables F#KEYS and F#RECS to be added to DRIVER block. But F#KEYS is itself a class type (ASSOCMEMORY) variable. Thus, F#KEYS generates F#KEYS#ASSOC, F#KEYS#EMPTY, and F#KEYS#DELETED in its place. This kind of secondary instantiation is not handled any differently in our preprocessor than the single level case. Conceptually, the outermost class variables are instantiated first, giving rise to an intermediate program. This program is then processed as before to eliminate class variables from the new variables, and so on. This conceptual view will help explain the double transformation of certain original names in the translation, e.g., LOCATION to ISAMFILE#ASSOCMEMORY#LOCATION (abbreviated as IS#AS#LOCATION). In practice, the secondary instantiations can be achieved without a second pass over the text. This is done by processing all declarations (variables and subprograms) stemming from the instantiation of a class at the time that they are being added to the host block as if they were originally part of the host block.

CONCLUSION

We have presented here a technique for implementing data abstraction in Pascal using a limited form of the class structure

provided in Concurrent Pascal. The class structures are translated to equivalent (sequential) Pascal constructs using a preprocessor whose design is also presented. The abstract data types supported by this source-to-source translation scheme are somewhat limited, chiefly because of the limitations of Pascal and a desire on our part to keep the translation simple and straightforward. Abstract data items may not be passed as parameters to subprograms, because this would have involved passing a variety of resources—data structures, procedures, and functions—as parameters in the translation, not all of which are supported in common implementations of Pascal. We have limited abstract data items from being part of a larger data structure, such as an array or a record, in order to simplify the task of translation as well as maintain a closer correspondence between the original and the translated programs. Even with these limitations, the scheme presented will permit a Pascal programmer to define and use sophisticated abstract data types, such as the index file type illustrated here, with only a small cost for preprocessing.

REFERENCES

1. Wirth, N. MODULA: A language for modular multiprogramming. *Software Practice and Experience,* 7 (1977), pp. 3–35.
2. Brinch Hansen, P. *The Architecture of Concurrent Programming.* Englewood Cliff, N.J.: Prentice-Hall, 1977.
3. *Reference manual for the Ada programming language:* U.S. Dept. of Defense, July 1980.
4. Wirth, N. "The Programming Language Pascal." *Acta Informatica,* 1 (1971), pp. 35–63.
5. Steensgaard-Madsen, J. "A Statement-Oriented Approach to Data Abstraction." *ACM Transactions on Programming Languages and Systems,* 3 (1981), pp. 1–10.
6. Jensen, K., and N. Wirth, *Pascal: User Manual and Report.* 2nd ed. New York: Springer-Verlag, 1978.

SPIRIT-III: an advanced relational database machine introducing a novel data-staging architecture with Tuple Stream Filters to preprocess relational algebra

by NORIYUKI KAMIBAYASHI
Hiroshima University
Hiroshima, Japan
and
KAZUO SEO
Mitsubishi Electric Corporation
Amagasaki, Japan

ABSTRACT

This paper proposes an advanced architecture of the relational database machine (RDBM), named SPIRIT-III, which is basically organized into a three-level memory hierarchy with a sophisticated data-staging and preprocessing architecture for executing relational algebra. SPIRIT-III aims at totally improving both I/O and CPU processing boundary problems and has two major architectural features. One is the introduction of the relational-database-oriented data-staging mechanism, called the look-ahead data-staging mechanism, which can optimally schedule data movement in the memory hierarchy. The other is to attach refined preprocessing mechanisms for relational algebra operations to data transfer lines connected between each memory stage. When a relation stages up or down in the memory hierarchy, these preprocessing mechanisms can function to select tuples and attributes needed by a query and to arrange the relation for parallel processing. SPIRIT-III provides three basic preprocessing filters, called as a whole the Tuple Stream Filter: the tuple selector, the attribute selector, and the grouping filter, implemented with a hash function, which rearranges an original relation and groups the relation into subrelations. The operation of this grouping filter is the primitive preprocessing operation for executing Join and Projection. Then, without the overhead of interprocessor communications, each microprocessor can execute relational algebra operations to a few subsegments assigned to it in parallel. Therefore, SPIRIT-III can perform Join and Projection operations by O (N/L) (L = number of microprocessors), whereas the early RDBMs required O $(N \times N/L)$. The proposed SPIRIT-III, which includes features from data-staging architecture to relational algebra execution architecture under the total concept, is the most powerful RDBM based on the state of the art.

1. INTRODUCTION

The relational model[1] proposed by E. F. Codd is based on the set theory and has the advantages of simplicity, data independence, and symmetry of access. These features make it possible to provide nonprocedural query languages and high-level user interface. This tends to enhance the usability and the intelligence of database management systems. When one tries to implement a relational database system on a von Neumann type of general-purpose computer, there are crucial problems, such as poor capacity of data transfer and inefficient relational operation. Thus, the need to develop a relational database machine (RDBM) has emerged. The primary aims in designing RDBMs are to improve the execution time of sophisticated relational algebra operations such as Join and Projection and to reduce the cost of data transfer between the database stores (DBS) and the primary processing subsystem.

However, it has been very difficult to solve both the two major problems in one try. Because traditional RDBMs aimed at improving the efficiency of relational operations, these machines had restricted ability from the viewpoint of total relational database processing and could not provide the harmonized mechanism that would reduce the burdens of both data staging and relational operations.

We discuss the dilemma that system designers face in developing practical RDBMs, taking an example of RAP[3,4] projects. The RAP project, well known as a pioneer of RDBM, based on the concept of logic-per-track, employs architecture on which the associative processing mechanism is added on a fixed-head disk, used as DBS to transfer the final results of a query to the host. It has been noted, however, that RAP[3] has difficulty in supporting efficiently such complicated set manipulations as Projection and Join, and also in handling large databases because of its restriction on the capacity of DBS. In RAP.2,[4] therefore, the processing subsystem is separated from DBS and the moving head disk is employed as DBS. This approach is realistic in regard to background, such as technical maturity and possibility, and is advanced from the viewpoints of (1) implementing RDBMs based on the state of the art and (2) the need to support large databases. Bottlenecks of data access and transfer, however, still exist in such RDBMs. Thus it is important to solve the problem of the cost of large data staging at the system architectural level.

In this paper, in order to develop realistic RDBM that copes with this problem, we propose an advanced architecture of the relational database machine, named SPIRIT-III. It is basically organized into a three-level memory hierarchy (MH): primary work memory (PWM), staging buffer (SB), and database store (DBS). SPIRIT-III aims at totally solving both I/O and CPU boundaries, which are key problems in implementing RDBMs. It consists of three cooperative subsystems and has two major architectural features. One is the introduction of the relational-database-oriented data-staging mechanism which can contribute to the optimal scheduling of data access and transfer between each two levels of the memory hierarchy. The other is to attach refined preprocessing mechanisms of relational algebra operations to data transfer lines connected between each two levels of the memory hierarchy. When a relation stages up or down from DBS or PWM to SB, these preprocessing mechanisms can function to select the tuples and attributes to be staged up and to arrange the relation for parallel processing. SPIRIT-III provides three basic preprocessing functions. Two of these are the tuple selector, filtering only tuples to match retrieval conditions; and the attribute selector, repacking only attributes needed by executing a given query. Because these functions, which select only attributes and tuples required by the query, contribute to reducing the data to be staged up and to saving the space of upper-level memory, SPIRIT-III can enhance both the throughput of data staging in the data staging subsystem (DSS) and the execution efficiency of relational operations in the relational processing subsystem. In addition to these filtering functions, SPIRIT-III incorporates a grouping filter, proposed by the authors, into each stage of memory hierarchy. In streaming a relation on data lines connected between each two memory stages, this grouping filter groups the relation into subrelations. This grouping operation is the primitive of executing heavy relational operations, such as Join and Projection. In practice, this grouping filter can be implemented with a hash function. Microprocessors of RPS can execute postoperations associated with relational algebra operations to a few subsegments assigned to each of these in parallel without the overhead of interprocesser (subsegment) communications. The relational algebra execution architecture of SPIRIT-III is composed of two stages. One stage, prepared in the DSS, contributes to selecting only necessary tuples and attributes and to arranging an original relation for parallel processing. The next stage in RPS parallel environment is responsible for performing the postoperations on the grouped subsegment. The postoperations include duplicate elimination within the subsegment and concatenation operation between tuples within the subsegment of relation R and tuples within the subsegment of relation S. Both subsegments are filtered by the same hash function.

Therefore, SPIRIT-III can obtain the time performance $O(N/L)$ in performing Join and Projection operations (L = number of microprocessors and N = cardinality of a relation); whereas the first-generation RDBMs[3,4,5,8] required $O(N \times N)$ and the second-generation machine[7] realized $O(N \times N/L)$.

We now view several architectural approaches to RDBM and discuss them. The first-generation RDBM attempted to

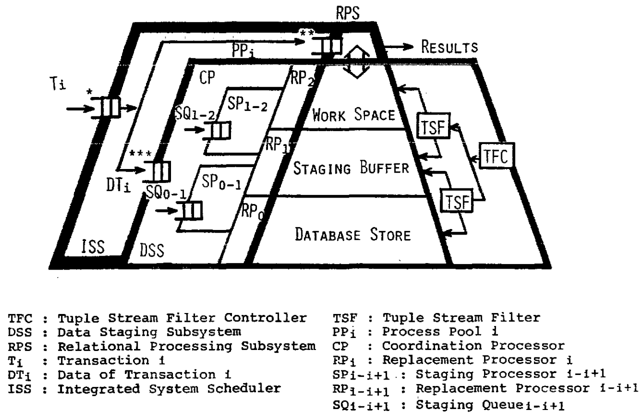


Figure 1—Conceptual system architecture of SPIRIT-III

directly process the stored relations with parallel architectures using microprocessors. Because of direct processing to original relations, which are generally unsorted, the first-generation RDBMs could not gain a desirable performance; and the time complexity of this type of machine was $O(N \times N/L)$ in executing Join and Projection operations. On the other hand, the second-generation RDBMs attempt to make it possible to process Join and Projection operations by a VLSI algorithm with $O(N)$. The basic concept of the second-generation RDBM is to employ two-step processing mechanisms. In the first step, a sort operation to an original relation is performed; in the next step, the sorted relations are handled. The first step, implemented by using a VLSI processor, can sort N-tuple relations by means of an algorithm with $O(N)$ time complexity. In the next step, Projection and Join operations on the sorted relations are essentially sequential algorithms and are out of harmony with parallel processing. This avoids introducing parallel architectures in the second step. SPIRIT-III is the third-generation RDBM, which copes with the limitation of the second-generation RDBM.

The proposed SPIRIT-III, which handles problems from the data staging architecture to the relational algebra execution architecture under the total concept, is the most practical and powerful RDBM based on the state of the art.

2. BASIC SYSTEM ARCHITECTURE OF SPIRIT-III

2.1 System Architecture

Figure 1 shows the conceptual system architecture of SPIRIT-III, which is composed of the three major subsystems: the integrated system scheduling subsystem (ISS), the data-staging subsystem (DSS), and the relational database processing subsystem (RPS).

In SPIRIT-III, a three-level memory hierarchy is introduced to enhance staging throughput of data[13]. The memory hierarchy consists of three types of memories, such as the primary work memory (PWM) for database microprocessors, the staging buffer (SB) as a disk cache, and the database store (DBS), composed of moving arm disks. SPIRIT-III incorporates refined preprocessing architecture, for efficient execution of relational operations, and advanced data-staging architecture into the memory hierarchy. These architectures cooperate to improve the throughput of data staging and the efficiency of set level operations.

ISS is responsible for scheduling the process execution in RPS and data staging in DSS according to the proposed policy,[10] called ISR, for Integrated process and data Scheduling in Relational database environments. The ISR policy integrates the data-driven process-scheduling policy in RPS environment and the look-ahead data-staging policy[8, 10] employed by DSS (described in Section 4). ISS also coordinates the actions of the other two subsystems. To be concrete, when a transaction arrives at the system, ISS registers the processes used by the transaction to the process pool and registers the data access request for the processes of the transaction to the data staging pool. DSS can schedule the sequence of data staging out of accordance with the predefined execution sequence of a transaction in normal mode. When all data required in executing a process are staged up to the memory of RPS, ISS changes the state of this process in the process pool to the executable state. Then, according to the policy of data-driven process scheduling, RPS executes the process under a multi-transactions environment. This scheduling strategy is based on the property of the high flexibility of the execution sequence of the relational algebra tree. The introduction of the ISR policy, minimizing the total data-staging cost, leads to the realization of a high-performance RDBM.

DSS optimally schedules data staging between each two memory hierarchies in order to minimize the total cost of data staging, which is defined as the cost of data access and transfer between storage devices in the memory hierarchy. DSS comprises three types of functional processors: the coordination processor (CP), the staging processors (SP), and the replacement processors (RP). These functional processors cooperate to schedule data staging in the memory hierarchy.

RPS is composed of multiple high-performance microprocessors. This subsystem can perform postoperations for relational algebra to each of the subsegments that has been already arranged for parallel processing in DSS. A typical postoperation is the link operation, which actually concatenates tuples within the subsegment in relation R and tuples within the subsegment in relation S. In this case, both the subsegments roughly are grouped through the grouping filter with the same hash function associated with joining attributes. The other postoperation of RPS is duplicate elimination within each subsegment which roughly is grouped through the grouping filter with the hash function for the attribute required by a Projection. Therefore, if different subsegments are assigned to different microprocessors in RPS, these microprocessors can simultaneously process the subsegment assigned to them without the overhead of interprocessor communication. This forms the ideal environment for parallel and asynchronous processing. This two-stage architecture for performing relational algebra can coordinate the data-staging strategy and can obtain the execution time complexity $O(N/L)$ in the case of Projection and Join, though the other RDBMs can realize $O(N \times N/L)$ execution time complexity.

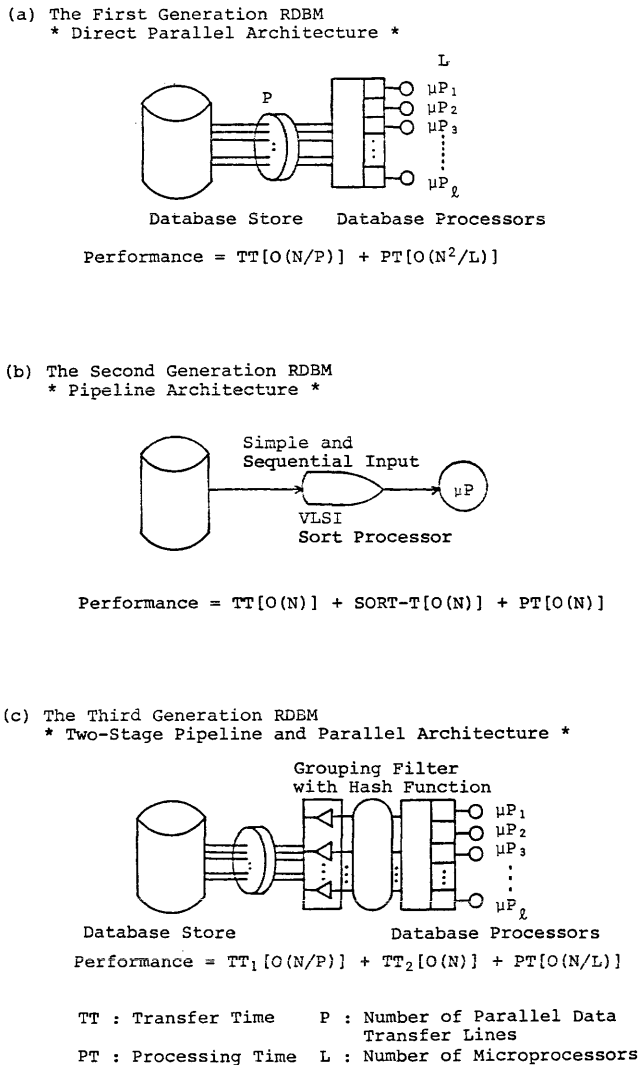


Figure 2—Comparison between SPIRIT-III architecture and other RDBM architectures

2.2 The Two-stage Execution Architecture of Relational Algebra

This section describes an advanced two-stage relational algebra execution architecture employed by SPIRIT-III. First of all, we discuss the following two architectures introduced by the conventional RDBMs.

1. *First-generation architecture.* The relational algebra execution architectures are employed by first-generation RDBMs such as DIRECT,[5] RAP,[3] RAP-2,[4] and SPIRIT-I.[8] The architectural feature is the direct execution of original and unordered relations. These architectures, implemented by a multimicroprocessor configuration, attempt to handle directly unordered relations in executing relational operations in parallel. This makes it possible to improve the execution time of search operations $O(N/L)$ by multiple processors. The search operation is the most primitive suboperation of relational algebra. Therefore, simple relational operations such as Selection and Restriction can be executed in $O(N/L)$ time. However, this type of architecture faces the difficulty of improving the execution time complexity $O(N \times N)$ of Join and Projection operations. Figure 2(a) shows the basic approach to processing relational operations supported by the early days' RDBMs.

2. *Second-generation architecture.* Second-generation architecture[7] uses the two-step process in executing relational algebra. In the first step, original relations that are read from the database store are searched and sorted by the dedicated VLSI processor. This processor, composed of many elementary cells, takes $O(N)$ time in performing the sort operation and selects the tuples that satisfy a given predicate. Next, in the second step, the primary database processor can handle Projection and Join operations to the presorted relations by the sequential algorithm with $O(N)$. Therefore, the best time performance that we can achieve for Projection and Join operations by means of this type of architecture is totally $O(N)$.

Figure 2(b) illustrates the basic concept and performance of the presorting method introduced by the second-generation RDBM. The performance of the second-generation RDBM is superior to that of the first-generation RDBM.

3. *Third-generation architecture.* SPIRIT-III is designed as a third-generation RDBM, which should enhance the capabilities of both relational algebra execution and data staging under an integrated concept. The real performance of the RDBM depends on the efficiency of relational operations and the throughput of data staging in the memory hierarchy.

SPIRIT-III employs two-stage architecture for executing relational algebra. Two-stage architecture is realized by using three major filtering functions. The filtering functions are implemented by the simple processor, called the Tuple Stream Filter (TSF). SPIRIT-III has three types of TSF. These TSFs are attached to all data lines connected between each two levels of the memory hierarchy.

Figure 2(c) shows the basic concept of manipulating relational algebra operations. The major difference between SPIRIT-III and the second-generation RDBM is the function adopted in the first stage. The second-generation RDBM adopted the sort function. However, this concept leads to being faced with the practical implementation issues of the dedicated VLSI sort processor with execution time performance $O(N)$. One of the implementation issues is the difficulty of developing a sort processor that can support the processing of any relation size, any relation cardinality, and any tuple length in the same fashion. The other is the inefficiency of postoperation to the presorted relation in Projection and Join, because the algorithm of the postoperation is essentially sequential and is not fit for parallel architecture. The third-generation RDBM, SPIRIT-III, is more advanced, because the architecture can support the processing of any size and cardinality of relation and any tuple length in the same manner and also manipulate the preprocessed relation in parallel without any overhead. Moreover, this architecture is very suitable for coordinating a data-staging subsystem introducing a memory hierarchy.

In this paper, we propose the powerful and flexible two-stage architecture, which uses the filtering functions embedded in each stage of memory hierarchy.

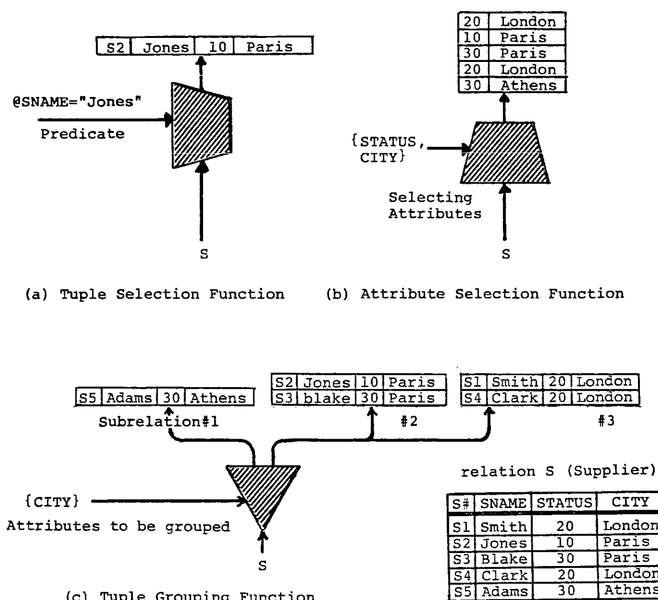


Figure 3—Functions of Tuple Stream Filters

The proposed three filtering functions, as shown in Figure 3, are the following:

1. The tuple selector function: This function is realized by the Selecting Tuple Filter (STF), as shown in Figure 3(a), which is one type of TSF. This function is the most primitive that selects the tuples that satisfy a given predicate. By means of this function, the most important relational operations, such as Selection and Restriction, can be completely achieved.
2. The attribute selector function: This function is realized by the Selecting Attribute Filter (SAF), as shown in Figure 3(b), which is one type of TSF. This function can filter the attributes needed by the next relational operations to be fired and repack each tuple, including items of only necessary attributes. This function performs a relational algebra operation, such as Projection, excluding duplicate elimination. This function contributes to saving work space and results in enhancing the performance of executing the relational algebra operations.
3. Segment-grouping function: This function is the key idea of SPIRIT-III architecture and is realized through the grouping filter, as shown in Figure 3(c). This grouping function is the most primitive suboperation of relational algebra operations, such as Projection and Join. This function groups input relations into subrelations. Next, in the case of Projection, each microprocessor of RPS performs a postoperation that is a complete duplicate elimination. RPS can process the assigned subrelation in parallel without the overhead of interprocessor communication. (The overhead of interprocessor communication degrades performance.) In the case of Join, each microprocessor of RPS handles the postoperation that is a concatenation of tuples in relation R and tuples in relation S. RPS can perform this link operation between tuples in subrelation R_i and tuples in subrelation S_i, both of which are grouped through the same filtering function with each of the joining attributes, because this grouping function guarantees that both subrelations contain same value items of joining attributes. Tuple grouping filters, realizing this function with a refined method of hash function, are appended to all data transfer lines linked between each stage of the memory hierarchy system in SPIRIT-III. Therefore, if all data need this preprocess, the data are grouped through the filter in passing on data lines in the memory hierarchy before the data are staged up to primary database microprocessors.

Without any overheads resulting in poor performance, SPIRIT-III, which can efficiently process set level relation operations, is the most powerful and practical machine designed on the basis of the state of the art.

3. RELATIONAL ALGEBRA EXECUTION ARCHITECTURE

3.1 Tuple Stream Filter

We have designed the following functional Tuple Stream Filters, as shown in Figure 3, which should be incorporated in the DSS.

1. The Selecting Tuple Filter (STF): This filter functions to yield a "horizontal" subset of a given relation, that is, a subset of tuples within input relation that satisfy a specified predicate. Figure 3(a) illustrates the symbol and function of the selecting tuple filter. This filter can perform the same relational algebraic operations such as Selection and Restriction, when tuples stream through STF.
2. The Selecting Attribute Filter (SAF): This filter functions to yield a "vertical" subset of input relation, that is, that subset obtained by selecting specified attributes, provided that the filter does not eliminate duplicate tuples within attributes selected. This filter can perform to repack items of specified attributes in a tuple into a new tuple without duplicate elimination. This filter is shown in Figure 3(b).
3. Tuple Grouping Filter (TGF): This filter functions to yield subrelations (subsegments), each of which is a subset of an input relation, which consists of tuples including the same specified attribute item. In practice, TGF is implemented by using the hash mechanism in SPIRIT-III. Figure 3(c) displays the symbol and function of TGF. Figure 4 shows a two-stage relational architecture using tuple stream filters embedded in the DSS.

3.2 Relational Algebra Processing

In this section, we explain processing methods of typical relational algebra operations.

1. Selection processing: Selection operation is the simplest but most important operation of relational algebra. In

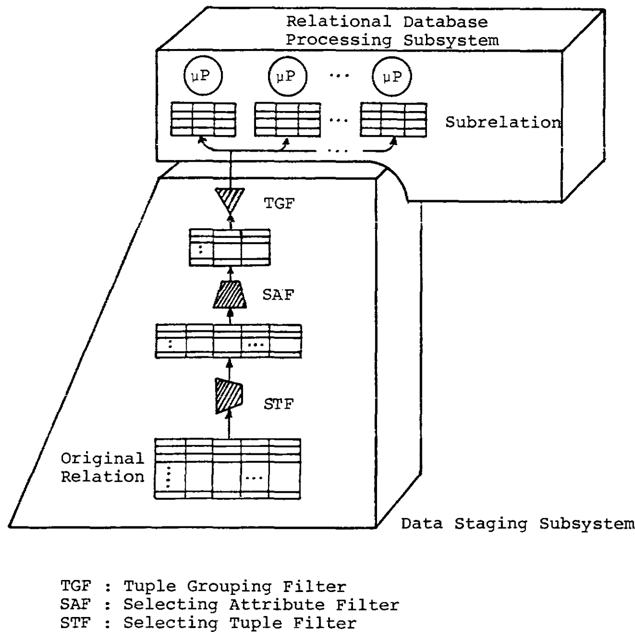


Figure 4—Two-stage relational algebra execution architecture with Tuple Stream Filter

SPIRIT-III, most Selection operations are performed while tuples are passed through TSFs attached on data channels in the memory hierarchy. This means that the execution time of Selection is actually overlapped with the data transfer time. Therefore, in SPIRIT-III, it is not necessary to consider Selection processing time in order to measure the total performance, because most Selections are performed while relations stream on data channels. Restriction is performed on the basis of the same method.

2. Projection processing: The Projection operator contains two major operations, which are to select specified attributes or a given relation and then to eliminate duplicate tuples within attributes selected. The first operation is completely performed through ASFs. Therefore, the processing time for the first operation of most Projections is overlapped with the data-staging time. TGF is responsible for performing the function to partition a relation into subrelations. TGFs distributed in DSS can roughly group an input relation into subrelations, using the hash mechanism, while original relations are passed through TGFs. Then each microprocessor for postoperation in RPS can eliminate completely duplicate tuples within the grouped subrelation assigned to it. In this case, the processing time associated with the hash function can be made very short by using specialized hardware.[6] Moreover, if TGF groups the relation composed of N tuples and K specific values into M subsegments, and the number of microprocessors in RPS is L, each grouped subsegment j contains (N/M) tuples and a few $(K_j = K/M)$ specific values. This makes each microprocessor execute the grouped subsegment assigned by the algorithm with time complexity $O\ (N_j(K_j + 1)/2)$, provided that N_j is the number of tuples within the grouped subsegment and K_j is the number of unique items in subsegment j. The best time performance of Projection that SPIRIT-III can achieve is near $O\ (N_j = N/M)$. This two-stage mechanism succeeds in gaining the time performance $O\ (N_j = N/M)$ for most Projections in principle. This performance is extremely superior to the performance $O\ (N \times (K + 1)/(2 \times L))$ achieved by early RDBMs and the performance $O\ (N)$ by the second-generation RDBM.

3. Join processing: Join is the key operation guaranteeing the flexibility and powerful capability of the relational model, but it is the most important operation that must be solved by the system designer. SPIRIT-III can perform the Join operation using the same mechanism for Projection. We can explain the equi-Join processing mechanism by giving an example: Relation R and relation S stage up from database store to staging buffer; Relations R and S are grouped into Subrelations $\{R_j\}$, $\{S_j\}$ through TGF over joining attributes R.A and S.A. Therefore, subrelations R_j and S_j consist of tuples containing the same items within the joining attributes R.A and S.A. Because the processing time associated with this grouping function is overlapped with the data-staging time, the processing time to group a relation into subrelations through TGF can be ignored. This advantage results in reducing the processing time of Join. Moreover, in the next stage, SPIRIT-III employs the subsegment allocation strategy that both the grouped subrelations R_j and S_j are assigned to the same microprocessor of RPS. In case of equi-Join, each microprocessor can perform the actual concatenation of tuples between the grouped subsegments R_j and S_j without the access to other subrelations R_j $(i \neq j)$. This capability can make possible an ideal parallel, asynchronous processing environment, which contributes to a remarkable enhancement of the time performance of postoperations associated with Join in the second stage. In order to evaluate the time complexity of Join processing in the SPIRIT-III environment, we assume the following:

   1. Relations R and S contain N_r and N_s tuples, respectively.
   2. TGF groups them into M subrelations by each of the joining attributes R.A and S.A.
   3. Each subrelation is exhibited as R_j and S_j, respectively.
   4. R_j and S_j include a few unique items within each joining attribute that are exhibited as K_{rj} and K_{sj}, respectively.

In case of equi-Join processing, the processing time required in the first stage in DSS is actually hidden within data transfer time in the memory hierarchy. In the second stage, if both R_j and S_j subrelations are assigned to the work space of the same microprocessor in RPS, each microprocessor can concatenate between tuples of R_j and tuples of S_j in parallel without the overhead of interprocessor communication. In this case, the processing time complexity of the second stage in RPS is $O\ ((N_{sj}\ (K_{sj+1})K_{rj})/2)$ or $O\ ((N_{rj}\ (K_{rj+1})\ K_{sj})/2)$. In

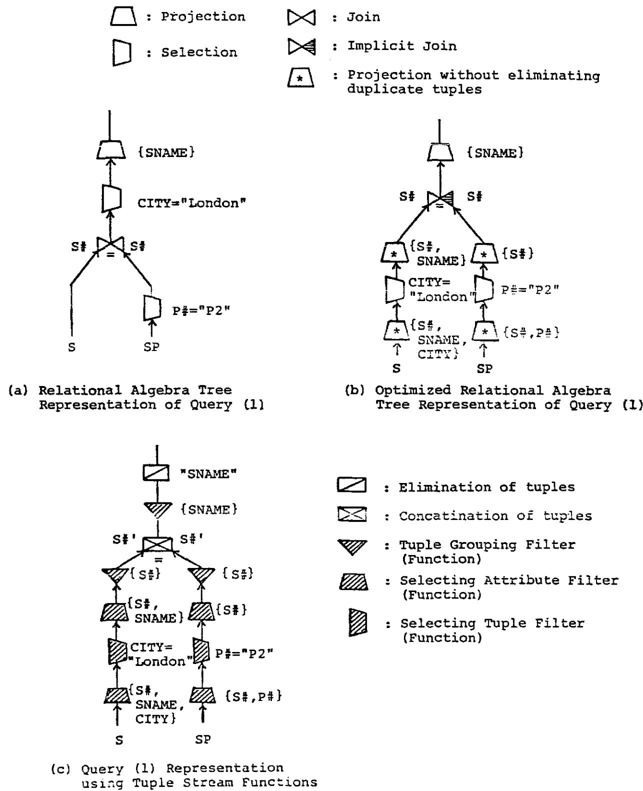


Figure 5—Representation of relational algebra execution sequence in SPIRIT-III environment

practice, this time complexity is nearly O $(N_{rj} \times u)$ or O $(N_{rj} \times v)$ because K_{rj} and K_{sj} are much less than N_{rj} and N_{sj}, respectively.

This means that SPIRIT-III succeeds in executing equi-Join with time complexity O (Min $(N_{rj} = N_{rj}/K, N_{sj} = K_{sj}/K)$. The relational algebra architecture employed by SPIRIT-III is very superior to the first-generation architecture of time complexity O $(N \times N/L)$ and the second-generation architecture of time complexity O (N).

3.3 An Optimized Relational Algebra Execution Mechanism Using Tuple Stream Filters

In this section we explain an optimized relational algebra tree execution mechanism utilizing Tuple Stream Filters attached on data transfer channels in the memory hierarchy. We describe the proposed two-stage relational algebra execution architecture in detail, using the following query as an example.

Query 1: "Get supplier names for suppliers in London who supply part P2"

It is assumed that a sample relational model consists of two relations: S (the SUPPLIER relation, composed of four attributes: S#, SNAME, STATUS and CITY) and SP (the SUPPLIER-PART relation, composed of three attributes: S#, P#, and QTY).

In this case, Query 1 may be expressed in SEQUEL as follows.

```
SELECT SNAME
FROM S
WHERE CITY = "London" and S# IS IN
  ( SELECT S#
    FROM SP
    WHERE P# = "P2" )
```

The SEQUEL compiler directly translates Query 1 into the relational algebra tree representation, as shown in Figure 5a. However, the relational algebra tree representation shown in Figure 5b is not always the most efficient execution sequence. We can translate this relational algebra tree into the more efficient execution sequence tree, according to the optimization policy proposed by Smith.[2] Figure 5b shows the optimized relational algebra tree from the query of Figure 5a. It is beneficial to move Selection operations as far down the tree as possible using such transformations. This is because the Selection operations reduce the number of tuples to be processed by subsequent operations. Any reduction is particularly advantageous when Join and Projection operations occur later. There are also benefits to be gained by moving Projection operations down a query tree. Projection operations decrease the width of tuples and, due to the elimination of duplicate tuples, may also decrease the number of tuples in a relation. Each of the new Projection operations perform selecting attributes needed by subsequent operations and does not eliminate the duplicate tuples input relation. Therefore, this transformation remarkably increases the efficiency of executing Query 1.

Next, the optimized relational algebra tree of the Query 1, as shown in Figure 5b, is smoothly translated into the execution procedure with the tuple stream filters of SPIRIT-III, as shown in Figure 5c.

The relational algebra execution architecture of SPIRIT-III harmonizes with the execution sequence expressed by the optimized relational algebra tree and can powerfully support this optimization concept.

3.4 Implementation Architecture

Figure 6 illustrates an implementation architecture of the proposed SPIRIT-III. SPIRIT-III employs large-capacity moving head disks with a track-in-parallel read/write mechanism as database stores, and a number of VLSI random-access memory chips as a staging buffer, which is used as a disk cache and tuple clustering memory.

This track-in-parallel read/write mechanism makes it possible to remarkably reduce the data transfer time between SB and DBS and to efficiently filter the relations that stream on parallel transfer lines.

Double-loop configuration is introduced as the interconnection of banks of SB and microprocessors of RPS. Each of the grouped subrelations in each bank of SB is allocated to a corresponding microprocessor through the double loops.

Figure 7 illustrates the detailed interconnection of the staging buffer and the Tuple Stream Filter. The Tuple Stream Filter Controller (TFC) issues the filtering conditions to each

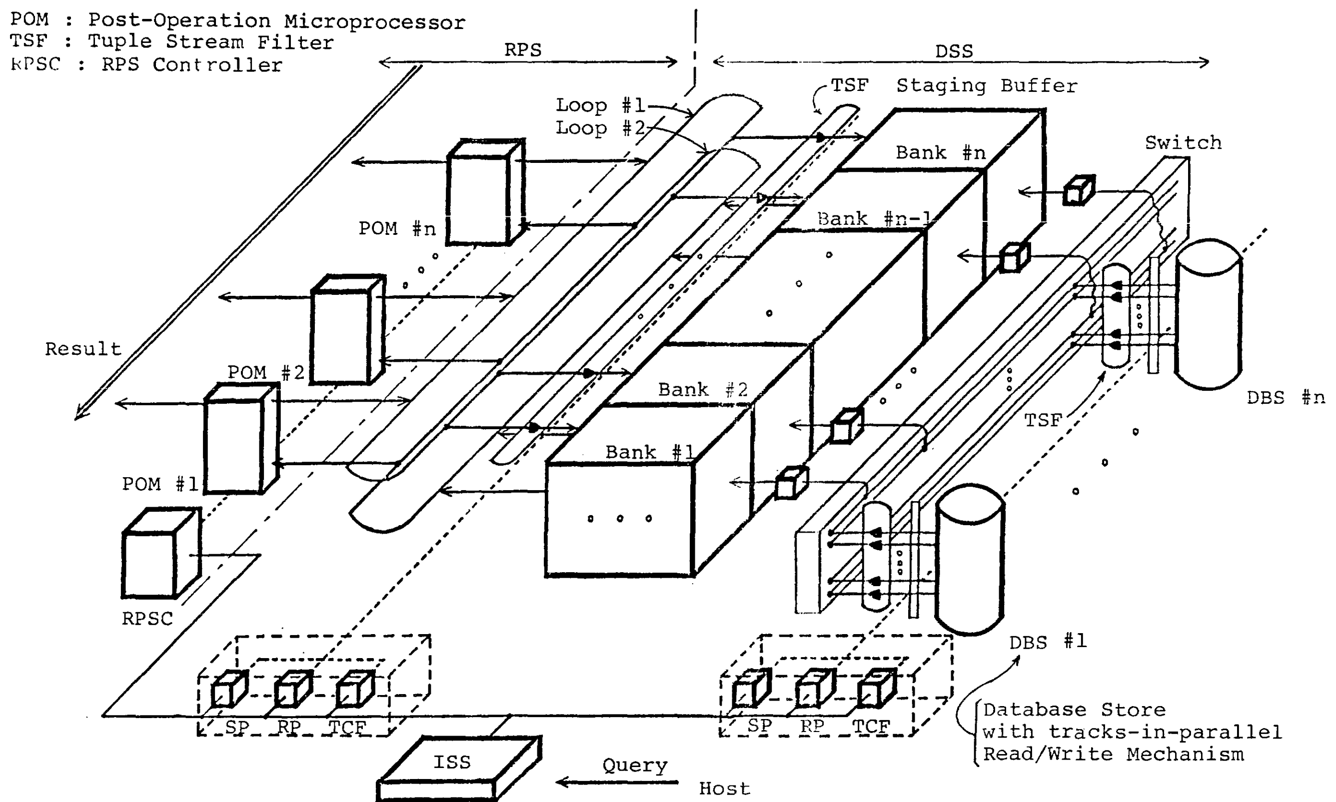


Figure 6—System architecture of SPIRIT-III

Tuple Stream Filter. TGF receives the grouping function and attribute IDs to be grouped.

The retrieval conditions of Selection and Restriction are set to the STF by the TFC, and Selecting attribute IDs are sent to the ASF.

As shown in Figure 6, TGF appends the subsegment number to the tuple when the tuple is passed through TGF, and then the staging buffer management processor actually clusters the tuples, using the attached subsegment number.

4. DATA STAGING ARCHITECTURE

4.1 Basic Concept and Implementation of ISR Policy

The look-ahead data staging policy depends on the following three properties of relational database environments:

1. Possibility of complete recognition of all data used by a transaction: The expression of query is nonprocedural, and the relations that are used by a transaction are declared explicitly.
2. Homogeneity of pages in a relation: Based on set theory, each page in a relation is operated equivalently.
3. High flexibility of execution sequence: Execution sequence of each leaf of a query that is expressed in the relational algebra tree has little restriction.

From properties 1 and 3, DSS can schedule data staging with the original strategy, taking physical environment such as storage device characteristics into consideration. Property 2 guarantees that there is no useless data transfer.

The example transaction shown in Figure 8 helps to explain the detail of the look-ahead data staging and the data-driven process scheduling.

The cost of a seek operation is the most expensive cost in RDBM that employs a moving head disk with the tracks-in-parallel read/write mechanism as DBS. In a conventional system, data staging is performed according to the execution sequence of a transaction: thus in this example, the data staging is performed in the order of A, B, C, and D, out of accordance with the data arrangement on the disk. As a result, the cost of data staging is high.

On the other hand, RDBM employing the look-ahead data-staging mechanism performs the data staging with the strategy of minimizing data access cost: thus in this example, data staging is performed in the order D, A, B, and C, according to the data arrangement; and as a result, the cost of the data staging is minimized.

Because of data-driven process scheduling, P_{i1} becomes executable when both A and B are staged up to the processor memory, P_{i2} becomes executable when C is created by P_{i1}, P_{i3} becomes executable when F is created by P_{i2}, and P_{i4} becomes executable when G is created by P_{i3} because D has already been staged up. Finally, P_{i5} becomes executable when H is created by P_{i4}. Staged up by the look-ahead data staging

from Relational Processing Subsystem

Staging Buffer Management Processor

Attributes to be hashed

TSF

TGF

SAF

STF

TFC

Selecting Attribute

Retrieval Condition

Bank #1

#2

#n

Free Space

Hash Address Entry Table

Nil

Staging Buffer

Database Store with tracks-in-parallel Read/Write Mechanism

TSF : Tuple Stream Filter
TFC : Tuple Filter Controller
TGF : Tuple Grouping Filter
SAF : Selecting Attribute Filter
STF : Selecting Tuple Filter

Figure 7—Construction of staging buffer with Tuple Stream Filter

mechanism, D stays until P_{i4} is completed by the processing subsystem. At peak activity, data like D increase in each level of the memory hierarchy. If staging based on the same strategy is continued, the processor memory will be filled with only this data. In this situation, the data replacement in the processor memory is needed in order to continue the execution of the processor even if the data replacement causes overhead. To avoid the overhead, it is necessary to change the strategy. To realize this, DSS must distinguish the data required by an executable process (active data) from the data required by an unexecutable process (inactive data).

The DSS takes the following two actions:

1. Sort the staging requests in staging classes.
   - SC1: Class of stage up (SU)-request issued by active data
   - SC2: Class of SU-request issued by inactive data
   - SC3: Class of stage down (SD)-request
2. Classify the memory blocks in MH_{i+1} into replacement states based on replacement cost.
   - RS1: State of block occupied by data issuing SC1 request
   - RS2: State of block occupied by data issuing SC2 request and having no copy in MH_i
   - RS3: State of block occupied by data issuing SC3 request
   - RS4: State of block occupied by data issuing SC2 request and having its copy in MH_i
   - RS5: Otherwise

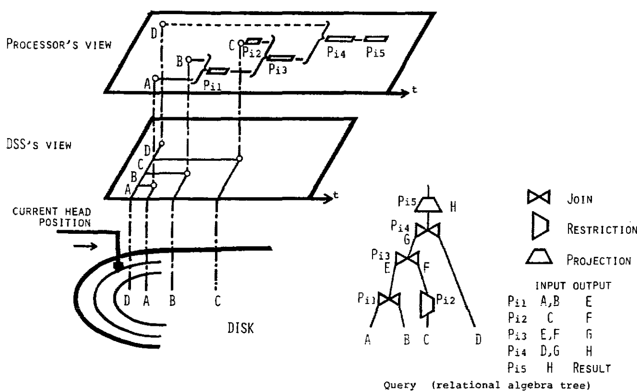


Figure 8—Outline of look-ahead data staging and data-driven process scheduling

Utilizing this information and mechanism, data stagings between each level can be executed synchronously. In addition, the optimal scheduling, taking physical environment into consideration, is performed at each level. Thus, the look-ahead data-staging policy attains high performance of data staging in the relational database environment.

To achieve the maximum advantage of data staging, the processing subsystem should use data-driven process scheduling in addition to DSS using the look-ahead staging mechanism.

4.2 Architecture of a Look-ahead Data Staging Subsystem

The data staging subsystem based on the ISR policy is different from the conventional memory management system in regard to management policy of data transfer. In the conventional memory management system, every reference by process is directed to the processor memory and causes data transfer among levels of memory hierarchy if the data do not exist in the processor memory; therefore the degree of freedom in scheduling the sequence of data transfer is strongly restricted. On the other hand, in DSS, all data transfers are scheduled by the independent strategy in each level of memory hierarchy; therefore, to enhance the capability of data staging, it is necessary to construct DSS that can asynchronously perform data staging between each level. As a result, DSS is composed of multiple processors, as shown in Figure 1.

(1) The Coordination Processor (CP)

The coordination processor (CP) functions as the interface to ISS and coordinates the processors that schedule the sequence of data staging between levels of the memory hierarchy. For example, the CP registers stage up-requests issued by data used by the transaction in the data-staging pool into relevant staging queues as a transaction arrives at the system. To accomplish this operation, the CP requires all RPs to search requested data in the memory hierarchy. The coordination processor also registers stage up-requests (or stage down-requests) into the staging queue of higher (or lower) levels when SP completes the data staging.

(2) The Staging Processor (SP)

The staging processor $SP_{(i,i+1)}$ manages data staging between MH_i and MH_{i+1}, using the staging queue $SQ_{(i,i+1)}$. All staging requests between MH_i and $MH_{(i+1)}$ are put into $SQ_{(i,i+1)}$ by CP. The $SP_{(i,i+1)}$ selects the staging request to be served according to the algorithm.[10] After this operation, $SP_{(i,i+1)}$ requires RP_i (or $RP_{(i+1)}$) to select the blocks to be replaced. The required scheduling policy for each SP is different from other SPs. Generally, the functions of SP fully depends on the storage devices of memory hierarchy.

The operations of a moving head disk consist of Seek, Search, and Transfer operations. The systems employing a moving head disk with a tracks-in-parallel read/write mechanism tend to increase the effect of seek time on total I/O processing time. Therefore, MINIMUM ACCESS COST DATA of the $SP_{(0,1)}$ in the algorithm[10] is selected by using the disk-scheduling algorithm that could minimize seek time.

In most conventional systems, the FCFS (first come, first served) policy is employed as the disk scheduling policy however, to enhance the efficiency of the Seek operation, SSTF (shortest seek time first), SCAN, and CSCAN (circular SCAN) policies have been developed and proposed. The important nature of disk scheduling policy, except the FCFS policy, is that as the number of I/O requests increases, so does the efficiency of the operation.

This suggests that DSS employing the look-ahead data-staging mechanism pulls out the maximum effect of disk-scheduling policy, except the FCFS policy, because all SU requests of transaction data are registered en bloc to the data-staging pool when a transaction enters RDBM.

(3) The Replacement Processor (RP)

The replacement processor for the staging buffer manages by using the memory map table (MMT). The memory map table contains the information of every block in every blank of the staging buffer: the identifier and the replacement state of the data stored in the block. When $SP_{(i,i+1)}$ (or $SP_{(i-1,i)}$) requests RP_i to select the block to be replaced, RP_i searches the block to be replaced in the order of blocks in RS5, RS4, RS3, and RS2. If there are only RS1 blocks, RP keeps $SP_{(i,i+1)}$ (or $SP_{(i-1,i)}$) waiting until any block becomes a replaceable state.

Our performance evaluation[10] by computer simulations shows that the proposed architecture improves both system throughput and response time of transaction.

5. CONCLUDING REMARKS

This paper has proposed an advanced architecture of the relational database machine, named SPIRIT-III, which is basically organized into a three-level memory hierarchy with sophisticated data-staging architecture and preprocessing architecture for relational algebra. The memory hierarchy system is composed of work space of primary database processors, a staging buffer as the intelligent disk cache, and moving arm disks as the database store.

SPIRIT-III aims at totally improving both I/O and CPU processing boundaries, which are key problems in implementing RDBMs; it has two major architectural features. One is the introduction of the relational-database-oriented data-staging mechanism, called the look-ahead data-staging mechanism, which can optimally schedule data access and transfer in the memory hierarchy. The other is that it attaches refined preprocessing mechanisms for relational algebra operations to data transfer lines connected between each two levels of memory hierarchy. When a relation stages up or down in the memory hierarchy, this preprocessing mechanism can function to select tuples and attributes and to rearrange original relations for parallel processing. SPIRIT-III provides three basic preprocessing functions. Two of these are the tuple selector, filtering only tuples to match retrieval conditions; and the attribute selector, repacking only attributes needed by executing a query. In addition to these filtering functions, SPIRIT-III incorporates the grouping filter proposed by the

authors into each stage of the memory hierarchy. The operation of the grouping filter is the primitive preprocessing operation for executing Join and Projection. In practice, the grouping filter implemented with hash function rearranges original relations and decomposes these into subsegments. Without the overhead of interprocessor communication, each database microprocessor can execute postoperations of relational algebra to a few subsegments assigned to it in parallel. The postoperations include duplicate elimination within the subsegment and concatenation operation between tuples within the subsegment in relation R and the tuples within the subsegment in relation S. Each subsegment is clustered by the same hash function while the original relation stages up.

Therefore, SPIRIT-III can smoothly perform Join and Projection operations by $O(N/L)$, whereas the early RDBMs required $O(N \times N/L)$. The proposed SPIRIT-III covering from data-staging architecture to relational algebra execution architecture under the integrated concept is one of the most practical and powerful RDBMs based on the state of the art.

ACKNOWLEDGMENTS

The authors gratefully acknowledge Professor H. Aiso of Keio University for his encouragement and valuable advice. They also thank members of the Mind group of Keio University for their valuable discussion and members of Ichikawa Laboratory of Hiroshima University for their encouragement.

REFERENCES

1. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM,* 13 (1970), pp. 377–397.
2. Smith, T. M., and P. Y. Chang. "Optimizing the Performance of a Relational Algebra Database Interface." *Communications of the ACM,* 18 (1975), pp. 568–579.
3. Schuster, S. A., F. A. Ozakarahan, and K. C. Smith. "RAP—An Associative Processor for Database Management." AFIPS Proc. NCC, 44, (1975), pp. 379–388.
4. Schuster, S. A., H. B. Nguyen, F. A. Ozakarahan, and K. C. Smith. "RAP.2—An Associative Processor for Database and Its Applications," IEEE Transactions on Computers, C-28 (1979), pp. 446–458.
5. Dewitt, D. J. "DIRECT—A Multiprocessor Organization for Supporting Relational Database Management Systems." *IEEE Transactions on Computers,* C-28 (1979), pp. 395–406.
6. Bobb, E. "Implementing a Relational Database by Means of Specialized Hardware." *ACM Transactions on Database Systems,* 4 (1979), pp. 1–29.
7. Tanaka, Y., Y. Nazuka, and A. Masuyama. "Pipeline Searching and Sorting Modules as Components of a Data Flow Database Computer." *Proceedings of the IFIP-80,* pp. 427–432.
8. Kamibayashi, N., H. Kato, Y. Kiyoki, H. Ozawa, K. Seo, and H. Aiso. "SPIRIT: A New Relational Database Computer Employing Functional-Distributed Multi-Microprocessor Configuration." *Proceedings of the First International Conference on Distributed Computing Systems,* (1979), pp. 757–771.
9. Kiyoki, Y., N. Kamibayashi, K. Tanaka, and H. Aiso. "Design and Evaluation of a Relational Database Machine Employing Advanced Data Structures and Algorithms." *Proceedings of the 8th International Symposium on Computer Architecture,* 1981, pp. 407–423.
10. Seo, K., N. Kamibayashi, A. Minematsu, and H. Aiso. "A Look-Ahead Data Staging Architecture for Relational Database Machines." *Proceedings of the 8th International Symposium on Computer Architecture,* 1981, pp. 389–406.
11. Teorey, T. J., and T. B. Pinkerton. "A Comparative Analysis of Disk Scheduling Policies." *Communications of the ACM,* 15 (1972), pp. 177–184.
12. Brice, R. S., and S. W. Sherman. "An Extension of the Performance of a database manager in a virtual memory system using partially locked virtual buffers." *ACM Transactions on Database Systems,* 2 (1977), pp. 196–207.
13. Law, S. Y., and S. E. Madnick. "Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data." *ACM Transactions on Database Systems,* 4 (1979), pp. 345–367.

Data language requirements of database machines

by DAWEI LUO
Harbin Institutes of Technology
Harbin, China
DAOZHONG XIA
Shanghai Shipbuilding Technology Research Institute
Shanghai, China
and
S. BING YAO
University of Maryland
College Park, Maryland

ABSTRACT

The selection of a proper set of operations for a database machine has a great influence on the performance of the overall system. In this paper, we will discuss the data language requirements of database machines. The data language for a proposed relational database machine is presented. Methods for supporting hierarchical and network interfaces using a relational database machine language set are introduced. The relative performance for data access using relational and network interfaces is compared through the evaluation of cost functions. It is shown that the support of a network interface in a relational database machine is feasible, but expensive.

1. INTRODUCTION

The development and application of database technology has made a great impact on data processing. It has made possible the sharing of common data resources and relieved the end user's burden of managing the data. The data independence provided by many database systems has made the maintenance of program and data more effective.

The advantages of database systems have encouraged the growth of the size and complexity of many database system applications. In many applications the efficiency and reliability of the system are becoming problems. A database management system is a fairly complex piece of software. It competes with other application programs for memory and computing resources. It also requires a certain effort for maintenance.

Database machines have been proposed as hardware solutions to these problems. The objective is to offload database management functions from the host computer to a specially designed, dedicated back-end computer whose sole function is to maintain the database and process database requests. A database machine interfaces with the front-end host computer through a high-level-language interface. By fixing such an interface, a database machine can communicate simultaneously to multiple host computers. Depending on the size and complexity of a system, the host computer could be a mini- or a microcomputer, or even, in some situations, an intelligent terminal.

To permit a database machine to communicate with various types of front-end processors, it is important to select a proper language interface. The front-end processors may vary in processing capabilities, type of applications, or even data models. This paper reports the development of a database machine language interface intended to support multiple types of processing effectively.

The end user interface in the host computer usually consists of a terminal handler, a communication processor, and a language parser and translator. The major function of the host computer is to translate end user queries into the database machine language, send it to the database machine, receive results from the database machine, and finally organize and display the result. Depending on the application, it is generally accepted that user interfaces to database systems can be functionally classified into three categories:

1. *Self-contained query language.* The end user requests are formulated directly in a query language and are translated into the database machine language for further processing.
2. *Predefined commands.* Certain frequently processed transactions can be predefined and stored. The end user simply activates a stored command and supplies a few required parameters. The predefined command can be stored in the host computer or in the database machine in an internal form. The advantage of this approach is that a stored command can be compiled only once and is therefore more efficient. However, this gives the database machine an additional burden of recompiling the stored commands when the access paths are changed by database reorganization.
3. *Embedded query language.* Transactions written in a host language, such as FORTRAN or COBOL, may access the database by using embedded queries. A language preprocessor replaces the embedded queries with procedure calls to a run-time system, which sends the embedded queries to the database machine during transaction execution. Alternatively, the embedded queries can be preprocessed and stored in the database machine as predefined commands. During execution the transaction simply activates appropriate stored commands through calls to the run-time system. The tradeoff between these two approaches is similar to that of the stored commands.

It is clear that the self-contained query language interface is the most fundamental, and it is the focus of this paper. Our design for the database machine language interface is based on a previous experience of implementing a SEQUEL (SQL) interface[2,9] for a rational database machine.[6,7] Although both the query language and the database machine are based on the relational data model, discrepancies in their detailed operations have made the implementation nontrivial. Next section discusses the general database machine language requirements for effective query languages implementation. The two sections following the next section consider issues for the implementation of hierarchical and network interfaces for database machines.

2. DATABASE MACHINE LANGUAGE REQUIREMENTS

Database applications have many varieties. They differ in type of operations and data models. A database machine cannot possibly support all these access requirements directly. It is only feasible to provide a basic set of operations from which other operations can be derived. To select a proper set of operations, several requirements must be satisfied:

1. *Completeness.* The set of operations should be, in some sense, complete. This means that all the operations performed on the database can be derived from the basic set of operations in a straightforward manner. Otherwise,

some database operations will have to be performed in the host processor, thus making the system less effective.

Codd[3] has shown that relational algebra is equivalent to relational calculus and has defined the notion of relational completeness. A query language is said to be complete if it can perform all the relational calculus operations. The completeness discussed here is not limited to the relational completeness. By completeness we mean the ability to perform all database operations effectively. Such completeness property is difficult to define formally, as data models and data languages are still continuously being developed. New data models and data languages are constantly being proposed, and it is difficult to make a list of all possible database operations. On the other hand, it seems reasonable to base the completeness requirements on the ability of performing all operations found in present database systems.

2. *Efficiency and flexibility*. Although most of database operations can be supported by a minimum set of basic operations, the performance of some of the operations can be greatly improved if a slightly larger set of basic operations is used. Appendix A provides some examples to illustrate the relative inefficiency of processing some derived operations by using a small set of basic operations. Some of these derived operations can be very easily implemented in the database machine.
3. *High-level operations*. The objectives of using a high-level-language interface are to suppress the irrelevant details of storage and access path information. This is appropriate, since the storage and access paths aspects are handled exclusively by the database machine. There are two principal advantages: (a) A high-level interface makes the processing in the front-end processor much simpler; (b) A high-level interface minimizes the number of communications between the front-end processor and the database machine and thus reduces the communication overhead.
4. *Extensibility*. Database machines should provide facilities for defining new operations. The new operation can be defined in terms of existing ones and included in the database machine as a *stored command*. These stored commands can be defined by the front-end processing system or by the end user.

A database machine language could be defined on the basis of a particular data model, such as the relational, hierarchical, or network model. In view of the above requirements, we have chosen to base the proposed database machine language on the relational data model. The advantage of the relational model is its simplicity and high level of operation. In fact, it is not surprising to see that most of the recently developed database machines are based on relational data models.

Typical relational data languages, however, do not have sufficient operations to implement the operations of other data models efficiently. Our design enhances the basic relational interface by additional operations that we feel are necessary in order to build an efficient interface for multiple data models. The proposed set of operations is given in Appendix B. We have investigated the problem of supporting several important languages, including the following:[1,4,8,10,11,12] (1) Relational: QUEL, SQL, QBE; (2) hierarchical: DL/1 (IMS), FQL; and (3) network: DML (DBTG).

Table I shows the database management language requirements for these query languages.

Table I—Database management language requirements for various query languages

operation	operand	result	QUEL	SQL	QBE	DL/I	FQL	DBTG
1) project	relation	relation	V	V	V	V	V	V
2) select	relation	relation	V	V	V	V	V	V
3) order	relation	relation	V	V	V	V	V	V
4) group	relation	relation	V	V	V	V	V	V
5) unique	relation	relation	V	V	V	V	V	V
6) select get next	relation	tuple		V		V	V	V
7) union	two relations	relation		V	V		V	
8) subtract	"	"		V	V		V	
9) intersect	"	"		V	V		V	
10) join	"	"	V	V	V		V	
11) join get next	"	tuple				V		V
12) aggregate function	relation	value	V	V	V	V	V	V
13) delete	relation	relation	V	V	V	V	V	V
14) insert	relation, tuple	relation	V	V	V	V	V	V
15) update	relation	relation	V	V	V	V	V	V
16) assign	relation	relation	V	V	V		V	
the following operations are for expanding select condition:								
1) contains	two relations	boolean		V	V		V	
2) does not contain	"	"		V	V		V	
3) equal	"	"		V	V		V	
4) in	tuple, relation	boolean		V	V		V	
5) not in	"	"		V	V		V	

3. THE DATA LANGUAGE FOR A HIERARCHICAL DATABASE

Two problems must be addressed when developing a hierarchical data language for a relational database machine. The first is to transform a hierarchical data structure into relations; the other is to transform the operations on the data into the basic operations. We will illustrate these transformations by using as an example a hierarchical data language.

FQL is a form language for the manipulation of data stored in a hierarchical database.[8] A hierarchical file containing nested *repeating groups* can be represented by a *form*, which is also called an unnormalized relation. More precisely, a form can be defined as follows:

$$F = R(K, A, \{F_i\})$$

where
K is the set of key attributes
A is the set of non-key attributes
$F_i = R_i\ (K_i, A_i,\ \{R_{ij}\})$ are repeating groups which are also forms, defined recursively

Example 1.
A DEPARTMENT form can be illustrated as in Figure 1.
The hierarchical schema given on the top of the form can also be represented as follows:

DEPARTMENT ($\underline{\text{DD\#}}$,DNAME,CLERK ($\underline{\text{CE\#}}$,CAGE),
ENGINEER ($\underline{\text{EE\#}}$,EAGE,
EXPERIENCE ($\underline{\text{ED\#}}$,DATE)))

where DD# is the key attribute of the form and CLERK, ENGINEER, and EXPERIENCE are repeating groups.

DEPARTMENT							
		CLERK		ENGINEER			
						EXPERIENCE	
DD#	DNAME	CE #	CAGE	EE#	EAGE	ED#	DATE
1	Car	101	25	110	25	1	1978
		102	28			2	1979
				115	35	3	1977
		103	35			2	1979
2	Ship	201	25	120	30	2	1978
						1	1979
		202	30	125	35	1	1976
						2	1978
		203	34			1	1979

Figure 1—A department form

In order to provide an FQL interface to a relational database machine, it is necessary to represent forms as relations. The following algorithm to perform this transformation is well known:

Algorithm 1 (Codd[3])
Decomposition of a form into relations.
INPUT: A form $F = R(K, A, \{F_i\})$
OUTPUT: A set of equivalent relations $R,\ R_i,\ R_{ij}, \ldots$
METHOD: For each repeating group, define a relation consisting of the repeating group's key and nonkey attributes and the key attributes of all its parent repeating groups. Therefore,

for F we have $R(\underline{K}, A)$,
for F_i we have $R_i\ (\underline{K, K_i}, A_i)$,

and in general, for $R_{ij \ldots k}$ we have

$R_{ij \ldots k}\ (\underline{K_i, K_{ij}, \ldots, K_{ij \ldots k}}, A_{ij \ldots k})$

Example 2
The form in Example 1 can be decomposed into the following four relations:

DEPARTMENT($\underline{\text{DD\#}}$, DNAME)
CLERK ($\underline{\text{DD\#,CE\#}}$, CAGE)
ENGINEER($\underline{\text{DD\#,EE\#}}$, EAGE)
EXPERIENCE($\underline{\text{DD\#,EE\#,ED\#}}$, DATE)

The extensions of the relations are given in the Figure 2.
We will now show the transformation of an FQL query to relational queries. An FQL query can be defined as follows:

SELECT $P_1, \ldots, P_k$
FROM $F = R\ (K, A, \{F_i\})$
WHERE <condition>

Here P_is are attribute names in the form F and <condition> is specified in the following:

<condition> :: = <condition> AND <condition> |
<condition> OR <condition> |
NOT <condition> | (<condition>) |
<elementary phrase>
<elementary phrase> :: =
$R.a\ op\ v \mid \{R.a\}_U sop\ S \mid$
$R1.a1\ op\ R2.a2 \mid \{R1.a1\}_{U1}\ sop\ \{R2.a2\}_{U2}$
$op ::= < \mid \leq \mid = \mid \neq \mid \geq \mid >$
$sop ::= \subset \mid \subseteq \mid \equiv \mid \supseteq \mid \supset$
a is an attribute,
R is a repeating group which contains a,
U is a parent repeating group of R,
$\{R.a\}_U$ represents a set of values of a group by U,
v is a value,
S is a constant set.

Our strategy is to translate the FQL query into an equivalent relational query on the decomposed relations. After the relational query is executed, the resulting relation is then transformed into a form using a special database machine instruction.

Algorithm 2
Translation of an FQL query.
INPUT: a form query Qf
Qf = { SELECT: $P_1, \ldots, P_k$
FROM $F = R(K, A, \{F\ ;\})$
WHERE <condition>}
OUTPUT: a result form Fr
METHOD:
(1) Construct a query on the decomposed relations:*

SELECT $P_1, \ldots, P_k$
FROM $R_1, R_2, \ldots, R_m$
WHERE <link condition>
AND <projection condition>

*For better readability, we use a SEQUEL-like syntax to describe the relational query. The translation of this query into the proposed database machine language is obvious.

DEPARTMENT

DD#	DNAME
1	Car
2	Ship

CLERK

DD#	CE#	CAGE
1	101	25
1	102	28
1	103	35
2	201	25
2	202	30
2	203	34

ENGINEER

DD#	EE#	EAGE
1	110	25
1	115	35
2	120	30
2	125	35

EXPERIENCE

DD#	EE#	ED#	DATE
1	110	1	1978
1	110	2	1979
1	115	3	1977
1	115	2	1979
1	120	2	1978
1	120	1	1979
1	125	1	1976
1	125	2	1978
1	125	1	1979

Figure 2—Decomposed relations

The relations $R_1, R_2, \ldots, R_m$ are the decomposed relations referenced in the <condition> of the FQL query. Since we use the same name for the repeating groups and the decomposed relations, this mapping is straightforward.

Two types of relations are included in the FROM list: (1) *select relation*—relations required to perform the <link condition>; and (2) *target relation*—relations required to perform the <projection condition>. The <link condition> is the joins created by the decomposition of a form into relations. Let K_x indicate the key of relation x and K_{xy} indicates the intersection of the keys of relations x and y. The select relations and <link condition> are derived from Table II.

TABLE II—Select relations and <link condition>

FQL <condition>	select relations	<link condition>
$R.a$ *op* v	R	$R.a$ *op* v
$\{R.a\}_U$ *sop* S	U, R	$R.K_U = U.K_U$ AND $\{R.a$ group by $K_U\}$ *sop* S
$R.a$ *op* $S.b$	R, S	$R.K_{RS} = S.K_{RS}$ AND $R.a$ *op* $S.b$
$\{R.a\}_U$ *sop* $\{S.b\}_W$	R, S, U, W	$R.K_U = U.K_U$ AND $S.K_W = W.K_W$ AND $U.K_{UW} = W.K_{UW}$ AND $\{R.a$ group by $K_U\}$ *sop* $\{S.b$ group by $K_W\}$

The <projection condition> are the joins between the target relations and certain select relations. Let Z denote a target relation. The <projection condition> are derived as shown in Table III:

TABLE III—Projection condition

FQL <condition>	<projection condition>
$R.a$ *op* V	$R.K_{RZ} = Z.K_{RZ}$
$\{R.a\}_U$ *sop* S	$U.K_{UZ} = Z.K_{UZ}$
$R.a$ *op* $S.b$	$R.K_{ZRS} = Z.K_{ZRS}$ AND $S.K_{ZRS} = Z.K_{ZRS}$
$\{R.a\}_U$ *sop* $\{S.b\}_W$	$U.K_{ZUW} = Z.K_{ZUW}$ AND $W.K_{ZUW} = Z.K_{ZUW}$

(2) Execute the transformed query and obtain a result relation B_f.
(3) Group B_f into form using the keys of $R_1, R_2, \ldots, R_m$. ■

4. THE DATA LANGUAGE FOR THE NETWORK APPROACH

To support a DBTG interface on a relational database machine, it is essential to solve the following two problems:[14]

1. Transformation of DBTG data structures into relational data structures.
2. Transformation of DBTG data manipulations into queries for the database machine language.

The network data structure of a DBTG system is more complex than the hierarchical approach. It has two basic constructs: RECORD type and SET type. A RECORD type is a hierarchical structure consisting of data items and repeating groups. It is similar to the concept already detailed in the previous section. The SET type represents links among RECORD types. Each SET consists of an OWNER record type and one or more MEMBER record types. This is typically implemented in DBTG as a pointer chain (one direction or binary direction) that starts from one owner instance and links up all member instances. In a relational database machine, SET types can be removed by a normalization procedure similar to Algorithm 1. Any data access traversing a SET instance can be performed by an equivalent join operation on the common attributes of the OWNER relation and the MEMBER relation. To make this operation possible in general, however, it is necessary to introduce an attribute in the MEMBER relation to represent the rank of each record within a SET instance.

The performance of the data access can be enhanced by additional data structures. It was indicated in Yao[13] that three types of additional storage organizations can be defined: (1) Indexing—defining a tree structure to provide random access to records in a relation, (2) linking—defining a pointer chain similar to that of the SET implementation, and (3) clustering—storing the member records of a set close to its owner. In another proposal, an index structure is designed to store or-

dered (owner,member) pairs of set instances.[5] All of these data structures may be employed in a relational database machine to enhance data access performance. The use of these features, a database design problem, obviously depends on system tradeoffs and their availability in a particular database machine.

Example 1

Given a DBTG data structure for a supplier-parts database (Figure 3).

Record types:
  S(S#,SNAME,STATUS,CITY)
  P(P#,PNAME,COLOR,WEIGHT,CITY)
  SP(DATE,QTY)

Set types:
  S-SP
  P-SP

The following equivalent relations may be created:

  S(S#,SNAME,STATUS,CITY)
  P(P#,PNAME,COLOR,WEIGHT,CITY)
  SP(S#,P#,DATE,QTY,SRANK,PRANK)

The following additional data structures can also be created:

  UNIQUE CLUSTERING INDEX INDS ON S# OF S
  UNIQUE CLUSTERING INDEX INDP ON P# OF P
  LINK S-SP FROM S(S#) TO SP(S#)
    ORDER BY P#
  LINK P-SP FROM P(P#) TO SP(P#)
    ORDER BY S#

■

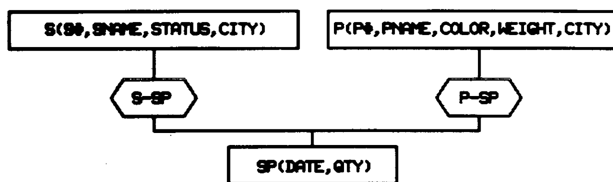


Figure 3—A supplier-parts DBTG database schema

The data manipulation language of DBTG system is a procedure-oriented language. Its operands are RECORD and SET instances. The operations on a hierarchical RECORD type (i.e., RECORD that contains repeating groups) have been discussed in the previous section. The transformation of SET traversal operations is given in Table IV.

Data manipulation using a back-end database machine requires additional communication time between the host and the back end. A data traversal interface can be very ineffective because of the number of times communications are initiated. Although the additional communication cost is unavoidable in this type of data access, it can be significantly reduced if enough buffer space is provided in the back-end and host systems.

TABLE IV—The transformation of SET traversal operations

DML of DBTG system	Database Machine Operation
MOVE 'New York' TO CITY IN S FIND ANY S USING CITY IN S	SELECT S.ALL FROM S WHERE S.SNAME = 'NewYork'
FIND <position> SP WITHIN S-SP FIRST NEXT WHERE <position> = PRIOR LAST N-TH	SELECT NEXT SP.ALL FROM S, SP WHERE S.S# = SP.S# AND S.SRANK = <position>
FIND OWNER WITHIN S-SP	SELECT NEXT S.ALL FROM S,SP WHERE S.S# = SP.S#
MOVE 'pl' TO P# IN SP FIND SP WITHIN S-SP CURRENT USING P# IN SP	SELECT NEXT SP.ALL FROM SP WHERE SP.P# = 'pl' GROUP BY SP.S#
MOVE 'S1' TO S# IN S GET SP	SELECT NEXT SP.ALL FROM SP WHERE S# = 'S1'
ADD 20 TO STATUS IN S MODIFY S	UPDATE S SET S.STATUS = S.STATUS + 20
CONNECT SP TO S-SP	UPDATE SP SET S.SRANK = MAX (S.SRANK) + 1 GROUP BY SP.S#
DISCONNECT SP FROM S-SP	UPDATE SP SET S.SRANK = 0

To consider the relative costs of SET traversal operations, let us define a few basic access cost equations. The following figure shows the two communication links in a host/back-end system.

The transmission cost between the host and the back-end can be represented by

$$f(x) = (c_1 + c_2 * w) * [x/b] + r * x$$

where x is the number of bytes transferred, b is the buffer size, $[x/b]$ is the number of transmissions required, w is the average waiting time, c_2 is the number of packets sent in each transmission, c_1 is the transmission overhead, and r is the transmission speed. In our implementation of a host interface using a 9600-baud connection, the parameter values are $c_1 = 20$, $c_2 = 4$, $w = 20$ ms, and $r = 0.83$.

The transmission time between the back-end and the disk can be similarly estimated:

$$g(y) = wd * [y/b] + y/s$$

where y is the number of bytes transferred, wd is the average disk access time, and s is the disk transfer rate. For 1 MB disk transfer rate, $s = 1000$ byte/ms. wd is assumed to be 50 ms.

We further assume that the SET type has an OWNER record type O and one MEMBER record type M. The record size of O is p, the record size of M is q, and the mapping between owner and member is $1{:}n$. In a relational system the time required to access all the member records for a given owner record can be estimated by

$$t_1(p,q) = t_0 + f(p) + f(n^*q) + g(p) + g(n^*q) \qquad (1)$$

where t_0 is the communication initiation time, assuming $t_0 = 863$ ms, $p = 128b$, $q = 256b$. In the case of DBTG SET traversal, one owner and one member record are transmitted each time. The transmission time required is therefore:

$$t_2(p,q) = t_0 + n^*(f(p) + f(q) + g(p) + g(q)) \qquad (2)$$

Assuming that the buffer size in the back-end is b. Since there are a total of n records of size q to be accessed, the number of times the disk must be accessed is $m = (n^*q)/b$.

The transmission times required for each access are illustrated as follows:

$$\begin{array}{ll} 1st: & f(p) + f(q) + g(p) + g(b) \\ 2nd: & f(p) + f(q) + g(p) + 0 \\ & \vdots \\ mth: & f(p) + f(q) + g(p) + g(b) \\ m+1th: & f(p) + f(q) + g(p) + 0 \\ & \vdots \end{array}$$

The total time required is therefore

$$t_3(p,q) = t_0 + n^*(f,(p) + g(p) + f(q)) + m^*g(b) \qquad (3)$$

The results of these cost equations are plotted in Figures 4 and 5. It is evident that the relational data access is much more efficient (see t_1). Network data access without using a buffer can be very inefficient, since multiple accesses are required. The curve t_2 gives the access cost without buffer and t_3 gives the access cost with a 256-byte buffer. Figure 5 shows the sensitivity of access time to various buffer sizes. It is interesting to note that even the worst case of relational data access is still far better than the best case of network data access. Therefore the support of a network data model interface on a relational database machine is feasible, but relatively inefficient.

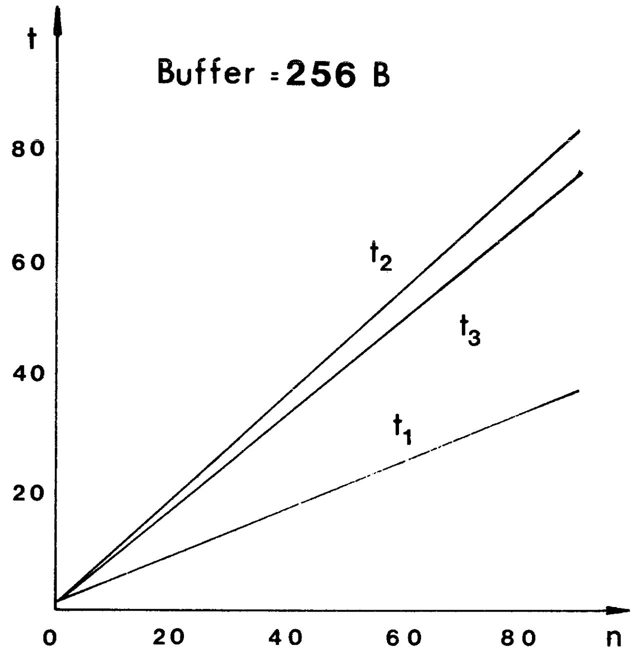


Figure 4—Comparison of relation and network data access

V. CONCLUSION

The choice of a data model and set of operations for a database machine has a great effect on the efficiency of its user interface implementation. Such data models and interfaces must be carefully determined during the initial design stage. A properly designed database machine language will make it feasible and efficient in supporting multiple data languages.

This paper has presented a set of database machine language requirements. These requirements are based on our experience in implementing the XQL system as a user interface to a database machine. The XQL system provides the SEQUEL query language, menu system, interactive form processor, and report writer. Host language interface is provided by imbedding SEQUEL statements in application programs.

The algorithms for supporting hierarchical and network interfaces by using a relational database machine are introduced. The relative performances for supporting relational

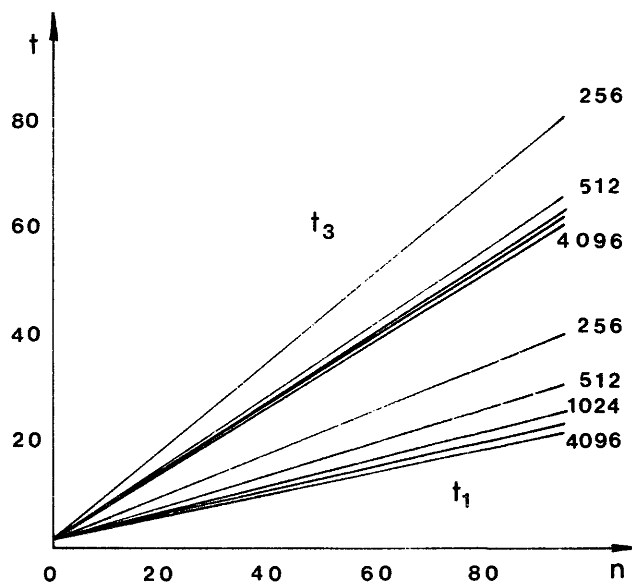


Figure 5—Comparison of the buffer size influence to relation and network data access

and network user interfaces are compared. Our analysis shows that it is possible to support multiple user interfaces by using a relational database machine. The support for a network user interface is less efficient when compared with the relational approach. The performance can be improved by providing additional buffer space. However, even with a very large buffer size, the performance of the network user interface is still significantly worse than that of the relational interface.

APPENDIX A

If the set of basic operations in the database machine is not sufficient, it can sometimes be difficult to support certain data language. Examples are provided here of our implementation of the system XQL/IDM based on a database machine. For better readability, we use a QUEL-like language to describe the object database machine language code.

Example 1

```
XQL: select name
     from Employee
     where salary > 2000
     minus
     select name
     from Department
     where manager = "D. Smith"
```

The object code:

```
range of e is Employee
retrieve into Templ (e.name)
where e.salary > 2000

range of d is Department
retrieve into Temp2(d.name)
where d.manager = "D. Smith"

range of t1 is Temp1
range of t2 is Temp2
delete t1
where t1.name = t2.name

retrieve(t1.name)

destroy Temp1
destroy Temp2
```

This query can be more efficiently handled, if the database machine has set operations.

Example 2

```
XQL: select name
     from Employee
     where departno not in
         select departno
         from Department
         where manager = "D. Smith"
```

The object code:

```
range of e is Employee
retrieve unique into Temp1(e.departno)

range of d is Department
retrieve unique into Temp2(d.departno)
where d.manager = "D. Smith"

range of t1 is Temp1
range of t2 is Temp2
delete t1
where t1.departno = t2.departno

retrieve(e.name)
where e.departno = t1.departno

destroy Temp1
destroy Temp2
```

This query can be more efficiently handled if the database machine can support a set operation in search conditions.

APPENDIX B—A PROPOSED SET OF OPERATIONS FOR THE RELATIONAL DATABASE MACHINE

1. Projection: Project $(a_1, \ldots, a_p)$ from R where a_i is an attribute name of the relation R. The result is a new relation obtained from R by deleting from each tuple all attributes not listed in $(a_1, \ldots, a_p)$. We note that after the elimination of some attributes from R, the new relation may contain duplicated tuples.
2. Selection: Select from R where <condition>
   Here the <condition> is a Boolean predicate which specifies the search condition. Each term in the predicate contains an arithmetic comparison operator and two operands that are constants or attribute names of R. The terms in the predicate are linked together by the logical operators AND, OR, and NOT. The result is a new relation which consists of all tuples from R satisfying the given <condition>.
3. Selection of next tuple: Select next from R where <condition>
   Same as 2 except only one tuple is returned. When a relation is ordered this tuple is the one logically next to the last selected tuple. The first execution of this operation returns the first tuple in R satisfying <condition>.
4. Ordering: Order R by a
   Where a is an (or a set of) attribute name of the relation R. The result is a new relation containing all the tuples from R sorted on the attribute (set) a.
5. Grouping: Group R by a
   Where a is an (or a set of) attribute name of the relation R. The result is a new relation containing all the tuples from R "grouped" on the values of the attribute (set) a. This is useful when aggregate functions are to be performed.
6. Unique: Unique R

Relation is sorted on its key and tuples with identical keys are removed.

7. Union: R union S
   The relations R and S are identically defined. The result is a new relation containing all tuples from both R and S.
8. Subtraction: R subtract S
   R and S again must be identically defined. The result is a new relation containing all tuples that are in R but not in S.
9. Intersection: R intersect S
   R and S again must be identically defined. The result is a new relation containing all tuples that are in both R and S.
10. Joining: R join S where <condition>
    Each term in the join <condition> is of the form (a op b) where a is an attribute name of R, b is an attribute name of S, and op is an arithmetic comparison operator. The result is a relation containing all possible concatenations of tuples from R and S satisfying the given <condition>.
11. Get next join tuple: R join next S where <condition>
    The join is defined as before. The result returned is the next tuple from the join result.
12. Aggregate functions: F ($R.a$)
    F is one of the aggregate functions: average, sum, min, max, count. The function is performed on the column a of the relation R. The result returned is a single scaler quantity. If the relation is grouped, then the aggregate function is performed for each group.
13. Delete: Delete from R where <condition>
    All the tuples from R satisfying the condition are deleted.
14. Insert tuple: Insert T into R
    The tuple as given in T is inserted into the relation R.
15. Update: Update <list> in R where <condition>
    <list> is a list of update actions. Each action is of the form $A_i = u_i$ where A_i is an attribute name of R and u_i is the new value to be assigned. The update list is performed for all tuples satisfying the given <condition>.
16. Assignment: $S := R$
    The action is to store R into database as a new relation named S.

The following operations are for expanding select condition. The result of these operations is a Boolean value that can be served as a condition.

1. Contains: R contains S
   R and S are two relations of arity k. The result is True if all tuples in S are tuples in R, False otherwise.
2. Does not contain: R does not contain S
   R and S are two relations of arity k. The result is True if there is any tuple in S is not tuple in R, False otherwise.
3. Equal: R equals to S
   The result is True if R and S have the same tuples, False otherwise.
4. In: T is in R
   R is a relation, T is a tuple. The result is True if there is a tuple in R equal to T, False otherwise.
5. Not in: T not in R
   The result is True if there is no tuple in R equal to T, False otherwise.

These operations are the most common ones. The database machine should contain all of these operations. On the basis of these operations, the queries expressed in most data models can be easily handled.

REFERENCES

1. M. M. Astrahan, et al. *System R: Relational Approach to Database Management.* ACM Trans. on Database Systems *1:2,* June, 1976.
2. D. D. Chamberlin, et al. *SEQUEL 2: A Unified Approach to data Definition, Manipulation, and Control.* IBM J. Res *20:6* 1976.
3. E. F. Codd. *Relational Completeness of Data Base Sublanguages.* Data Base Systems, Courant Computer Science Symposia Series, *Vol* 6. 1972.
4. C. J. Date. *An Introduction to Database Systems.* Third Edition Addison-Wesley Publishing Company, *1981.*
5. T. Haerder. *Implementing a Generalized Access Path Structure for a Relational Database System.* ACM Trans. Database Systems, *3:3,* Sept. 1978.
6. Britton-Lee Inc. *IDM 500 Intelligent Database Machine 1980.*
7. Software Systems Technology, *XQL/IDM System Reference Manual,* Version 1.0, Jan. 1982.
8. D. Luo, and S. B. Yao. *Form Operation By Example—Language for Office Information Processing.* Proc. ACM-SIGMOD International Conference on Management of Data, *1981.*
9. *SQL/DS Reference Manual 1981.*
10. M. Stonebraker, et al. *The Design and Implementation of INGRES* ACM Trans. on Database Systems, *1:3,* Sept. 1976.
11. J. D. Ullman. *Principles of Database Systems.* Computer Science Press, *1980.*
12. J. Woodfill, et al. *INGRES Reference Manual,* Version 7, *April, 1981.*
13. S. B. Yao. *Optimization of Query Evaluation Algorithms.* ACM Trans. on Database Systems, *4:2,* June 1979.
14. R. H. Katz and E Wong. *Decompiling CODASYL DML into Relational Queries.* ACM Trans. on Database Systems, *7:1,* March 1982.

Performance analysis of database join processors

by FU TONG
Chongquing University
Chongquing, China
and
S. BING YAO
University of Maryland
College Park, MD

ABSTRACT

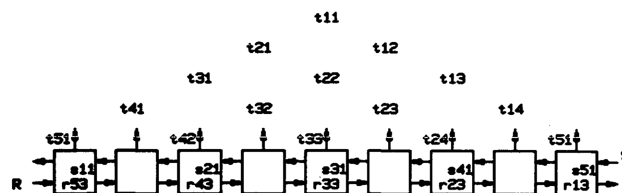
The architecture and performance of a two-dimensional join processor for relational databases are introduced in this paper. The performance of several recently proposed join processor designs is also analyzed. The processing time (performance) and hardware complexity (cost) of these different join processor approaches are compared in detail. The result shows that the two-dimensional join processor array has the best performance/cost ratio among the design approaches considered.

1. INTRODUCTION

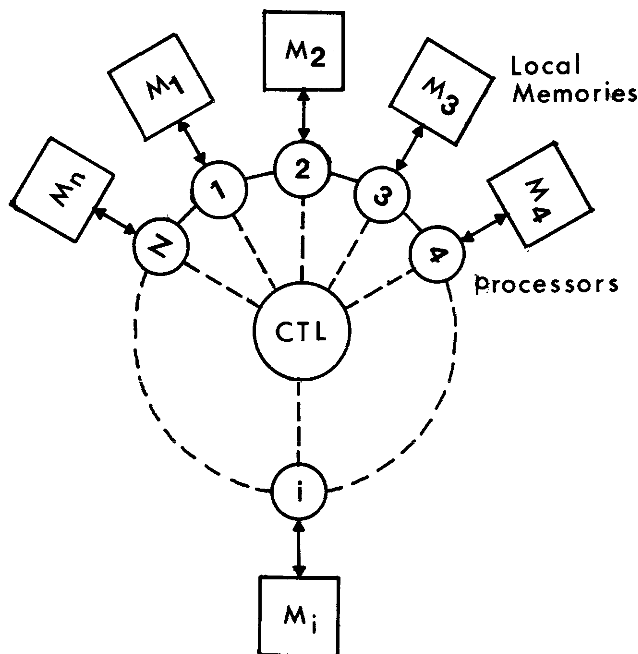
In a relational database system, the data manipulation operations usually involve selection, projection, and join operations. One of the most important operations in query processing is the join operation. It is usually the most costly operation to perform. The efficiency of the join operation has a determining effect on the system performance. Because of its importance, the join operation has been a subject of intensive study in the development of a relational database system. To maximize the join performance, query optimization algorithms have been developed to determine the most effective access path.[15–17] While these methods do prove to be effective in the content of conventional computer systems, a closer look at the nature of the join operation reveals the opportunity for concurrent operations. Using appropriate hardware and system organization, the concepts such as parallel processing and pipelining could be applied to greatly enhance the performance of the join operation. This new approach has become feasible with recent advances of hardware technology. It is now possible to design specialized processors to perform complex, dedicated functions.

Several designs for processing the join by hardware have been proposed. These include Kung's systolic array,[5] DeWitt's MIMD machine DIRECT,[3] Hsiao's DBC,[9] Wah and Yao's DIALOG,[14] and CAFS.[8] Among these special join processor designs, only the *systolic* array, DBC, and DIALOG processors are specifically designed for VLSI implementation. Let us briefly examine and compare these design approaches (see Figure 1).

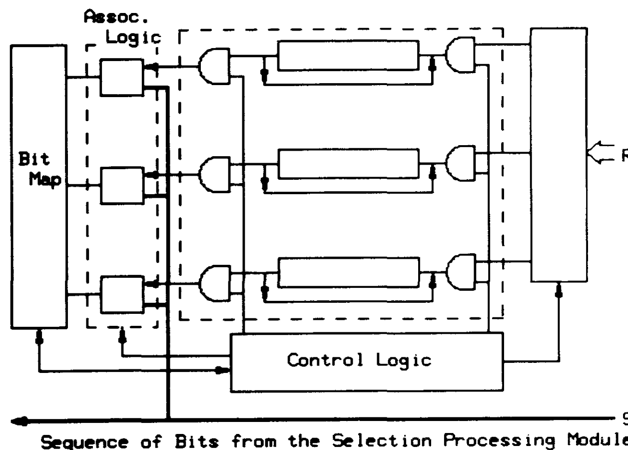
1. The systolic array join processor is basically a one-dimensional processor array. The two relations to be joined are piped into the array from two opposite directions. The two pipelines move in synchrony, one step for each time unit. Join processing takes place when values of records from different relations meet in a join processor element. The number of processors required in the systolic array is twice the number of records in the larger relation to be joined. An alternative method is to load the smaller relation and pipe only the larger one. In this case, the number of processors required is equal to the number of records in the smaller relation.
2. The join processor array in DBC is arranged in a circular fashion. One of the relations to be joined is partitioned and loaded into the memories of the join processors. The keys of records are stored in an associative memory, while the records are stored in a hash table. After initial loading is complete, the other relation to be joined is piped into the join processors. Each processor compares the join values associatively and, if successful, accesses



a. "Systolic" array



b. Join Processor in DBC



c. Bit-sliced associative join processor

Figure 1—Various VLSI join processor designs

the records in the hash table; then it propagates the record into the next processor.

3. In the bit-sliced associative join processor of DIALOG, the join values from one of the relations to be joined are stored in a bit-sliced associative memory. Records of the other relation are compared with the contents of the associative memory in bit slices. The bit-sliced associative memory can be viewed as a one-dimensional array of processors.

All the previous designs of join processors are, in fact, variations of one-dimensional processor arrays and are based on pipelining or a serial processing mechanism. In this paper, the design of a two-dimensional join processor array will be introduced. The one-dimensional array designs are viewed as special cases representing a row from a two-dimensional array. While most of the proposed join processors claim to have *linear* processing time, a careful inspection of their design shows significant performance differences. The object of this paper is to analyze these different approaches, using a unified framework and parameters in order to gain insight into the nature of join processing.

We begin by defining the join operation. The join of two relations R and S can be expressed by

$$R \underset{F}{\overset{x}{\bowtie}} S = \{(r,s) \mid r \in R,\ s \in S,\ r.A_i\ \theta\ s.B_j\}$$

where F is a logical expression on the attributes of relations R and S, i.e.,

$$F = F_1 \cdots F_k \cdots F_q$$
$$F_k = A_i \theta B_j$$

Where A_i and B_j are the join attributes of the relations R and S respectively, θ represents one of the following arithmetic comparison operators: $<, \leq, =, \neq, \geq, >$.

Example: Consider the two relations:

EMPLOYEE(EMP,	DEPT#)
Carter	1
Smith	1
Todd	2
Wang	3

DEPARTMENT(DEPT#,	LOCATION)
1	NY
2	DC
3	DC

To make a list of all employees and their working locations, we must perform the following join:

EMPLOYEE⋈DEPARTMENT

(EMP,	DEPT#,	LOCATION)
Carter	1	NY
Smith	1	NY
Todd	2	DC
Wang	3	DC

Conventionally there are two basic approaches to evaluate a join:[15]

1. Nested loop. Each S-record is compared with all the R-records on all q join values. This operation, in fact, performs a Cartesian product of the two relations.
2. Ordered merge. The records in both relations are ordered on the join values by either sorting or indexing. Records in both relations are then scanned and compared in one pass.

Although the second method appears to make fewer record accesses, there is an added cost for sorting and/or maintaining indices. Furthermore, during the ordered merge, if multiple records from both relations contain a matching join value, a Cartesian product of these records must be performed. This step is usually performed using a nested loop method.

The sequential evaluation of a nested loop is very inefficient. Assume that the number of Boolean terms (i.e., the number of comparison value-pairs from R and S) in F is q and the number of tuples (i.e., records) in R and S are m and n respectively. The number of operations required for sequential evaluation of a nested loop is

$$N_j = qmn$$

This can be greatly improved by parallel join processors. In principle, the nested loops can be realized with a three-dimensional join processor architecture, in which there is one dedicated $m*n$ processor matrix assigned for each of the q terms in F. In practice, however, there are seldom situations where many pairs of attributes are involved in F (i.e., q is small). Thus in our analysis we will consider only one- and two-dimensional join processor array architectures. We will show that a nested loop algorithm can be realized by specially designed hardware having a very high processing speed.

The following section presents a design of a two-dimensional join processor array. An analysis of alternative join processor approaches is given in Section 3. The one-dimensional join processor array is considered as a row of the two-dimensional join processor array. In Section 4, the performance/cost index of these different approaches is estimated and compared.

2. THE ARCHITECTURE OF A TWO-DIMENSIONAL JOIN PROCESSOR

a) Organization of the Processor Matrix

As mentioned above, if the number of tuples in the two relations are m and n respectively, the complexity of this operation is proportional to $m * n$. If we divide the two relations into x and y subrelations respectively and process all of the subrelations in parallel, as shown in Figure 2, the complexity of the join operation can be reduced to $(m * n)/(x * y)$.

The architecture of the join processor array is shown in Figure 3. There are four components of the system: the internal buffer storages for partitioned key values from the relations R and S; the processors which compare the key values

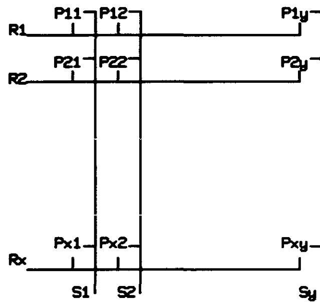


Figure 2—The join processor matrix

read from the buffer storages; the output buffer storages which store the addresses (or pointers) of the matched tuples; and control/timing logic which issues the microoperation signals. Each of the four components, except the control/timing logic, is in turn composed of a set of identical cells. The highly regular structure makes it a candidate for VLSI implementation.

When processing the join, only the join values (attributes) in records from relations R and S need to be compared for a match. Therefore, only these specified values need to be stored in the internal buffer memories RB_i, s and SB_j, s (Figure 3). The processors P_{ij}s in Figure 2 are simply a set of comparators that compares one byte of x values from RB_is with one byte of y values from SB_js in parallel. The results of comparison are indicated by the flag registers F_{ij}s. Usually a value may have a length of many bytes.

Instead of concatenating and storing the joined records directly in the output buffer B (Figure 3), only the addresses (or pointers) of the joined records are stored. The final join result can be constructed using these pointers. These two processes can be pipelined to gain efficiency.

Now let us describe briefly how the JP-matrix (Figure 3) works. Before the join processor is activated, the two relations (files) are accessed through the *data filter*.[11] The selected/projected records are stored in the system buffer, and the corresponding key values are stored in the internal buffers $RB's/SB's$ respectively. At the same time, other initial parameters such as the beginning (head) addresses of the two selected relations R/S in the system buffer (HAR/HAS), the

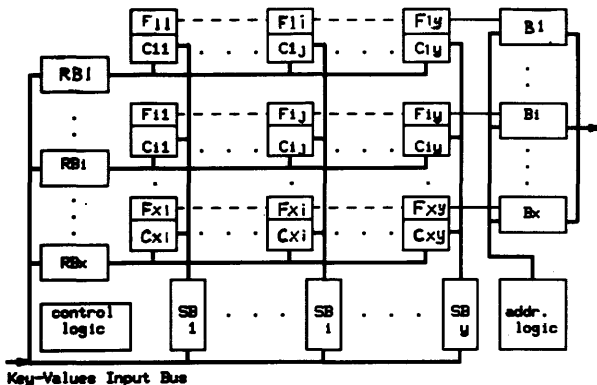


Figure 3—The architecture of a two-dimensional join processor

record length (LRR/LSR), the key value length (LVR/LVS), the number of records in the selected relation ($\#R$, $\#S$), and the constants x and y are input into the JP.

As soon as the initialization of the JP is done, the join processor is started by the "start" signal from the system controller. The first phase of the join process is to read out the first set of key value pairs from the $RB's$ and $SB's$ and broadcast them to the comparator matrix along the x- and y-internal data busses respectively (refer to Figure 3). The x R-values are compared with y S-values in $x*y$ comparators simultaneously.

The second phase assembles the result of the join. If there exists at least one nonzero flag in a row of the flag-matrix, say the (i, j)-th bit in Figure 4, the address of the record corresponding to the key value read from RB_i must be stored along with the address of the corresponding S-record into the output buffer B_i. If more than one flag are nonzero, then their corresponding record addresses must also be stored. On the other hand, an all-zero row indicates failure in matching, and no result is formed.

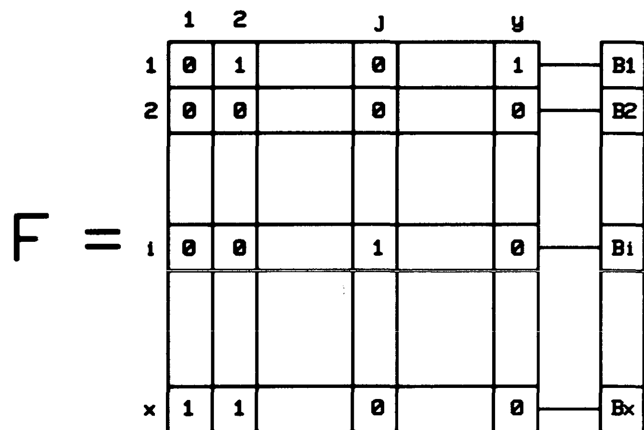


Figure 4—The flag matrix of a 2-D join processor

The comparison of the next set of S-values (with the same set of R-values) takes place immediately after the resultant flag-matrix is accessed and the first set of matched records addresses are stored in the output buffers. This two-phase process repeats until all the S-values stored in each SB are exhausted. The same process is then repeated for all the R-value sets. The detailed algorithm for performing the join is given in Appendix A.

b) Performance Evaluations

The actual time required for the join process depends on the following factors:

1. The basic parameters such as m, n, x, y, etc.
2. The expectation of the number of matched records, E
3. Memory access time
4. Word length of the memories
5. Initialization overhead
6. Terminating overhead

Suppose that we neglect the last two factors and consider only the time interval from the beginning of the join operation to the end of the matched record address output operation. We will first estimate the processing time T_b required for matching one set of R- and S-records. This will be used subsequently to compute the total join processing time. The basic join processing time T_b consists of two sub-intervals (refer to Figure 5):

1. The time required to compare the key-value pairs and get the comparison result, T_c
2. The time required for computing the addresses and write into the output buffer, T_a

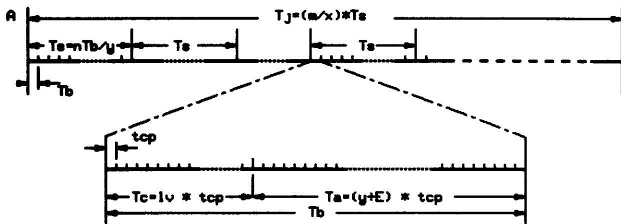


Figure 5—The scheduling of a 2-D join processor

Assume that a pair of l_c-byte values can be compared in one clock period t_{cp}. Assume further that the times required to generate a set of addresses of the compared records and to store them in the output buffer are both equal to t_{cp}. The basic processing time is

$$T_b = T_c + T_a = \frac{l_v}{l_c} t_{cp} + t_{cp} + (y + E) t_{cp} \tag{1}$$

where

$$\frac{l_v}{l_c} t_{cp} + t_{cp}$$

is the time required to compare a value of l_v bytes in a l_c bytes comparator and get the comparison result, yt_{cp} is the time needed for generating the addresses of y S-values in sequence, and Et_{cp} is the expected time needed for storing E matched addresses (among the y addresses generated) into B.

The expectation for y S-values matching with one R-value, E, may be estimated as follows. Assume that the total distinct values in join attribute domain is k. The value of k depends on the property of the attribute and usually can only be estimated. For a particular join value in a R-tuple, the probability for it to match one of the y selected join values of the S-tuples is $1/k$. The expectation of the number of successful matching in y join values is, therefore,

$$E = \sum_{j=1}^{y} 1 * \frac{1}{k} = \frac{y}{k} \tag{2}$$

The processing time required for one iteration which compares all S-values with one set of R-values is simply n/y repetitions of T_b. The total join processing time is therefore m/x this single iteration:

$$T_{j2} = \frac{m}{x}\left(\frac{n}{y} T_b\right) = \frac{m*n}{x*y}\left(\frac{lv}{lc} + y + E\right) t_{cp} \tag{3}$$

Substituting E from expression (2), we have

$$T_{j2} = \frac{mnl_v}{xyl_c} t_{cp} + \frac{mn}{x}\left(1 + \frac{1}{k}\right) t_{cp} \tag{4}$$

3. ANALYSIS OF ALTERNATIVE JOIN PROCESSOR APPROACHES

In this section several join processor designs are reviewed, and equations for estimating the performance for each design are derived. These equations will be used as the basis for the performance comparison.

a) One-dimensional Processor Array with Broadcasting

Each row of the two-dimensional join processor matrix defines a one-dimensional join processor array. Figure 6 shows that only the S relation is partitioned into N subrelations. Join values of the R relation stored in the R buffer are broadcasted to the processors $P_1, P_2, \ldots, P_N$ for parallel comparison.

The basic processing time T_b can also be estimated by using expression (1), in which parameter y now represents the total number of processors. If we denote the total number of parallel processors by N, the total processing time, T_j is simply $\frac{mn}{N}$ repetitions of the basic cycle:

$$T_{j1} = \frac{mn}{N} T_b = \frac{mn}{N}\left(t_t + t_2\right) = \frac{mn}{N}\left(\frac{l_v}{l_c} + 1 + N + \frac{N}{k}\right) t_{cp} \tag{5}$$

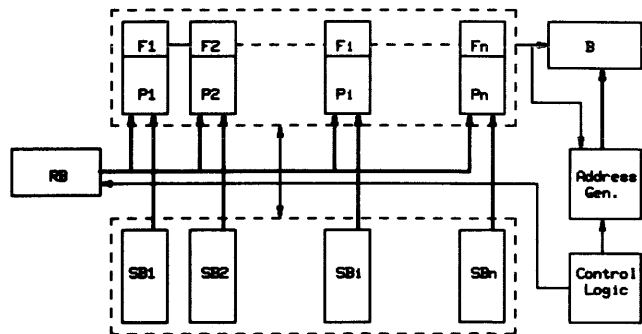


Figure 6—Architecture of a one-dimensional join processor

b) One-dimensional Processor Array with Pipelining

(1) Systolic join processor array[5]

The general organization of this approach can be represented by the organization shown in Figure 7. Suppose we only consider the join values to be processed. The R-values

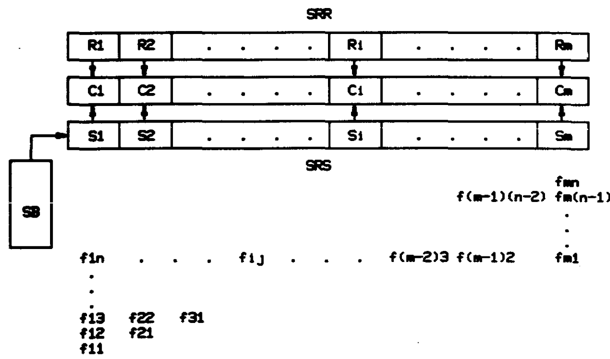


Figure 7—Architecture of a "systolic" array

are first loaded in the shift registers SRR. The S-values are then piped through the processors via the shift registers SRS. The join results, which are produced after comparing the i-th S-value with the j-th R-value, are indicated by f_{ij}.

The processing time required is $t_p = t_r + t_s + t_{tr}$, where t_r is the time required for preloading all the R-values, t_s is the time required for piping all S-values, and T_{tr} is the time required to test the result f_{ij}'s and produce the join result. Assuming $t_{tr} > t_s$, if the S-value piping is overlapped with the result processing, the join processing time is now

$$T_{js} = t_r + t_i + t_{tr} \qquad (6)$$

where t_i is the time required for inputing the first S-value into the S shift registers. Usually t_i is much smaller than t_r or t_{tr}.

(2) Join processor of DBC

The architecture of the join processor proposed by the DBC project[9] can be illustrated in Figure 8. The S relation (i.e., the source relation) is partitioned into N subsets. Each S record is entered into a hash table (i.e., A memories) by the values in associative memory *am*. After all the S records are in place, the R records are piped through the N processors.

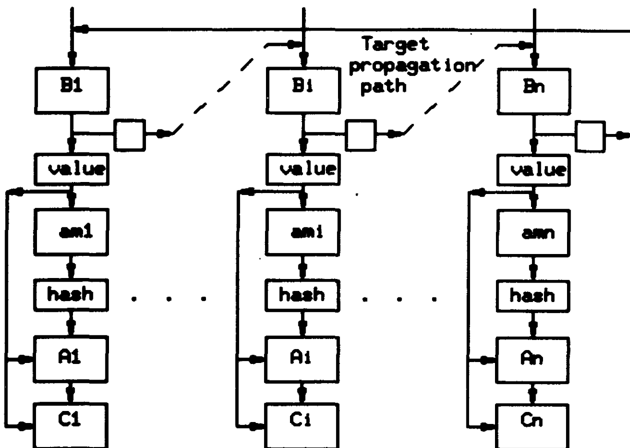


Figure 8—Architecture of a DBC join processor

The join values of R are compared with those stored in *am*'s. If a match is found, the corresponding record is retrieved from A memories and output to the C memories.

Conceptually this method is identical to the *systolic* array. The implementation difference is that an associative memory is used for join value matching, and a hash table is used for record storage. The R records are allowed to enter into all of the processors simultaneously. This could cause implementation difficulties if many processors are to be integrated on a single VLSI device.

The processing time required for this join processor can be estimated by the following formula:

$$T_{jd} = \frac{m}{N}\left(t_b + t_{am} + t_a\right) + n\left(t_b + t_{am}\right) + \left(\frac{n_r}{Ne}\right)t \qquad (7)$$

where t_a, t_b, t_{am} are the access times for the A, B, and associative memory respectively. The size of the result relation is given by n_r. The hashing function efficiency is e.

4. PERFORMANCE EVALUATIONS

Let us first investigate the average processing time required by each of the four different join processor designs. Their hardware complexity will be investigated next. Finally, we will estimate their performance/cost ratio, based on these calculations.

a) Processing Time

In order to compare alternative designs in a common framework, we will make several assumptions. The two relations to be joined are assumed to have the same size of m records. The maximum length of a record is l_r bytes, and the maximum length of an attribute value is l_v bytes. All the join processing operations are synchronized by the same system clock period t_{cp}. For example, in a one- or two-dimensional join processor, if we assume that the buffer memory access time is one clock period t_{cp}, the time t_1 required for reading one R-value and N S-values and obtaining the comparison results is

$$t_1 = \frac{l_v}{l_c} * t_{cp} + t_{cp} \qquad (8)$$

where l_c is the length of the comparator. The first term is for the access and comparison of a join value in l_c-byte units. The second term is for getting the result of the comparison. Further, we use the symbol w_z for denoting the word length of memory z.

The analysis of the join processing time given in the previous section is now summarized as follows:

1-D array:

$$T_{j1} = \frac{m^2}{N}\left[\left(\frac{l_v}{l_c} + 1\right) + N + 2\frac{N}{k}\right] t_{cp} \qquad (9)$$

2-D array:

$$T_{j2} = \frac{m^2}{N}\left[\left(\frac{l_v}{l_c} + 1\right) + \sqrt{N} + 2\frac{\sqrt{N}}{k}\right] t_{cp}$$

systolic array:

$$T_{js}=\left[(m+1)\,(8l_v)+m^2+\alpha\,\frac{m^2}{k}\right]t_{cp} \quad (11)$$

where the value of α depends on whether the join result records are to be concatenated $\left(\alpha=2\,\frac{l_r}{w_r}\right)$ or only the addresses of the joined records are to be produced (α = 2).

DBC join processor:

$$T_{jd}=\left[\frac{m}{N}\left(\frac{l_r}{w_b}+\frac{l_v}{w_{am}}h+\frac{l_r}{w_a}\right)+m\left(\frac{l_r}{w_b}+\frac{l_v}{w_{am}}h\right)+\frac{m^2}{Nke}\left(\frac{l_r}{w_a}+2\frac{l_r}{w_c}\right)\right]t_{cp} \quad (12)$$

where $h>1$ is the hashing overhead, which depends upon the hashing circuit and associative logic being used.

If we assume all the memories except the *am* have the same word length, w_a and $e=1$, then we have:

$$T_{jd}=\left[\frac{m}{N}\left(2\frac{l_r}{w_a}+\frac{l_v}{w_{am}}h\right)+m\left(\frac{l_r}{w_a}+\frac{l_v}{w_{am}}h\right)+\frac{m^2}{Nk}\left(3\frac{l_r}{w_a}\right)\right]t_{cp} \quad (13)$$

Based on the equations (9) through (13), and taking one of the parameters as the variable we will compute and compare the performance of these four design approaches. We first consider the join processing time as a function of N, the number of processors. After simplification of the equations, we have:

$$T_{j1}=A\,\frac{1}{N}+B \quad (14)$$

$$T_{j2}=A\,\frac{1}{N}+B\,\frac{1}{\sqrt{N}} \quad (15)$$

$$T_{js}=C \quad (16)$$

$$T_{jd}=D\,\frac{1}{N}+G \quad (17)$$

where A, B, C, D, and G are constants readily derived from the system parameters using equations (9) through (13).

Next, let us consider the join processing time as a function of m, the size of the relations to be joined. Again we can simplify the equations as follows:

$$T_{j1}=C_1m^2 \quad (18)$$

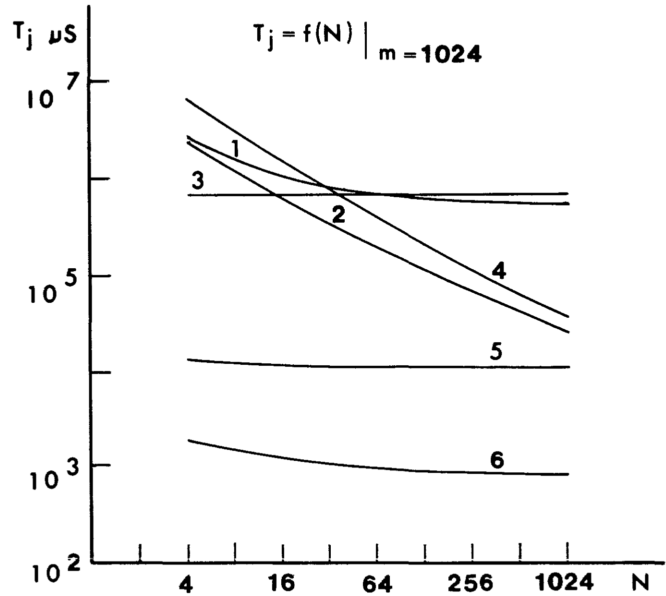
$$T_{j2}=C_2m^2 \quad (19)$$

$$T_{js}=C_3m^2+C_4m+C_5 \quad (20)$$

$$T_{jd}=C_6m^2+C_7m \quad (21)$$

where C_i's are constants determined by the join processor system parameters.

Suppose we have the following parameters: length of record $l_r=512$ bytes, length of value $l_v=64$ bytes, length of address word $w_{ad}=18$ bits, system clock period $t_{cp}=0.25$ microsec., and consider in worst case, $k=2$, length of comparators in 1-D and 2-D processor arrays $l_c=2$ bytes. Furthermore, assume the word lengths of A, B, C memories in DBC join processor are all equal to 8 bytes, the word length of the associate memory is 16 bytes, and the hash function efficiency $e=1$. In *systolic* array, we assume that only the pointers to the matched records are stored. Based on these parameters, it is easy to calculate the sets of constants A, B, C, D, G, and C_i ($i=1,\ldots 7$). The calculated results of the functions $T_j=f(N)$ and $T_j=f(m)$ for the investigated join processor designs are shown in Figures 9 and 10 respectively.



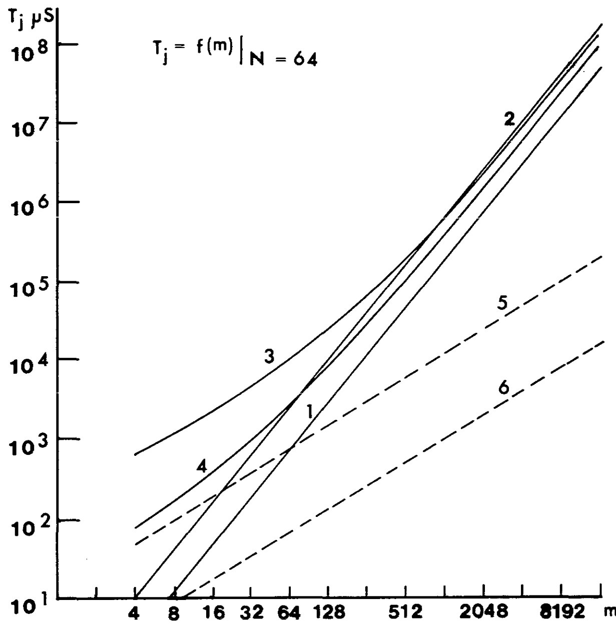
1—1-D JP
2—2-D JP
3—"systolic" array
4—DBC JP
5—DBC JP, input values output pointers
6—2-D JP, using associative memories

Figure 9—Join processing time vs. number of parallel processors (m = const)

Since *systolic* array requires the number of processors to be at least the size of the smaller relation, it is expected that the processing time is independent of the number of processors, N, as shown in Figure 9. For fixed relation size (e.g., $m=1024$), the processing time for the other designs decreases rapidly with increasing N. The rate of decreasing slows down when $N\geq 32$ in the case of the one-dimensional join processor array (curve 1 in Figure 9). No apparent improvement is noticed when N reaches 256 or greater.

The DBC join processor is the slowest for small relations and a small number of processors. Its performance improves when more processors are used. For a fixed number of processors, say $N=64$, and small relations, DBC and *systolic* array join processors are worse than one-dimensional and two-dimensional arrays. However, for a very large relation size, they become faster than the one-dimensional array, because a large number of processors are working simultaneously. In any case, the two-dimensional array has the best performance.

If we assume that the DBC join processor also works on join values instead of the entire records and producing result



1—D JP
2—2-D JP
3—"systolic" array
4—DBC JP
5—DBC JP, input values
output pointers
6—2-D JP, using associative memories

Figure 10—Join processing time vs. size of relation (N = const)

pointers instead of the result relation, its performance index should be greatly improved (curve 5 of Figures 9 and 10). The high speed is due to the use of associative memories. Obviously associative memory can be used also for the one- and two-dimensional designs.[14] The performance of the two-dimensional design using associative memories is shown in curve 6 of Figures 9 and 10.

It is apparent that the join processing times increase in proportion to the square of the relation size m for all four approaches. It was claimed by Menon and Hsiao[9] that T_{jd} is a linear function. Our result shows that T_{jd} also has exponential characteristics, because n_r, the result relation size, is actually proportional to m^2.

We also note that since the join processor must obtain its input from the relations stored in secondary storage, its performance is limited by the retrieval time for the relations R and S. If we assume these times to be T_s and T_r respectively, the actual join processing time must be greater than the sum of these two terms, i.e.,

$$T_{ja} > T_s + T_r \tag{22}$$

b) Hardware Complexity

To estimate the relative cost of the different join processor approaches, it is reasonable to compare the total amount of equivalent memory cells. The largest amount of components required is for the buffer memories and the processor array. The complexity of the controllers is almost the same for these four different designs. With these assumptions, we can derive the hardware complexity of the four designs. The following equations are derived in terms of total amount of memory cells (including those memory cells contained in the processors), Q, as the index of hardware complexity.

One-dimensional join processor array:

$$\begin{aligned} Q_1 &= 8*l_v(m+m) + 2*w_{ad}(N+1) + 8*l_c(2N+1) \\ &= 16*l_v*m + (16*l_c + 2w_a)N + (8*l_c + 2w_a) \end{aligned} \tag{23}$$

Two-dimensional join processor array:

$$\begin{aligned} Q_2 &= 8*l_v(m+m) + 2*w_{ad}(N+\sqrt{N}) + 8*l_c(N+2\sqrt{N}) \\ &= 16*l_v*m + (8*l_c + 2w_a)N + (16*l_c + 2w_a)\sqrt{N} \end{aligned} \tag{24}$$

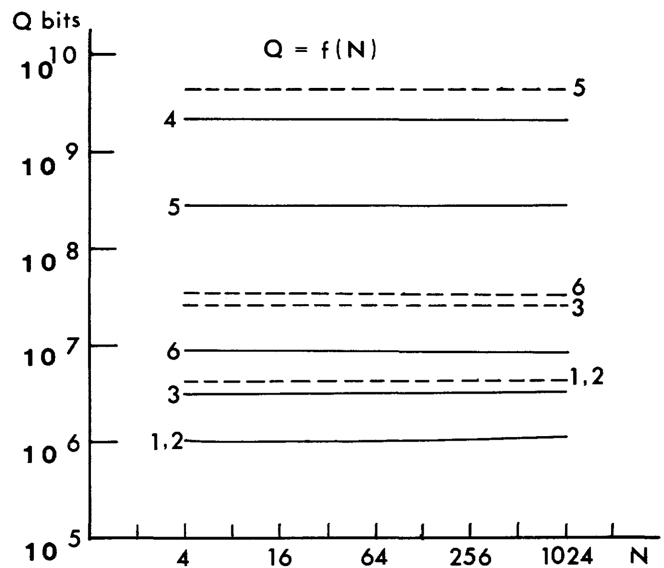
Systolic array:

$$\begin{aligned} Q_s &= 3*8*l_v*m + 8*l_v*m + m*m \\ &= 32*l_v*m + m^2 \end{aligned} \tag{25}$$

DBC join processor:

$$\begin{aligned} Q_d &= 8*l_r*m + f*8*l_r*m + w_{am}*m + 8l_r\frac{m^2}{k} \\ &= 8l_rm\left(1+f+\frac{m}{k}\right) + 8w_{am}m \\ &= 8l_rm^2 + 8[l_r(1+f) + w_{am}]m \end{aligned} \tag{26}$$

where the l_r and l_v are the lengths of record and join value in bytes respectively. The width of the associative memory word in bytes is w_{am}, and $0 < f < 1$ is the percentage coefficient for

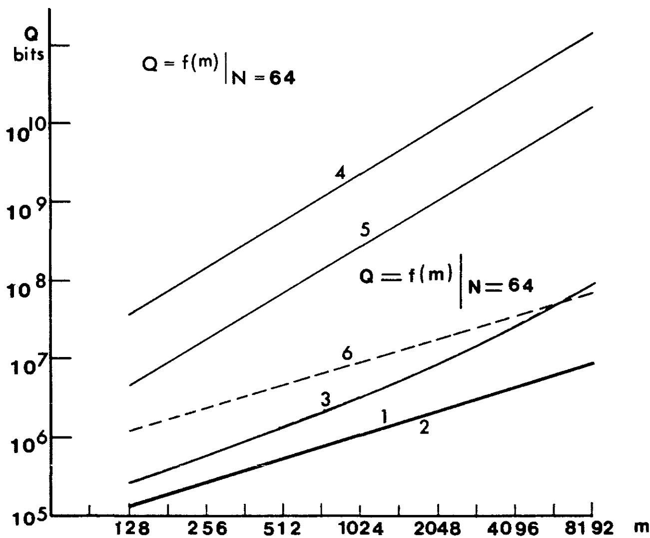


1—1-D JP
2—2-D JP
3—"systolic" array
4—DBC JP
5—DBC JP, input values
output pointers

Figure 11—Number of memory cells vs. number of parallel processors

the B memory capacity, which takes into account the speed matching between the input and output of the B memory.[9]

Figures 11 and 12 show the number of memory cells Q required by each design as a function of N and m respectively. As shown in Figure 11, the complexities are almost independent of N for constant m. On the other hand, Figure 12 shows that the complexities are proportional to m^2 for constant N. Both figures show that the complexity of one- and two-dimensional designs are less than other designs.



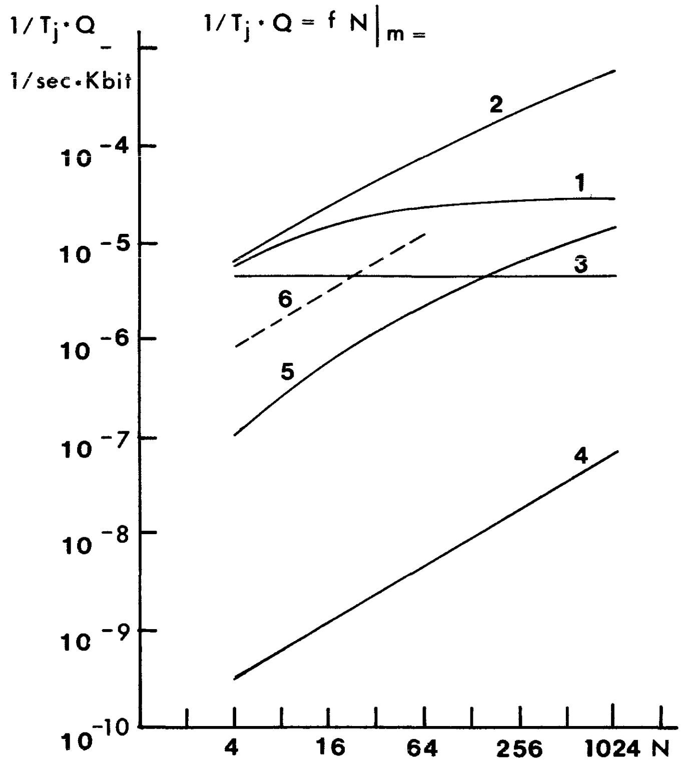
1—1-D JP
2—2-D JP
3—"systolic" array
4—DBC JP
5—DBC JP input values
output pointers
6—DBC JP, based on the A, B memories sizes given in reference 9

Figure 12—Number of memory cells vs. size of relations

We note that the complexity estimation for the DBC join processor does not include the A memory overflow capacity. The DBC join processor design is so costly that even if we eliminate the cost of the C memory, it still has the highest hardware complexity due to the use of associative memories and hashing.

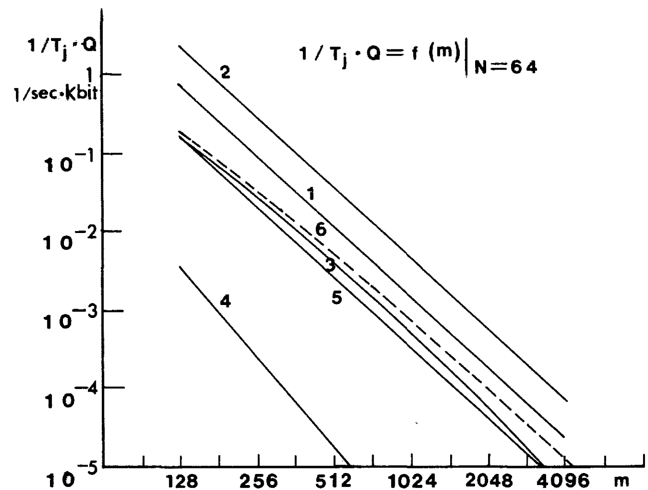
c) *Performance/Cost Ratio*

Using the equations developed above, we can compare the performance/cost ratio of the four design approaches. The calculated results are shown in Figures 13 and 14 with constant N or m values respectively. The result of computation indicates that the two-dimensional join processor array has the most favorable performance/cost characteristics. The performance/cost indices for the four designs, under the same parameters of $N = 64$ and $m = 1024$, are listed in Table I.



1—1-D JP
2—2-D JP
3—"systolic" array
4—DBC JP
5—DBC JP, input values
output pointers

Figure 13—Performance/cost ratio vs. number of parallel processors (m = const)



1—1-D JP
2—2-D JP
3—"systolic" array
4—DBC JP
5—DBC JP, using values and pointers
6—DBC JP, based on the processing time and A, B memories sizes given in reference 9

Figure 14—Performance/cost ratio vs. size of relations (N = const)

Table I—Performance/cost indices of join processors

Approach	Pfm/cost (1/sec·Kbit)
2-D Array	$4850 \cdot 10^{-6}$
1-D Array	$1440 \cdot 10^{-6}$
systolic	$495 \cdot 10^{-6}$
DBC JP	$1.2 \cdot 10^{-6}$
DBC JP*	$330 \cdot 10^{-6}$

*Using values and pointers instead of entire records.

5. CONCLUSIONS

In this paper we have analyzed several approaches for the join processor design. The proposed join processors can be classified into three categories: (1) one-dimensional array with pipelining, (2) one-dimensional array with broadcasting, and (3) two-dimensional array. Our analysis shows that the two-dimensional array approach has significant advantages in terms of both processing speed and hardware complexity. In addition, the two-dimensional array processor has a simple organization. The regularity of this design also makes it suitable for VLSI implementation. A small-scale experimental VLSI chip has already been implemented. The experimentation of a more complete join processor array is being planned.

The join processor reported here is an integral part of a database machine which also contains a hardware *data filter* that performs selection functions, and a database operating system. Analysis and performance of the experimental machine will be reported elsewhere.

APPENDIX A

The join processor requires initialization before it is started. The system must transfer the following parameters to the corresponding internal registers:

1. Starting address of the selected file R in system buffer area, *HAR*
2. Starting address of the selected file S in system buffer area, *HAS*
3. Number of records in file *R, #R*
4. Number of records in file *S, #S*
5. Length of R-record, *LRR,* in bytes or words
6. Length of S-record, *LSR*, in bytes or words
7. The relational join operator, *op, op*
8. Length of selected R-value, *LVR*
9. Length of selected S-value, *LVS*
10. Partition constants for R and S, i.e., x and y

All the initial parameters are transmitted through the data bus under the control of the system controller.

Suppose file R and file S contain m and n records respectively. The join processor performs $[m/x]^*[n/y]$ iterations of the basic value-matching process.

The algorithm shown in Appendix Figure 1 describes the iterations in more detail.

```
PROCEDURE JOIN ( rvalue, svalue )

    CONST INTEGER: m, n, x, y;
    VAR   INTEGER: i, j, countf, countv, countr, counts, rvalue, svalue;
    VAR   BOOLEAN: start, stop, F, F(i), error, zero;
    MEMO  REGISTER: LRR, LSR, LVR, LVS, HAR, HAS, RAddr, SAddr, OP, C1,
                    C2, #R, #S, X, Y;
    MEMO  BUFFER: B(i), RB(i), SB(j);

    /* initialization */
    LRR := length of R-record;
    LSR := length of S-record;
    HAR := head address of file R in system buffer area;
    HAS := head address of file S in system buffer area;
    OP  := comparison operator code;
    LVR := length of selected R-value; /* LVR must equal to LVS,
                                          otherwise, it is recognized
                                          as error */
    LVS := length of selected S-value;
    #R  := number of records in R;
    #S  := number of records in S;
    X   := x;
    Y   := y;

    BEGIN /* start the join process */
        RAddr := HAR;
        SAddr := HAS;
        countv:= ~LVR + 1;   /* complement of LVR */
        countr:= ~[m/x] + 1;
        counts:= ~[n/y] + 1;

        IF error = true THEN stop ELSE
        WHILE countr != 0 DO /* (m/x) R-iteration control */
            BEGIN /* the S-iteration */
                WHILE counts != 0 DO /* (n/y) S-iteration control */
                    countf := 0;
                    WHILE countv != 0 DO
                        BEGIN /* enter the 1st phase, broadcasting the R-
                                 and S-values byte- or word-wise */

                            C1 := BR;
                            C2 := BS;
                            countv := countv - 1;
                        END
                    F := C1 op C2; /* set the flag-matrix according to the
                                      result of comparison */
                    countv := ~LVR + 1 /* recover the value length counter */
                    IF F = true THEN /* check the entire flag-matrix */
                                     /* enter the 2nd phase: check each row of
                                        the flag-matrix, performing and storing
                                        the addresses of the matched records */

                        IF F(i) = true THEN
                            BEGIN /* check flag and store address iteration */
                                B(i) := RAddr(i);
                                FOR countf = 0 UNTIL y - 1
                                    /* check the flag-matrix one column at a
                                       time, store the addresses of the matched
                                       S-records */


                                    IF F(i,j) != 0 THEN
                                        B(i) := SAddr;
                                    countf := countf + 1;
                                    SAddr := SAddr + LSR;
                            END
                    counts := counts - y;
            END
        RAddr(1) := RAddr(y) + LRR; /* produce the next set of R-record
                                       addresses */
        countr := countr - x;
    END.
```

Appendix Figure 1

REFERENCES

1. Babb, E. "Implementing a Relational Database by Means of Specialized Hardware." *ACM TODS,* Vol. 4, No. 1, Mar. 79, pp. 1–29.
2. Banerjee, J., and D. K. Hsiao. "Concepts and Capabilities of a Database Computer." *ACM Trans. on Database Sys.*, Vol. 3, No. 4, Dec. 1978.
3. DeWitt, D. J. "DIRECT—A Multiprocessor Organization for Supporting Relational Data Bases Management Systems." *IEEE Trans. on Computers,* Vol. C-28, No. 6, June 1979.
4. DeWitt, D. J., and W. I. Madison. "A Performance Evaluation of Database Machine Architectures." *7th International Conference on VLDB,* Cannes, France, Sept. 9–11, 1981.
5. Kung, H. T., and P. L. Lehman. "Systolic (VLSI) Arrays for Relational Database Operations." *ACM SIGMOD,* 1980.
6. Leilish, H. O., G. Stiege, and H. Ch. Zeidler. "A Search Processor for Database Management Systems." *IEEE,* 1978.
7. Luque, E., J. J. Ruz, A. Ripell, and A. Bautista. "Database Concurrent Processor." *IEEE,* 1979.
8. V. A. J. Maller. "The Content Addressable File Store—CAFS." *ICL Tech J.*, Nov. 1979, pp. 265–279. February 1981.
9. Menon, M. J., and David K. Hsiao. "Design and Analysis of a Relational Join Operation for VLSI." *Report,* Dept. of Computer and Information Science, The Ohio State University, Feb. 1981. Vol. 1, No. 3, September 1976.
10. Stanley, Y., W. Su et al. "Database Machines and Some Issues on DBMS Standards." *NCC,* 1980.

11. Sheng, Y. Z., F. Tong, and S. B. Yao. "Data Filter—A Relational Selection Processor." *Tech. Report,* Database Research Laboratory, University of Maryland, College Park, MD 20742, October 1981.
12. Tong, F., and S. B. Yao. "Design of a Two-Dimensional Join Processor Array." *6-th Workshop on Computer Architecture for Non-Numerical Processing,* Hyeres, France, June 1981.
13. Tong, F., and S. B. Yao. "Logical Organization of Two-Dimensional Join Processor Matrix." *Technical Report,* Database Research Laboratory, Univ. of Maryland, College Park, MD 20742, 1981.
14. Wah, B. W., and S. B. Yao. "DIALOG—A Distributed Processor Organization for Database Machines." *AFIPS Press,* Vol. 49, 1980.
15. Yao, S. B. "Optimization of Query Evaluation Algorithms." *ACM TODS,* 4, 2 (June 1979).
16. Wong, E., and K. Yousseffi. "Decomposition—A Strategy for Query Processing." *ACM Transactions, Database Systems I,* 3 (1976), pp. 223–241.
17. Blasgen, M. W., and K. P. Eswaran. "Storage Access in Relational Databases." *IBM Systems Journal,* 4 (1977), pp. 363–377.

Evaluating database management systems

by EDWARD DAVIDSON
General Electric Company
Philadelphia, Pennsylvania

ABSTRACT

This report documents a methodology developed and used for the evaluation and selection of database management software. The basic methodology can be used in the evaluation of other types of software. The report describes the step-by-step process and provides an extensive discussion of the definition of evaluation and selection criteria. The report will be of most use to first-time evaluators, but may also be of use to more experienced personnel.

BACKGROUND AND SCOPE

In the course of conducting an evaluation of several candidate database management software packages with the objective of selecting one of them for a particular application, it was noted that there were no clear and concise guides available to assist novice evaluators in this type of task. With the rapid changes in computer technology and the decline of hardware costs, more and more applications will be turning to database management systems. It was felt that a guide to the evaluation of database management systems would be of great use to the many organizations that will be faced with the task of selecting a software package for a particular environment or application.

This guide has been written for first-time evaluators; however, more experienced personnel may find some useful information presented. It is not meant to be the sole source of information needed to conduct thorough evaluations. Rather, it presents a sequence of events, suggestions of sources of information, and a set of evaluation criteria that should be augmented or amended as necessary to suit particular environments or applications. In some cases, the evaluation criteria present alternative conditions and possible implications of each alternative.

ENVIRONMENT IDENTIFICATION

Prior to selecting candidate database management systems for evaluation, a determination must be made as to the operating environment in which the software will be used. This environment includes both the computer hardware and the specific operating system installed or to be installed. Consideration must also be taken of possible subsequent installations of the system (i.e., second-site or distributed processing environments). Will all installations involve the same hardware and operating system? Some database management software packages, especially those marketed by the hardware vendors themselves, may be limited to one specific type of hardware or operating system, while other packages will run on a wide variety of systems. Although the single-environment package may solve the current need, it may limit future growth or dispersion of the application.

CANDIDATE SELECTION

Once the operating environment has been established, it is possible to select one or more database management software packages for evaluation. There are several sources of information that can be used to determine which packages are available for the appropriate operating environment.

1. *Datamation* is a monthly magazine on data processing that contains excellent articles on database technology as well as other computer-related topics. Each issue has a particular theme and some issues are devoted to database management systems. This publication frequently has product surveys (software in general, including database management systems, or specific types of software only, such as only database management systems). About two years' worth of the publication should be scanned for surveys and other pertinent articles. This magazine is popular with data processing personnel and should be widely available.
2. *Computerworld* is a weekly newspaper of data-processing information. In addition to product advertising, this publication has articles describing new product releases and enhancements or problems affecting other products as well as articles of general interest related to database technology.
3. *Datapro Reports,* found in many data-processing shops or technical reference libraries, contain articles on database techniques and general database topics, as well as articles on or reviews of specific systems. In addition to synopsis reviews on specific software packages, there are charts of comparisons between several similar systems. The product reviews are the results of *Datapro* surveys of product users and, depending on the number of respondees, pertinent or meaningful information may or may not be complete. Not all products are included; however, most of the popular, established packages are represented.
4. *Interface* is a quarterly publication, primarily available, and used, for vendor advertising. There are several editions, one of which contains database management systems. There is also an edition devoted to minicomputer software. With this publication, most of the popular database management systems can be compared briefly on relative features and price, vendors' addresses and other contact information can be determined, and promotional literature can be requested through reader-service cards.
5. *Micro-Mini Systems* is a monthly publication of interest to micro and minicomputer environments. During the past year, they have reviewed a number of database management systems for the mini and micro environments.

In addition to utilizing the publications described above, information on possible candidates may be provided by the personal experiences of the organization's staff members or from the hardware vendors, who may be able to provide a list of software which runs on their equipment.

Upon compilation of the list of candidate software, each software vendor should be requested to send its promotional literature, which can give an overview of the features of the package. In addition, vendors may offer, and should be requested, to send a copy of the appropriate user's and/or technical manuals, which can be used to evaluate exactly how easy or difficult it is to use the package, how specific features work, the quality and depth of the documentation (how easy will it be for data processing personnel to use the manuals effectively, how much training will be required, and so on). Some vendors may submit or loan the material free of charge and some may invoice the organization for the manuals. However, there should be no need to purchase any materials until the final selection is made. Upon completion of the evaluation, all unneeded documentation should be returned to the vendors concerned, suitable for resale, or else purchased from the vendor.

Vendors should be asked to supply a copy of the latest annual report (if they are publicly held companies) or a description of the company with some financial information that can be used to evaluate the growth and stability of the vendor. If it is felt to be important, the vendor may be asked whether there is any litigation in progress that might affect the product under evaluation.

DETERMINING SOFTWARE REQUIREMENTS

Before the evaluation process can begin, operational and functional requirements that the database management software must satisfy should be documented. The requirements will be used to select, categorize, and order according to priority the specific evaluation criteria that will be used to review each candidate package.

The requirements may cover one or more of the following categories (some specific requirements for a particular application may not be covered in this document but should be included as necessary):

1. Portability—on what computer hardware and under what operating systems must the software run? What is the timeframe for implementation on different hardware or operating systems? On what other hardware/operating system might the software run in the future?
2. Flexibility—what flexibility features must the software have? Integrated data dictionary? Integrated query language? Dynamic generation or deletion of keys, user views, access privileges? Variable-length records? Must the software support multiple users at one time? Multiple databases? Distributed processing? Batch and online users concurrently?
3. Security—what level of data security is required? File? Field? Will it be necessary to restrict access to specific data according to the values of one or more fields (i.e., user A is limited to data for his department only)? Some database management systems provide that level of security—"security by value"—in the definition of user views, while other packages restrict access to the field level, so that if a user is granted access to a field, he has access to all values of the field. Certain applications, such as personnel or payroll, may need to restrict some users to specific portions of the database. Most applications will not need this level of security.
4. Recoverability—is it required, or desired, that the database management software provide the capability for transaction journaling, before-image or after-image journaling, or system checkpointing? Must the software provide automatic recovery (automatically restore the database to a certain level based on a checkpoint file or journal file when the system is brought back up) or is it acceptable to require the intervention of programmers or operators to restore the database? Some DBMS software packages do not provide recovery features at all, relying only on system or user recovery procedures.
5. High Order Language (HOL) Interface—will the application require access to the database from HOLs (i.e., COBOL, FORTRAN, PL-1, C, etc.)? Which languages? How complex will the access keys be (i.e., multiple fields, etc)? The HOL call procedures for some database management systems are more restrictive or less flexible than others. Some applications may have access requirements that include accessing data in more than one file at one time.
6. Performance—is software response time critical? What is the maximum desired response time for a query? Update? Will the application be primarily used for database maintenance (update, preplanned reporting) or will it be used mostly for ad hoc queries or random retrievals? Is data storage critical? Some database software systems will make optimum use of online storage by compressing multiple blanks, zeros, or empty fields. For extremely large databases, depending on the operating environment, space utilization may be critical.

After the specific requirements have been determined, they should be prioritized according to their importance to the success of the application. If some requirements have alternative implementation possibilities (i.e., good recovery procedures are required, preferably automatic but we could live with semi-automatic procedures as long as ...), the alternatives should be prioritized.

When the prioritized requirements have been documented, they should be used to develop a prioritized list of criteria which will be used in the evaluation of the candidate systems. One requirement may result in several specific criteria. The following section lists some possible criteria with potential implications of alternative implementations.

EVALUATION CRITERIA

The items listed below represent possible areas of concern in the evaluation of database management systems. These criteria are not intended to be a complete list and are not presented in any particular order. These items, and any others that organization staff members may suggest, and any that may be imposed directly by the requirements, should be reviewed for applicability to the system requirements. Where alternative implementations are possible, a determination should be made as to which implementation is appropriate for

the application, or the order of preference if more than one might be appropriate. Items that have no bearing on the specific requirements may be discarded, or may be used to futher evaluate or compare several possible candidates that satisfy the required criteria. These optional criteria might be concerned with vendor reputation and support, level and quality of documentation, or features that are not currently required but may be of concern for future, as yet unplanned, application development activities.

Performance Features

1. Does the software optimize multikey retrievals? Some database management software will determine the shortest path to the data described by the combined keys by analyzing the number of records qualifying for each key and selecting the shortest access parth. Other software packages will analyze records based on the order of the keys expressed in the query or call statement. Depending on the number of records qualifying for each key, one technique may result in better performance than the other. These concerns apply mainly to relational database management systems. Network-type software depends on imbedded pointers in the data, although some network systems are developing relational-type query languages that allow some level of optimization.
2. Does the software run in the "native" mode or in the "compatibility" mode on the particular computer system? Native mode processing is the most efficient, taking advantage of the hardware and system software features of the host system. It is usually written or compiled in the assembler language of the host system. Under "compatibility" mode the host system emulates another operating system in order to run the software. Emulation can have a significant detrimental impact on performance.
3. If two or more applications or users attempt to access the same record at exactly the same time, what level or type of lockout occurs, if any? Can two or more users access the same record simultaneously for "read-only" or "browsing"? Some software allows only one user access to a record, regardless of the type of operation performed. Other packages will allow a user to hold a record during an update operation to prevent any other user's accessing it for updating but will permit other users to read or browse the held record. In applications that may have many simultaneous access attempts for browsing and updating the same record, this could affect the response time for some of the users. Some vendors use the term "multi-threading" to describe this feature.
4. Does the database management software allow simultaneous batch and interactive applications? Is there a software limit to the number of simultaneous users or applications?
5. What are the software limitations, if any, with regard to the number of databases supported by one copy of the software? Are there any limitations to number of files per database, number of records per file, number of bytes per record, number of bytes per database, and so on? Certain applications with large data requirements may find specific software packages too restrictive or limited with regard to the amount of data that can be handled.
6. Will the vendor provide a copy of the database management software, or allow access to a copy, for benchmarking? Will there be a charge for this? Will the vendor provide any assistance in the benchmarking set-up or execution? If benchmarking of the software is not possible or feasible for an organization, attempts should be made to obtain benchmark results or reports from other customers who have performed benchmarks. Although the specific controls of the current evaluators cannot be tested, pertinent performance information may be derived from these reports.
7. Must the database be taken off-line or out of productive use to perform any or all database administration functions such as defining a new field, expanding a field's size, deleting a field, assigning a new key, or granting or revoking access provileges? Some database management software allow many of the database administration functions to be dynamic and have no impact on executing applications, except for those directly affected by the specific changes. Other packages require database utility programs to run with exclusive access and control of the database, some requiring the database to be dumped and restored prior to restarting the applications. This could have significant impact on environments where there is constant, heavy activity on the system. Alternative solutions to this problem could be to have database modifications performed during periods of little or no application usage (nights, weekends, or holidays) or prescheduled downtime for database maintenance.
8. If the software provides for the compression/decompression of repeated blanks, zeroes, and null-value fields, is the feature optional? How is the performance affected by invoking the option?

Data Dictionary

1. Does the software have an integrated data dictionary? If so, is the data dictionary active or passive? Most database management systems provide some sort of data dictionary. The more sophisticated packages have dictionaries that control all access to the data for the users and database administrator. This arrangement allows all access to a particular piece of data to view the data consistently, eliminates the need to have data definition statements in each program, standardizes the names of each field, and controls access to the database, file, and/or field level. Some of these also are used to store precoded queries and user views. Passive data dictionaries record some information about the data for documentation purposes but exert no control over access to the data.
2. Can data dictionary information be modified without affecting active users? Some data dictionary software

allow fields to be added to the file or database definition, or information about currently defined fields to be modified, without restricting the use of the database to active users. The changes are made dynamically. Other data dictionary software provides database maintenance utilities that must be run with complete control of the database. Some database management systems require that the database be unloaded and reloaded after certain types of changes in or additions to the data dictionary. Other packages can handle changes with little or no impact on the physical database or existing data.

3. Is the data dictionary used to control the compression of multiple blanks, zeroes, and/or null-value fields? Some database management systems that offer data-compression features allow the data-compression option at the field level to be controlled by parameters in the data dictionary. In other software, compression is mandatory for all data or not allowed at all.

Recovery Features

1. Does the database management software contain any integrated recovery features at all? Some database software rely entirely on the operating system's backup and recovery features to protect the database. The more sophisticated systems provide some level of checkpoints, transaction journaling, before-image journaling, or after-image journaling, or a combination of these options.
2. If recovery is provided for, are the recovery procedures automatic? Following a crash, some database software will automatically restore the database through the last successfully completed transaction as soon as the system is brought back up. Other database systems will require operator or programmer action to restore the database, possibly application by application, from journal and checkpoint files. If specific personnel are required to restore the database, delays to production could be experienced if the personnel are not immediately available following a crash.

Flexibility Features

1. Does the software support concurrent processing by multiple interactive users or by interactive and batch applications? Some packages do not allow more than one application or user at a time. Others may have a limit to the number of active users or applications that can process concurrently. The design and potential usage of an application may require the capability to support several, if not many, concurrent users.
2. What is the format of the High-Order Language (HOL) interface ("call") and what languages does the software interface with? Does the software provide preprocessors for the applicable languages to translate and create the specific calls to imbed in the high-order language programs or does the programmer have to code each call directly in the program? Different packages provide different facilities. The most sophisticated packages allow the coding of English-like statements in the application program with subsequent preprocessors converting the query-type statements into the appropriate calls. This permits more productive coding by the programmers. Other packages allow query-type statements to be passed in call statements. The remainder of the packages require some type of control blocks to be defined, and control block parameters to be coded by the programmer. Of the last category, some packages will allow more complex key structures to be used in the access calls than in other database systems. As it can be seen, the method and format of the call structure may have an impact on the complexity, flexibility, and efficiency of the calls.
3. Does the software allow complex or concatenated keys in queries and HOL calls? Some packages allow the use of complex keys in both interactive queries and through calls from HOL programs. Other software packages are very limited in the access keys allowed in either query or call modes, although usually the call mode is the more restrictive. The ability to code very explicit access keys may be important in some applications. The use of complex keys may be limited by the database architecture employed by specific packages; many hierarchical and network database systems use simpler keys requiring the programmers to "navigate" the database for the desired data.
4. Can key fields and user views be created and deleted dynamically? Can new data items be defined to the database dynamically? In environments where the data requirements are continually changing, it may be advantageous to be able to expand the database, create and delete keys, and create, modify, and delete user views without having to restrict the use of the database or require the database to be unloaded and reloaded. Other applications, where the data requirements are static, may not need these features. Some of the newer packages allow for these dynamic operations and some of the more established packages are developing these capabilities.
5. Does the software permit the accessing of multiple databases and/or files in one access attempt (commonly called "joining")? Some software can only access one database at a time. Some packages restrict queries and calls to one database file at a time, requiring multiple queries or calls to satisfy complex retrieval requests. Other packages permit multiple-file access in a single query or call, simplifying the access procedure and increasing the flexibility of the system.
6. On what hardware does the software run? Some packages, especially those developed or marketed by a specific hardware vendor, may be restricted to the vendor's own hardware. Packages developed by software houses that do not construct hardware may run on almost every computer system available in certain classes of system. The current and future hardware configurations on which an application may run place constraints on the specific packages that should be considered for a specific

evaluation. A related question is whether the software runs in native mode or compatibility (emulation) mode on the hardware on which it runs.

7. Does the software have currently, or have planned for future implementation, features that will enhance future application development by the organization? Has the vendor demonstrated a capacity for, or is the vendor committed to, keeping the product current with advances in database technology? Some of the database management software packages stabilize as they mature, and although they may satisfy current requirements they may not be able to support substantial future growth. Other products, as they mature, implement features that reflect the advances in technology since the original implementation. Organizations that expect substantial increases in the number, size, and complexity of database applications using the same software, may want to consider the vendor's future plans for the software.
8. What "user-friendly" features does the software provide to reduce the application-development effort and permit less skilled and nontechnical personnel to make effective use of the data? Some features to be considered include screen generation facilities, integrated query languages, integrated report writers, "HELP" commands accessible interactively, and meaningful error and diagnostic messages. While these features may not be used for initial selection and rejection of candidate packages, they may be considered when comparing closely similar systems to determine which is more "friendly" and easy to use.

Security Features

1. At what level is data security provided by the software? Some products provide security to the database of file level only. If you have access to the file, you have access to all data in the file. Other products provide access restrictions to the field level through user views. Still other products can provide security based on the value of specific fields. This is called "security by value." An example would be to restrict a user to viewing data for only one department, or to prevent a manager from viewing data for employees earning more than he or she does. Although not widely needed, this feature may be important in some applications.
2. Can access privileges be granted and/or revoked dynamically? In some environments, it may be advantageous to be able to modify access privileges instantaneously, without interfering with active users. Some applications may require access authorizations to be established or deleted on short notice or on a frequent basis. For those environments, this feature would be useful.

Vendor Support and Reputation

The items listed below, while most likely not constituting reasons to initially select or reject a particular candidate package, may be used to rank otherwise similar packages and can be used to identify potential areas of concern for products that otherwise fully meet all other selection criteria.

1. Does the vendor have a good reputation for responding to requests for information, dealing with reported problems with the software, and providing technical support of the software? This information can be obtained by interviews with current or past users of the software or through software surveys conducted by data-processing periodicals such as *Datapro, Datamation,* and *Computerworld.*
2. Are current users of the software satisfied with the performance of the software? Are they satisfied enough with the product to acquire other software produced by the same vendor? During interviews with the current users, comments may be made or solicited that will indicate whether the product is worth recommending to others, whether the product has lived up to its expectations, and whether the customer has enough faith in the quality of the vendor's product to use other software from the same vendor. Some users, after acquiring a product, find out that it does not function as expected but have gone too far into development to discontinue use of the product; they have to modify their design to fit the product, which is not the ideal way to go. An isolated complaint should not cause undue alarm, but similar comments from several users about the same product should be considered significant.
3. Does the vendor provide a "hot line" for reporting problems with the product operation? Many vendors provide this service. Some have toll-free numbers, some operate twenty-four hours a day, including weekends. For applications that operate during nonbusiness hours, a twenty-four hour hot line may be an important consideration.
4. Are the software technical and user manuals clearly written, with adequate examples and illustrations? One significant area where products vary is in the quality of the documentation. Some otherwise excellent products have poorly written documentation, which greatly increases the effort required to learn and use the software effectively. Many application-development staffs do not have large training budgets and work under tight time constraints. To maximize the development effort, the staff should be able to quickly find and understand the information needed in the documentation, without having to constantly call the vendor's technical staff for interpretation or explanation. Sample programs and routines, as well as examples of individual instructions, provided where the instructions are described, greatly improve the usefulness of the documentation.
5. Does the vendor supply small test databases with the installation of the product? Some vendors supply one or several test files with a small number of records to be used by the acquiring organization for training and experimenting with the data without having to define and load its own test data. This arrangement greatly speeds up training in the use of the product and allows a rapid review and evaluation of many software features and procedures. Some vendors also provide canned routines

or scenarios to be used with the test database to demonstrate features.

6. Does the vendor have a procedure for notifying all users, on a regular basis, of reported problems and solutions or of the status of unresolved items? Some vendors conscientiously keep all users informed about the problem status of their product. Other vendors will respond to a reported problem directly to the reporting organization so that many organizations may experience the same problem without knowing that the problem may be global in nature and without knowing a workaround that might avoid the problem altogether.
7. Does the vendor provide installation support? If so, is it necessary, and at what cost? Some database management systems can be installed by the user by simply loading one or more files from a tape supplied by the vendor. Other packages require more technical support to install. Some vendor installation support is provided in the cost of the package, and some is available on a fee basis on top of the purchase price.
8. Does the vendor supply, or provide for, customer education and training in the use of the product? Is there any other training available that is related to the product? Some vendors have excellent training and education programs, not only in the elementary use of the software but in advanced techniques, internals, design, and so on. For other software, all education beyond a brief introduction to the software comes from manuals on a do-it-yourself basis. For organizations with small staffs and a need for rapid productivity, the customer training programs may be a critical factor.
9. Are enhancements and modifications to the products, either for product error correction or for the addition of new features, accomplished by user-installed zaps or patches, or by a complete replacement of the software? There is potential in user-installed zaps, especially where routines have to be coded in, for errors to be made because of poor documentation or human error. There is faster installation and less likelihood for error if a whole module is replaced with a new copy provided on tape directly from the vendor.
10. What is the cost of the various acquisition arrangements (buy/lease)? Are quantity discounts available? Are execute-only copies available? The acquiring organization should review the prices for the various options with regard to the features offered by the product, the support available, and the reputation of the vendor and product. A more expensive product is not necessarily a better product.
11. What is the cost of a maintenance agreement? What is the term of the maintenance agreement? Exactly what is provided for in the maintenance agreement? Different vendors provided different periods and levels of support.
12. Does the vendor provide additional features and capabilities to current users as enhancements to the existing software under the maintenance agreement, or are the features released as new products at additional cost? Some vendors are beginning to release additional features as add-on, extra-cost products. Before the purchase/lease/maintenance agreements are signed, it should be fully understood what the vendor's position is on this.
13. Is there any indication of problems, including litigation, that might affect future use of the product? This may be discernible from the company's annual report or may be asked directly of the vendor. Make sure you get all of the details, including the vendor's position, and follow up where possible to check the results of the action. It might be wise to have an alternative plan in the event the software is affected after acquisition.

EVALUATING CANDIDATE SYSTEMS

When the system requirements and evaluation criteria have been defined and prioritized, and upon receipt of the literature and documentation requested of the vendors, the evaluation and review of specific products can begin.

It might be advantageous to develop some form of documentation that summarizes the specific requirements, evaluation criteria, and features desired of the software so that appropriate notes and comments can be made as information on each product's capabilities becomes known. Also, a form may be developed for use when contacting users of each system, specifying questions or points of concern to be discussed with the user, and providing a place for recording the user's responses. These types of forms will standardize the information gathered during the evaluation and simplify the subsequent decision. Examples of these types of forms are shown in Figures 1, 2, and 3.

There is no specific method of evaluating the vendor literature and documentation. Since vendors do not stress the features or capabilities they do not provide, it will be necessary to identify those features and capabilities they do provide and infer that others are not provided. If in doubt, contact the vendor representative. Many vendors provide a brief technical summary of the software features in addition to the more detailed technical material. The summary will indicate what the software can do, but may not explain how it does it. The summary is frequently a sales brochure; it should not be used as the sole source of information about a product unless the vendor is unable or unwilling to provide additional information, in which case that product should be eliminated from further consideration.

The documentation for each product should be reviewed for information relating to the specific requirements or criteria. This review may be accomplished in several passes of the literature, each delving deeper and deeper into the technical aspects of the data structure, commands, command sequence and structure, utility functions and features, and so on.

The first pass should attempt to identify those products that clearly are lacking in one or more criteria or requirements. A decision can be made to eliminate those products entirely or make allowances or exceptions for the lack of compliance. The object of these reviews is to gradually weed out software products that clearly do not satisfy the requirements. It may be that none of the candidate systems can match all of the requirements, and a decision may have to be made to relax or revise the requirements, develop an in-house system entirely,

USER PRODUCT CRITIQUE (Sample Form)

PRODUCT REVIEW

Product Name:______
Application Uses:______
Competive Products Considered:______
Reason for Selection:______
Has Product Performed as Expected:______ Comments:______
Comments on Ease of Use:______
Performance:______
Installation/Loading:______
Utility Programs/Functions:______
Data Dictionary:______
Restart/Recovery/Security:______
Update Transactions:______
Retrieval Transactions:______
Data Base Reorganization:______
Data Base Capacity:______
Technical Documentation:______
User Documentation:______
Additional Comments:______

______ Would you Recommend Product:______

VENDOR REVIEW

Vendor Name:______
Has Vendor Provided SATISFACTORY Installation Support:______
Timely Technical Support:______
Responsiveness to Your Requests:______
Training/Education:______
Documentation:______
Additional Comments:______

Would You Purchase Additional Software from this Vendor:______

Reporting Company:______
Contact:______ Phone:______
Environment:______ Date Package Acquired:______

Figure 1—User product critique (sample form)

PRODUCT FEATURES SUMMARY (Sample Form)

SYSTEM NAME:

HARDWARE:	COST:	VENDOR:
	SINGLE SITE/CENTRAL CONTROL MULTI-SITE MAINTENANCE CONTRACT	REP: TEL:

DATA DICTIONARY:

RESOURCE REQMTS:

HOST LANGUAGE INTERFACE(S):	DATA BASE LOADING PROCEDURE
TRAINING/EDUCATION AVAIL:	
SYSTEM DOCUMENTATION/MANUALS:	
SYSTEM FEATURES: __ DATA COMPRESSION __ USER QUERY LANGUAGE __ USER APPLICATION LANGUAGE __ TRANSACTION LOGGING __ SECURITY __ FIELD __ FILE __ BACK-UP/RECOVERY FEATURES __ RESTART/RESTORE FEATURES __ UTILITIES __ INTERACTIVE PROCESSING __ BATCH PROCESSING __ SOURCE CODE PROVIDED/AVAILABLE	REORGANIZATION PROCEDURE ADDING FIELDS

EASE OF USE:

CAPACITY: ______ FILES ______ RECORDS ______ BYTES ______ KEYS/DESCRIPTORS	OPTIMUM FILE ORGANIZATION: ALTERNATE ORGANIZATIONS: FILE ACCESS METHOD:

OTHER USERS: ______ NUMBER OF INSTALLATIONS
REF1______
REF2______

VENDOR SUPPORT CRITIQUE:

Figure 2—Product feature summary (sample form)

or develop a workaround for those requirements which cannot be satisfied by the software.

Through the increasingly detailed review of the documentation, it may be possible to eliminate some products and group others together by the features provided or techniques employed. It is then necessary to evaluate the competing products on how they handle certain conditions or situations and on their vendor support, quality of documentation, and other subjective criteria. Where one product does not surpass all the others on these subjective criteria, a decision should be made about how to rate the criteria.

Unless one product clearly surpasses all others in the majority of the criteria, there may be several candidates remaining after this review. At this point, it may be advantageous to arrange for demonstrations of the software by the vendor, or, if it can be arranged, for benchmarking the candidates.

Although benchmarking is the best method, it may not be feasible. Some vendors will provide a copy of their software free of charge for a limited time for this purpose, while other vendors may impose a fee. Other vendors may give the organization access to the software at the vendor's site or at another customer's site at a time when the software is not in active use by other users. One must define the benchmark criteria, develop and package tests for use on the benchmark system, and schedule the time and personnel required for the tests. In some cases the vendor will conduct or assist in the testing. The results of the benchmark tests can be used to rank the candidates tested.

If benchmarking is not feasible, owing to time, cost, or personnel availability, it may be possible to evaluate the software by attending a demonstration of the software by the vendor. Although not as precise as benchmarking, the demonstrations may illustrate some interesting features of, or facts about, the system. What is the response time exhibited during the canned demonstrations? Although subject to the type, size, and load of the machine during the test, certain trends may be noticed. Did the demonstrator need reference manuals in order to conduct the demonstration? If so, it may indicate that the system is complex or difficult to use, since usually the vendor puts its best available personnel on the system to make a good impression.

In addition to observing the canned demonstration, it would be advantageous to be prepared with a small database definition (perhaps five fields) that you can ask to have defined and installed while you watch. This will let you see exactly what is required to accomplish the task, and will allow you to create some of your own data to test on the system. The same database definition can be used with each vendor and a measure of comparative complexity may be determined.

The vendors of the leading candidate packages should be

EVALUATION CRITERIA BY PRODUCT-SUMMARY (Sample Form)

PRODUCT NAME ____________________

EVALUATION CRITERIA	PRIORITY	ACCEPTABLE/ AVAILABLE	UNACCEPTABLE/ UNAVAILABLE	PLANNED FOR FUTURE IMPLE.
HIGH-ORDER LANGUAGE INTERFACE				
INTEGRATED DATA DICTIONARY				
INTEGRATED QUERY LANGUAGE				
DYNAMIC KEY DEFINITION				
DYNAMIC ADDITION OF DATA ITEMS				
OPTIMIZED DATA STORAGE (COMPRESSION)				
OPTIMIZED MULTI-KEY ACCESS FEATURES				
MULTI-THREADED ACCESS				
FULLY AUTOMATED RESTART/RECOVERY FEATURES				
HIGH QUALITY DOCUMENTATION (TECHNICAL/USER)				
GOOD DATA SECURITY FEATURES				
OPERATES IN CURRENT AND FUTURE ENVIRONMENTS				
HIGH DEGREE OF DATA INDEPENDENCE				
DBA UTILITIES				
HOT-LINE FOR PROBLEM-REPORTING				
COSTS FOR LEASE/MAINTENANCE				

2293a/1058a/tmg

Figure 3—Evaluation criteria by product summary

asked to provide the names and telephone numbers of several customers of their products. Some vendors may be unwilling or unable (because of customer requests for confidentiality) to divulge the information. In any case, remember that the names supplied will most likely be those of their most satisfied customers and meaningful or pertinent negative comments may not be made. These users should be asked to comment on the performance of the product, whether the product has performed as expected, the effort required and problems experienced in installing the product, feature performance, and the support of the vendor in answering questions and resolving problems. Even the most satisfied customer may have some negative comments about specific features. Similar comments from several customers may indicate inherent problems, which can then be discussed with the vendor. The customers' responses should be documented for future reference.

At this point, sufficient information should be available for the selection of one database management system for the application. If selection is still not possible, the evaluating criteria should be reviewed, made more explicit, or expanded to allow for further refinement and elimination of candidates until one product is considered suitable for acquisition. The Evaluation Procedure Flow is illustrated in Figure 4.

POST-SELECTION TASKS

Following the selection of a particular product, materials developed or acquired during the evaluation process should be reviewed, discarded, or filed, as appropriate, for further reference. If possible, the results of the study and supporting materials should be centrally filed and made available to other internal users as the need arises. *Any materials, documentation, and manuals on temporary loan from a vendor should be returned in a condition suitable for resale, or should be purchased from the vendor.*

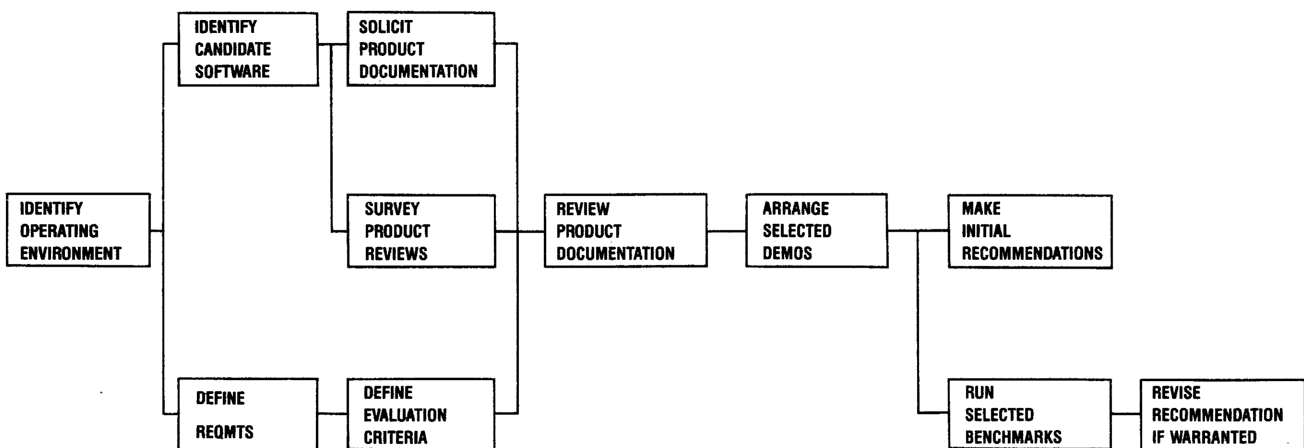


Figure 4—Evaluation procedure flow

Performance study of a dual CDC Cyber 170/750 system*

by M. SEETHA LAKSHMI and TOM W. KELLER
The University of Texas at Austin
Austin, Texas

ABSTRACT

A performance study of a dual CDC Cyber 170/750 system is presented. The purpose of the study was twofold: to predict the maximum number of interactive users the system could satisfactorily support and to predict the effect of changes in workload and configuration. Two noteworthy aspects of the study are the use of a detailed event trace system to parameterize and validate the model and the use of a very-high-level simulation language. The results of the study, including the accuracy of predictions and the feasibility of the approach, are presented.

*This work was supported in part by Control Data Corporation Grant 80T02.

1.0 INTRODUCTION

The backbone of the University of Texas at Austin academic computing facilities is a pair of CDC Cyber 170/750s running under a locally developed operating system, UT-2D. The dual Cyber system configuration features two loosely coupled CPUs sharing all peripherals. One mainframe presently supports an interactive workload in excess of 180 logged-on users; the other mainframe is devoted to a sizeable batch workload. The configuration is presented in Figure 1.

The system has evolved to its present state by a series of upgrades. The original CDC 6600 and CDC 6400 systems were replaced by the two CDC Cybers in 1979. The present configuration is at maximum capacity in central memory, peripheral processors, and electronic semiconductor memory (ESM, being used as a very fast swapping storage). A significant upgrade was replacing the older 7154-model disk controllers with 7155-model controllers, resulting in a smaller number of controllers and higher effective bandwidth. Another upgrade was the fourfold increase in the storage capacity of the fast-swapping storage. The existing configuration is believed to be balanced with respect to the existing workload. It was necessary to determine the effective saturation capacity of the interactive mainframe and to determine where improvements could be made in configuration or operating system tuning to increase the saturation limit.

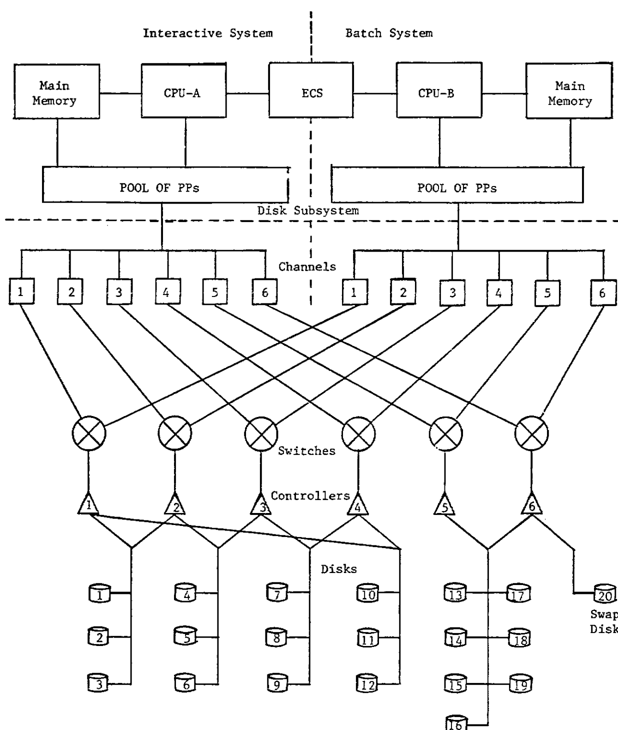


Figure 1—Configuration of the dual Cyber system

The first goal of this study was to predict the performance of the system when the configuration was upgraded. In June 1981 four CDC 7154 disk controllers were replaced by three CDC 7155 disk controllers. We wished to study the effect, on I/O throughput and disk controller use, of reducing the number of disk controllers. The second goal was to determine at what level the currently available computation power could support the steadily increasing interactive workload. This second question effectively reduces to estimating the number of interactive users beyond which the response time for the interactive users becomes unacceptable.

We first constructed a model capturing the essential features of the configuration of the system (before June 1981) accurately. The model is parameterized with the data measured from the real system. Further, we validated the model by comparing the performance obtained from the model with the measured values. We consider the credibility of the model established if the model results differ no more than 10% from the measured results.

The model and its parameters were then altered to reflect the system and workload changes. The performance of the altered model is projected as the future system performance.

In Section 2 we give a brief overview of the system under study and the model obtained for it. The parameters for the model and the validation results are described in Sections 3 and 4. The predictive experiments and an analysis of the results can be found in Section 5. Section 6 presents the conclusions derived from this study and suggestions for optimizing expected future performance.

2.0 SYSTEM DESCRIPTION

The older configuration of the system that was modeled is shown in Figure 1. It consisted of two Cyber 170/750s, a shared Extended Core Storage (ECS) system, and a shared disk subsystem. The two-mainframe system is symmetrical. Each of the Cyber machines has 262K 60-bit words of main memory, 20 peripheral processors (PPs), and 24 channels. Both the CPUs can access ECS where interactive swap files and coordination information for both interactive and batch jobs are being kept. The shared-disk subsystem has 20 disk spindles (12 CDC 844s and 8 CDC 885s) and a total of 6 controllers (4 CDC 7154s and 2 CDC 7155s). An access path can be established between any machine and a disk spindle via a channel and a controller. A schematic of the interconnections between disks, controllers, and channels appears in Figure 2. A channel remains connected to the disk through a controller throughout a disk access; there is no seek-read/write overlap.

The dual Cyber system runs under the control of the UT-2D operating system. In the normal mode of operation one of the CPUs (CPU-A) is used exclusively by the interactive users,

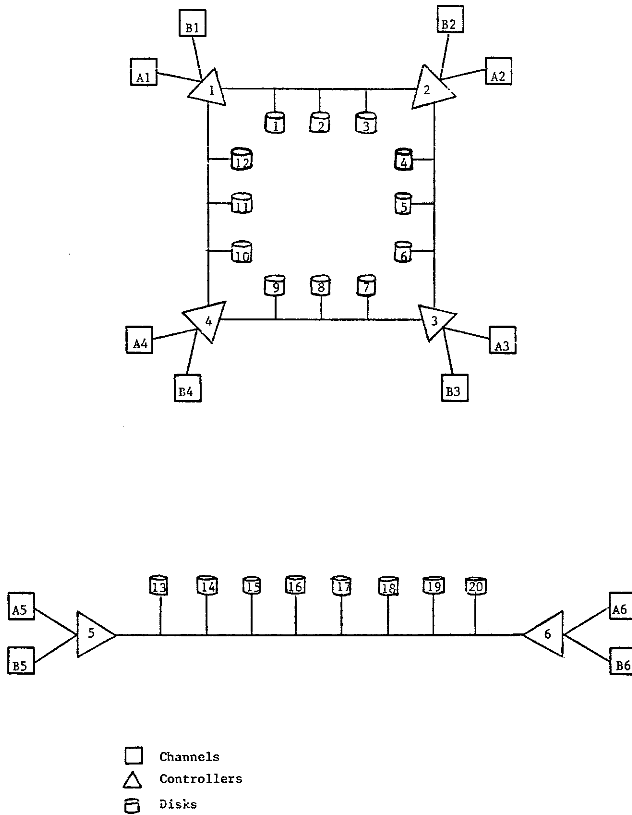


Figure 2—Disk system interconnection

and the other CPU (CPU-B) is dedicated to the batch workload. The scheduling policy for the interactive system favors truly interactive and I/O-bound jobs over CPU-bound jobs. Interactive jobs are swapped to ECS (or to disk when ECS is full) when in the user think state. Some of the swapped jobs migrate from ECS to disk.

When the user hits a carriage return, the job waits in a ready queue until the memory scheduler initiates a swap-in from ECS (or disk) to main memory. The job is then scheduled for a CPU burst. It remains in the main memory until it has completed its CPU and I/O requirements or until it is preempted by a higher priority job, at which time it is swapped out to ECS. While resident in the main memory it alternates between the CPU queue and the I/O queues.

In the disk subsystem the 844 disks contain local files and permanent (archival) files for both batch and interactive jobs. One of the 885 disks is used to store the swap files for both types of jobs; the other 885s contain local files, permanent files, and system files. Each disk has two controllers (a primary and a secondary) associated with it. Each of the controllers is in turn connected to two channels (one from each Cyber). The disk I/O scheduler works as follows: When a disk request arrives, it first checks to see whether the primary channel associated with the disk is available. If it finds the primary channel busy, then it checks the associated secondary channel. When it finds both the primary and secondary channels busy, it waits in a queue for the first channel (primary or secondary) that becomes free.

2.1 MODEL REPRESENTATION

The conceptual model we use is an extended version of the one developed by Atwood and Yu,[1,5] whose primary goal was to analyze the disk I/O workload.

A queueing network model of the system is given in Figure 3. It consists of two independent closed queueing networks (representing the interactive and batch systems) that interfere with each other at the disk controller level. The service time of the Rollin/Rollout server in the queueing network representation includes the time spent waiting for the scheduler to initiate a swap as well as the actual transfer time. This server is modeled as a delay node; i.e., there is no queueing associated with this node. In our first modeling attempt we have not considered the memory contention and the peripheral processor contention (PPs perform the actual I/O transfers). We model the batch system by a closed queueing network, with a fixed degree of multiprogramming. In practice the degree of multiprogramming is not fixed, since jobs arrive at and depart from the batch system.

One of the objectives of this study was to predict the effect of changing the disk subsystem configuration. The nature of the disk system interconnection and the scheduler precludes the use of any exact analytical solution technique for solving the model. Hence we rely upon a simulation approach. The model is written in a queueing network simulation language, PAWS,[4] which has powerful features to accurately model the intricacies of the disk subsystem. The PAWS model also lends itself to parametric analysis of the system.

3.0 MODEL PARAMETERS

An event trace monitor incorporated in the operating system records the occurrences and duration of each event in the system onto a tape.[2] These event sequences are later reduced to useful statistics and histograms by the data reduction package EVENTD.[3] The parameters for the model are derived from the data recorded on two trace tapes (one for each machine) during the same period of time. These data are assumed to be representative and hence are used for parameterizing the model. The major parameters required by the

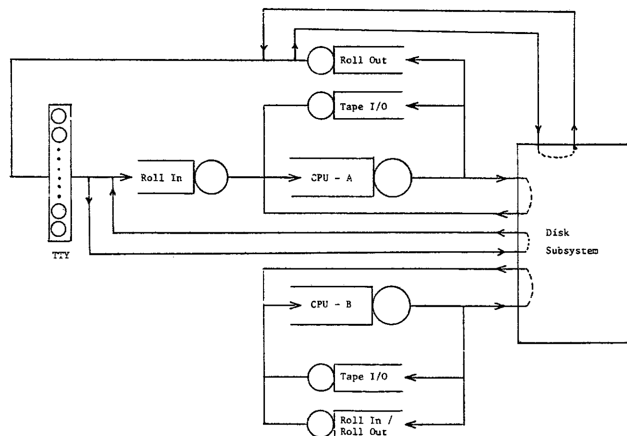


Figure 3—Queueing network model of the system

```
no of interactive users                          120

batch system degree of multiprogramming            6

service times (ms):
  Interactive system
   CPU                                          21.0
   Tape (9-track)                               32.0
   Rollin                                      232.0
   Rollout                                      43.0
   TTY think time                            17000.0
   Disk I/O time    :844                        28.0
                    :885                        29.0
   Swap disk time                               86.0
  Batch system
   CPU                                          27.0
   Tape  (9-track)                              24.0
   Rollov                                        8.9
   Disk I/O time     :844                       35.0
                     :885                       24.0
   Swap disk time                              116.0

Branching probabilities     interactive          batch
     CPU to ROLL-OUT          0.144              0.0158
     CPU to TAPE I/O          0.087              0.27
     CPU TO DISK I/O          0.768              0.714
     ROLL-OUT to DISK         0.27               1.0
```

Figure 4—Parameters for the model
(obtained from trace tapes recorded on Feb. 18, 1981, at 14:10)

model are the mean user think time, mean CPU burst time, mean disk, rollin and rollout service times, probability of accessing different disks, and similar parameters. The histograms produced by EVENTD help in determining the distribution and discipline for various servers in the model. In the first iteration all servers are assumed to have a first-come-first-served discipline with an exponential service time distribution. The values of different parameters are listed in Figure 4.

4.0 MODEL VALIDATION

The metrics used for validation purpose are the mean CPU utilization, mean channel utilization, channel hold time, controller hold time, mean disk throughput, and interactive user mean response time. The values recorded by EVENTD and those obtained from the model are reported in Table I. It can be observed from Table I that most of the model results are within satisfactory (10% tolerance) agreement with the measured values. Only the interactive user response time prediction differs significantly from the measured value. This is primarily due to the difference in definition of the response time, as represented in the model and as reported by EVENTD.

From the measured data we see that the CPUs (with greater than 95% use) are the principal bottleneck of both the interactive and the batch systems. The model reported that the mean queue length at the CPU (at 93%) plus the number of jobs in the I/O system is of the order of 8.5. This gives us the average number of jobs occupying the central memory at any time. The Cyber system allows at most 13 control points, and so at most 13 active jobs may be simultaneously resident in the main memory. Since the CPU becomes a bottleneck before the effect of the limited number of control points can be felt, we conclude that not including the memory contention in the model does not affect its credibility. In addition, our assumptions for service time distributions (exponential) and disciplines (FCFS) appear to be reasonable.

TABLE I—Validation results
(data from trace tape recorded on Feb. 18)

	interactive		batch	
	meas.	sim.	meas.	sim.
CPU util.	.973	.936	.971	.988
Total channel util. :844	1.097	.980	.932	.900
:885	.643	.638	.362	.380
Average channel util.	.282	.262	.218	.214
Controller hold time:844	30.940	30.130	38.260	37.140
:885	49.440	47.800	42.580	40.570
Controller wait time:844	8.328	7.300	7.605	7.400
:885	9.656	8.930	12.400	13.010
Channel hold time :844	39.260	37.290	45.860	44.400
:885	59.100	57.510	54.980	57.290
Disk I/O rate	35.380	34.190	25.800	25.490
Swap disk rate	3.540	3.480	1.202	1.356
Tape I/O rate	4.030	3.980	9.567	9.740
Rollout rate	6.644	6.300	.558	.542

Interactive user response time:

measured 1.453 sec.

simulation 1.800 sec.

5.0 PREDICTION EXPERIMENTS

5.1 Change In Configuration

First an attempt is made to predict the performance of the system when the four controller (CDC 7154) configuration (Figure 3) is replaced by a three controller (CDC 7155) configuration as shown in Figure 5. The CDC 7155 controllers allow the effective transfer rate to be increased by a factor of 2; the disk seek time and positioning time are not altered.

At the same time as the controllers were replaced, the one-half million words ECS (swap storage unit) was upgraded to two million words of Electronic Semiconductor Memory (ESM). With this fourfold increase in the amount of available swap storage, we expected that a job would never be rolled out to the disk. This has been verified by measurement. This is reflected in the model by making the probability of the ROLLOUT TO DISK transition zero.

Table II shows the expected performance of the system with the modified disk configuration. Table III compares the mean holding time and throughput of the controllers of the system under the old configuration with those obtained for the system with the new configuration. As one would expect, the total throughput of the controllers associated with the 844 disks has not changed. The absence of disk roll-in/roll-out is reflected in the reduced throughput of the controllers associated with the 885 disks. The controller hold time has decreased in the new configuration. This is because the transfer rate of the 844 disks has doubled and there are no disk roll-in/roll-out requests from the interactive system.

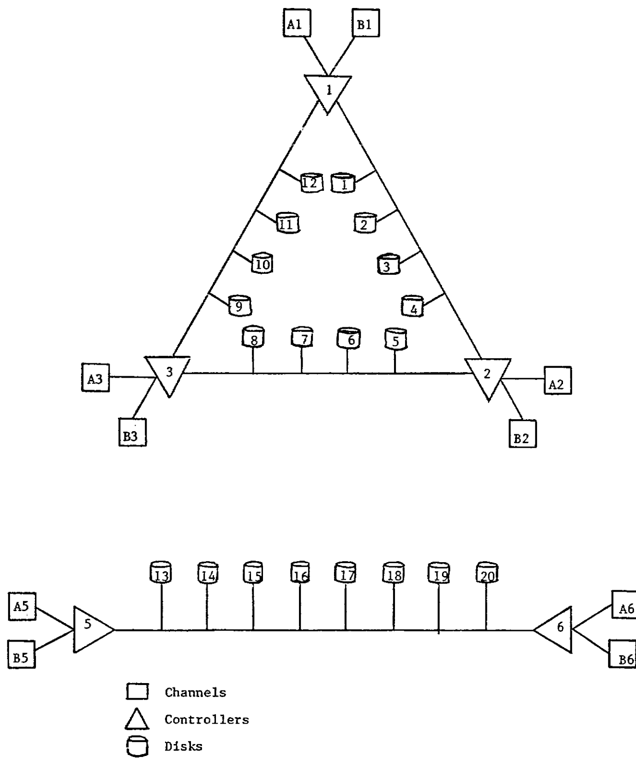


Figure 5—New disk system configuration

5.2 *Change In Workload*

In the second experiment we increase the mean number of interactive users in the system until complete saturation of the system can be observed. The model is used to predict the response time of the interactive users when the number of interactive users is increased from 120 to 300. In Figure 6 we have plotted the expected mean response times against the number of interactive users. The model for this experiment included the new disk system configuration and the increased swapping storage capacity. Other parameter values were unaltered; they were the same as the values used for the validation run. The response time curve with a mean CPU burst time of 21.1 msec is the performance projection under these circumstances. This curve shows an almost linear dependence of the mean response time on the number of interactive users *(N)* for any value of *N* greater than 120. This is to be expected, since we are already dealing with a saturated (CPU) system.

A closer look at the detailed breakdown of the CPU service time requirements of the interactive users revealed that there are a few jobs that are highly CPU-bound and contribute to as much as 24% of CPU use. In the real system these CPU-bound jobs receive CPU service only when there are no short interactive jobs present in the CPU queue. That is, the short interactive jobs have higher priority than the compute-bound interactive jobs.

TABLE II—Expected performance of the system with the modified disk system

		Interactive	batch
Cpu util.		.943	.993
Total channel util.	:844	.824	.715
	:885	.287	.351
Controller hold time	:844	24.750	28.700
	:885	30.380	45.280
Channel hold time	:844	30.860	35.570
	:885	36.330	50.140
Disk I/O rate		34.600	25.600
Swap disk rate		0.000	1.560
Tape I/O rate		3.900	9.900
Rollout rate		6.500	.644

TABLE III—Comparison of disk system performance (old configuration vs new configuration)

		new config.	old config.
Total disk I/O throughput			
interactive		34.600	34.190
batch		25.600	25.490
Swap disk throughput			
interactive		0.000	3.480
batch		1.560	1.356
Total throughput of the controllers	:844	46.800	46.620
	:885	14.900	17.890
Average Controller holding time	:844	26.230	33.330
	:885	37.180	45.550

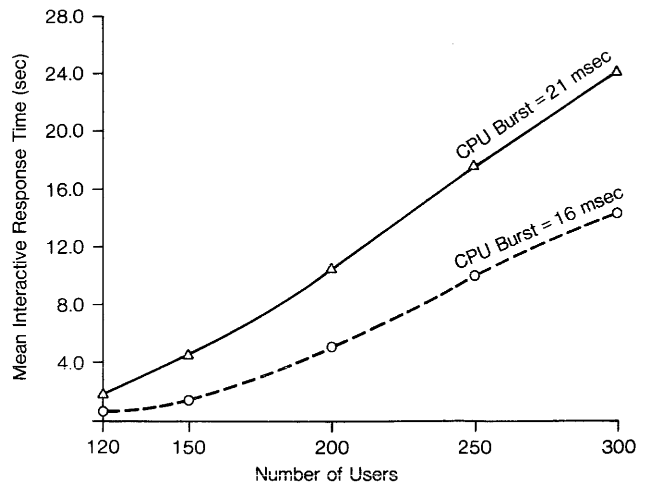


Figure 6—Expected mean response time vs. number of users

The priority discipline at the CPU inherent in the physical system is not implemented in the model. Due to the nature of the priority discipline, we expect the impact of the compute-bound jobs to diminish in the system as the number of interactive users increases, thus reducing the mean CPU burst time. We examined various trace tapes to observe the trend in CPU burst time (Table IV). In 15 trace tape data the mean CPU burst time ranged from 11.8 msec to 23.3 msec. We saw that even the presence of a few highly CPU-bound, interactive jobs significantly increased the mean CPU burst time.

From the data used for parameterizing the validation model we estimated the CPU burst time after eliminating one CPU-bound job (which never performed any I/O during the entire trace period) to be 16 msec. More simulation runs were made with the new mean CPU burst time of 16 msec. We observe that, as expected, the system can support a larger number of interactive users before the response time significantly degrades.

TABLE IV—Mean CPU burst time from various trace tapes

trace tape date	CPU burst (ms)	no. of users
2/10/81,14:26:11	23.3	117
2/18/81,14:10:06	21.1	121
7/1/80, 15:01:43	16.8	125
2/18/81,15:11:48	21.9	125
6/24/80,14:44:14	16.6	134
7/29/80,10:10:43	11.8	136
5/12/81,10:19:58	18.6	144
5/9/81, 16:17:40	18.6	148
5/12/81,13:01:55	20.2	152
10/29/80,15:50:3	15.4	153
7/8/80, 15:22:30	15.7	154
7/15/80,15:16:29	17.8	202
11/2/79,14:06:08	13.1	211
10/8/80,14:34:12	12.7	267

6.0 CONCLUSION

Many capacity planning decisions were made as a result of this study. From the study we concluded that the system bottleneck for both the interactive and batch mainframes was the CPU. When the number of interactive users was increased in the model from 120 to 300, there was no significant increase in the contention for the disk systems. As a consequence, no additional controllers are expected to be procured. Projects to improve the performance of the disk system have been postponed.

Performance improvement efforts now focus on increasing the fraction of the saturated "interactive" CPU available to the users. At the time of this writing, changes in memory scheduling policies were being made to reduce the system overhead load on the CPU. Additional changes in the operating system to shift functions from the CPU to the PPs are anticipated. A study is under way to examine the feasibility of shifting some of the interactive load to the batch mainframe during prime shift, at the expense of batch responsiveness. The feasibility of upgrading the CPUs from 170/750s to 170/760s is also being investigated.

A number of conclusions regarding the measurement and modeling tools were also reached during this study. The event trace facility was repeatedly used to track performance under a steadily increasing workload. The resolution of detail available by the use of this technique is excellent. Detailed data on the behavior of the disk subsystem was necessary because of the interactions of the two mainframes at the individual drive level. Many modifications to the event trace reduction facility were made to provide the statistics necessary to drive the simulation model. The ability to enhance the performance measurement system to meet specific modeling requirements greatly increased the speed and scope of the study.

Our experience in the study showed the value of using a high-level computer-oriented modeling language. Using the PAWS modeling language reduced the coding effort of the model by more than two orders of magnitude from an equivalent FORTRAN model. Numerous test runs against alternate service time distributions and priority schemes were made possible by the ease with which the model could be changed.

ACKNOWLEDGMENTS

We are indebted to W. Jones of the Computation Center staff, who has enhanced the event trace reduction facility to make available the detailed parameters necessary for this study; and to G. Smith, also of the Computation Center staff, who significantly enhanced the event trace recording system.

REFERENCES

1. Atwood, J. W., and K.-C. Yu. "An Empirical Study of a CDC 844–41 Disk Subsystem." To appear in *Performance Evaluation,* 1, 4, or 2, 1 (1981).
2. Howard, J. H., and W. M. Wedel. "The UT-2D Operating System Event Recorder." Technical Report TSN-37, University of Texas at Austin, Austin, Texas, 1974.
3. Howard, J. H., and W. M. Wedel. "EVENTD: UT-2D Event Tape Summary/Dump." Technical Report CCSN-38 (Revised), University of Texas at Austin, Austin, Texas, 1977.
4. "PAWS 1.2 Performance Analysts Workbench System, Modelling Methodology and User Manual." Information Research Associates, Austin, Texas, 1981.
5. Yu, K.-C, "The Effect of System Configuration and File Placement on the Performance of a CDC 844-41 Disk Subsystem." Master's Thesis, Department of Computer Science, University of Texas at Austin, Austin, Texas, 1980.

Computational lexicology: a research program

by ROBERT A. AMSLER
SRI International
Menlo Park, California

ABSTRACT

Computational lexicology may be defined as the application of computers to the study of the lexicon. Taken in its broadest sense, it would be a multidisciplinary field involving the analysis of man-made dictionaries using computers to study their machine-readable text as well as a study of the computational linguistic content and organization of lexicons for use by natural-language processing programs.

Computational lexicology is an emerging field of study for the 1980s being created by converging trends in other disciplines. This paper attempts to outline some of the applications that a knowledge of computational lexicology will facilitate and the possible means of extending such lexical knowledge.

REASONS FOR A COMPUTATIONAL LEXICOLOGY RESEARCH PROGRAM

Computational lexicology may be defined as the application of computers to the study of the lexicon. Taken in its broadest sense, it is a multidisciplinary field involving the analysis of dictionaries written by human beings, using computers to study the machine-readable texts, as well as a study of the computational linguistic content and organization of lexicon for use by natural-language-processing programs. Using the term in this broader sense, I shall refer to the computational lexicology task aspects of computational linguistics, artificial intelligence, cognitive science, and information science that are concerned with the lexicon in computational processes.

I see computational lexicology as an emerging field of study for the 1980s being created by converging trends in other disciplines. Two major reasons for the growth of computational lexicology are that computer typesetting is widely available and machine-readable texts of dictionaries are increasingly available. These primary source materials have also appeared at a time when rapidly declining online disk storage costs and increased acceptance of interactive computation have given rise to natural-language-processing systems needing access to sizable quantities of lexical information in real time.

The rising volume of full-text sources, also a consequence of the increasing availability of computer typesetting, has additionally placed a higher premium on the development of computational systems that can access textual information interactively. Full-text database management, advanced computational linguistic processing systems, and other artificial intelligence applications are striving to encompass new domains and larger text volumes to make full use of this increased availability of text and interactive computation. For these reasons, computational lexicology has emerged as a discipline with both a considerable body of data to begin analyzing and a heightened motivation to provide structured lexical information for existing and future natural-language-processing systems.

Computational lexicology also seems to promise to reveal, in greater depth and detail than ever before possible, the structure and consequences of the organization of the lexicon. Interest in the lexicon from disciplines such as philosophy, linguistics, psychology, and anthropology has never been greater. By introducing computation into the study of the human organization of the lexicon in dictionaries, it has become possible to make new observations and theories about the nature of very ancient and perplexing lexical problems. Computational lexicology sheds new light on the deep semantic relationships involved in the process of defining, the nature of our ability to perform lexical disambiguation from context, and the connectivity of the mental lexicon. Our dictionaries, having evolved over a few centuries of effort, are now as much an artifact of our innate language abilities as they are a product illustrating our civilization's view of the world. Study of the lexicon's organization can consequently reveal basic information about the nature and organization of human knowledge and of our civilization.

Text Streams

Whereas in the past computer technology was applied to solving natural-language information-processing problems for selected segments of textual data, many of which were specifically rendered machine-readable by manual keyboarding, there will soon be more machine-readable textual data than can readily be processed by even the most efficient information-processing algorithms. One may soon be able to seek out and obtain machine-readable text in a specific domain by simply contacting a publisher or a text database vendor instead of contemplating hand entry of the data.

This new volume of text will offer us literally infinite "text streams" of information, and each new publication will be added as a tiny rivulet joining the river of new text data. Just as with a river, there will be opportunities to exploit this massive flow. The equivalent of dams can be set up to gather together large textual databases and control their release, generating information power through the passage of the text through text-processing programs. However, also as in the case of a river, since the flow is continuous, we must develop new methods of selectively sampling and extracting information without trying to fully process every sentence that appears.

There will be opportunities to gather information about the nature of the syntax and semantics of the language as well as opportunities to accumulate knowledge based on the content of the text processed. By establishing selective filters in text streams we can extract examples of linguistic phenomena we wish to study; by watching for certain content phenomena in syntactic patterns we can computationally parse, we can extract information from text streams as they pass.

The text streams will comprise several types of flows. Some, such as electronic mail, will be primarily directed at a few users. They may be casually written, without always having subject headings or titles. The text styles used may approach spoken English more than formal written English, with new conventions, such as capitalization for emphasis and new abbreviations and signals in the text to represent mechanisms possible in human voice, but left out of written communication until now.

Text streams will move at varying rates. Some, such as the news wire services, will flow 24 hours a day and emit millions

of words each month. Others will experience periodic flash flooding and only emit text as publications appear weekly, monthly, quarterly, etc. Among these will be the electronic journals, which are machine-readable copies of paper journals produced by publishers as a result of the computer typesetting process. Finally, some text streams will be completely aperiodic, such as books and similar documents. Each of these will appear almost as a tidal wave and not be subjected to ordinary flow monitoring or processing, but each become a special processing task.

Text streams can be monitored by various methods. Indicators of the flow level and rate can be provided. Sampling to determine the content by subject matter or level of difficulty or for other purposes can be performed. Text streams can be filtered to extract different types of information, either to separate out unwanted text or to separate the constituents of the stream by category for different dispositions. Varying filters can provide some destinations of the text with high-level tables of contents and others with selected extracted sections. A text stream filter can be an arbitrary program that processes text information, monitors for certain types of data, and performs other similar tasks.

Problems Facing Computational Linguistic Systems in the 1980s

Computational linguistics has made vast gains over the previous two decades in parser design and grammar construction. Several successful systems for parsing database queries in ordinary English or directing robot operations and expert problem solving already exist. Nevertheless, computational linguistics has not yet reached the point at which parsing can be performed over unrestricted input text. There are many reasons for this.

Although linguistics has entered the age of computation, the ultimate basis for any sophisticated computational linguistic technique is the computable knowledge accessible in the system's lexicon. The lexicon serves not only as the basis for the recognition vocabulary of any text-processing system, but as the indexed repository of the vast array of additional syntactic, semantic, and pragmatic information upon which text-processing algorithms are based. The scope of this information for each lexical entry has been steadily increasing in both volume and complexity over the past decade. From simple feature-value lists, it has progressed to advanced nonplanar graph-theoretic representations and lexicons combining computable code with stored information.

All current natural-language systems use lexicon composed by hand. The number of person-years necessary to describe the tens of thousands of word senses required for processing unrestricted text cannot be completed by any individual research project also concerned with building sophisticated software to access this lexicon. This has led to a lexical plateau for computational linguistic systems that limits lexicon to a few hundred entries in most cases while researchers concentrate on improving the cleverness of the parsing algorithms and the grammatical coverage of the systems. What cannot be achieved by the application of cleverness is the lexical coverage necessary to process text in the expanded domain of unrestricted English, which is inundating our electronic and magnetic media.

Without concentrated effort on computational lexicology and lexicography, the progress of natural-language-processing systems toward production use in processing unrestricted English text or the extension of existing restricted-domain systems to new domains will be extremely slow. The time may be rapidly approaching when computational linguistics and lexicon building should be undertaken as part of separate research tasks, studied independently.

Progress in Computational Lexicology

The previous two sections provide two motivations for working on computational lexicology: (1) the desirability of making use of new full-text sources and (2) the problem of increasing the size and detail of lexical entries for computational linguistic systems to the extent that they can begin to cope with the increased quantities of text provided by new full-text sources. In addition to these two motivations, the successful completion of analyses of the structure of two machine-readable dictionaries, the *Merriam-Webster New Pocket Dictionary*[1-3] and *The Longman Dictionary of Contemporary English*[15, 16] has promoted interest from computational linguistics itself in this type of study. Thus, there are also new methods available for studying machine-readable dictionaries.

Computational lexicology seeks to understand the meanings of words by accumulating information about their usages. This is the basic task performed by lexicographers when they accumulate citations to use as the basis for writing dictionary entries. Among the potential items of information which can be accumulated from examination of text are

- Verbs with which a given noun is used
- Nominal compounds in which a word occurs
- Hyphenated forms of other combining forms in which words occur
- Adjectives used with nouns
- Morphological variants in which a word form occurs
- Subject domains in which given terms appear
- First occurrences of new word forms
- The identification of defining contexts for new terms in free-running text
- The assimilation of information about word forms from many separate text sources
- Frequency data on the number of textual occurrences, with examples of infrequent phenomena as well as methods for recognizing such phenomena when they occur

All of this lexical information can be gathered by examining machine-readable full-text sources, and it constitutes a new type of information gathering that is of interest to computational lexicology.

RESEARCH IN THE ACQUISITION OF LEXICAL KNOWLEDGE

The following program of research in the new area of computational lexicology outlines several areas in which research

could be performed to build an adequate knowledge of the requirements for natural-language-processing-system lexicons. It is directed primarily at ways to use the computer to extract and organize lexical information, which can then be applied to natural-language-processing tasks. Our knowledge of the lexicon and its parameters is extremely deficient.

Analysis of Machine-Readable Dictionaries As a Source of Lexical Knowledge

Lexical knowledge derives from the analysis of the usages of words. Dictionaries are one form in which such analysis has traditionally been presented. Dictionaries are based on written textual instances of word usage, called citations, which lexicographers organize into lexical entries that present the basic information about how such a word can be used in textual contexts. Typically, the information presented in an ordinary dictionary is limited to morphology (spellings, hyphenation points), pronunciation, etymology, and syntax (parts of speech, usage notes) with semantic information being given in informal fashion as part of the entry's definition. Certain dictionaries in the advanced learner's class[9] include semo-syntactic pattern codes in their lexical entries.

Even in the cases where the semantic information is not codified by the publishers, but appears as part of the text of dictionary definitions, it is possible to apply advanced computational processing to extract much data for potential text system use.[2]

Building Lexical Knowledge from Full-Text Sources

Second in importance as a source of lexical data to direct access to machine-readable dictionary texts is access to the raw data which form the basis for the construction of dictionaries. This task can be approached in three ways.

- First, one can interview human subjects and use a computer to perform bookkeeping, cross-referencing and question-answering to solicit the needed information in an organized and systematic manner.
- Second, one can attempt to extract dictionary definitions from text using fully automatic language analysis and seeking specific definition indicators in texts.
- Third, one can use ordinary text and apply computational linguistic (morphological and syntactic parsing) and information science techniques (cluster analysis, co-occurrence relationships, frequency counts) to gather and present the raw text to human lexicographers or experts for their assimilation and restructuring into formal analyses of terminology.

For the purposes of this paper, the term *full-text source* will be used to refer to any form of data which represents the full text of some document. This might be a publisher's phototypesetting tape containing the text of an article, chapter, book, or any other source. It may contain tables or figures such as would appear in the full text of the published source. Full-text sources will also include materials not intended for publication, such as mail messages; or word processor output, such as business correspondence.

Among the more standard interpretations of *full text* is the notion of machine-readable bodies of text specifically assembled for the study of language. Numerous efforts have been undertaken to assemble such bodies of text in many of the world's major languages.

The task of assembling such a text requires careful planning so that the data selected will be useful to computational lexicology. Simply obtaining a multimegabyte set of words by grabbing all the machine-readable sources available and merging them together hardly provides the basis for scientific observation of the nature of the language as a whole. Thus, ten million words of newspaper stories can be less useful than one million words of carefully sampled text taken from a variety of sources. In this regard, perhaps the most carefully prepared body of English is the one-million-word one prepared by Francis and Kucera at Brown University from full-text sources of 1961.[7, 11] Additionally, a 5-megabyte body of English based upon third- to ninth-grade textbooks was prepared in 1969 by the American Heritage Publishing Company.[4]

APPLIED COMPUTATIONAL LEXICOLOGY

Although this paper is intended to support the case for basic research in computational lexicology, it is also worthwhile examining what would be some applications of such research.

There is no limit to the number of areas in which computers are applied to processing natural-language text. To some degree all these areas could be affected by developments in computational lexicology. Among the first to be affected might be fields that already have a considerable emphasis on the availability of words (e.g., spelling correction) or have established uses for lexicons (e.g., content analysis systems, mechanical translation systems, and word processing systems). One promising new area that might benefit would be full-text database retrieval.[14]

Spelling Correction

Spelling correction is one of the most neglected aspects of natural-language system design. It is a stepchild of the sophisticated computational linguistics systems designed within artificial intelligence and often regarded as a simple instance of "bells and whistles" to be added to a natural-language system after the fact. This situation is changing, because studies are beginning to show that of all the areas in which natural-langauge systems fail, spelling correction (and unknown lexicon) are two of the most frequent causes of failures. In some systems spelling errors and unknown vocabulary may account for nearly 50% of all failures in system response.[21]

Spelling correction as a technique has recently become the topic of intensive, but largely unreported, research in the word processing field. IBM's DisplayWriter system offers a multilingual spelling correction capability, which, though originally thought to be a simple matter of checking lists of correctly spelled words, turned out to be a significant problem

when languages other than English were assumed to be correctable with the same algorithm as English. This is because spelling correction depends greatly on suffix analysis in English, and these techniques do not work, for example, in German, where suffix information can be embedded inside agglutinated compound words. Thus, a knowledge of the morphology of a language is necessary for performing multilingual spelling correction properly.

In addition to morphological considerations, there are numerous strategic variations in spelling correction algorithms that can have a significant impact on their performance. Current spelling correction systems in wide use on the ARPANET generally make an assumption that there is only one error in a misspelled word. This assumption lends itself to a convenient and inexpensive correction algorithm which in effect considers every letter in the word except one to be correct (or considers there to be only one transposition of two letters). The consequence of this simple algorithm is that often dozens of possible corrections are offered for a single-letter misspelling in a four-letter word, yet very little is done to help the user making two or more errors in spelling a 15-letter word. "Internationalization" can readily be recognized by English speakers even with several errors—"innternattionalisation" contains nn for n, tt for t, and s for z, yet it is readily manually corrected. Finally, spelling correction is beginning to appear in the literature of computer science as both the theoretical basis and the necessity for it are becoming better recognized.[18, 19, 22, 23]

Computational lexicology will assist spelling correction developments in three ways. By surveying full-text sources, it will be able to provide information on the frequency of spelling errors and their types. By amassing new lexicon, it will be able to provide spelling correction algorithms with improved data for their use—lexicon that will contain spelling forms from special subject domains as well as increased quantities of error-free lexical and morphological information (from published dictionaries). Finally, by providing statistical information on the frequencies of morphological and phonological forms, it may be possible to design better spelling correction algorithms.

A note to the topic of spelling correction: Word hyphenation is becoming a feature of some word processing systems. Word hyphenation is poorly understood and even more poorly described in algorithmic form.[20] Spelling correction lexicons are logically the place to incorporate hyphenation data because of the savings in storage accompanied by inserting the hyphenation points into the spelling dictionary vs. having a separate lexicon for hyphenation. Thus the field of spelling correction may soon broaden its boundaries to become the field of word correction. It would involve not only repairing misspellings, but providing correct hyphenation information and perhaps additional capabilities such as checking for abbreviation consistency, checking for capitalization, foreign language transliteration, and even font changes (e.g., italicizing foreign phrases). The work on the "Writer's Workbench" UNIX programs at Bell Labs [5,6,13,17] and the WORDS package offered by Houghton Mifflin (Houghton Mifflin, private communication, 1981) appears to be leading in the direction of developing such computational tools for correcting words in text.

Machine Translation

The current state of the art in mechanical translation (MT) is extremely hard to assess. The field is charged with highly controversial claims, international rivalries, and a historical stigma. Despite these problems, one facet of mechanical translation does become clear. It depends on a massive lexicon.

Although practitioners of MT may never resolve their arguments over computational linguistic strategy or even systems design directions (machine-assisted vs. fully-automatic) it is possible to make advances in the necessary prerequisites for mechanical translation without entering into the fray (or at least without becoming locked in unresolvable battle).

One of the conspicuous lacks in mechanical translation is the availability of an adequate lexicon. Naive MT consumers very often conceive of MT as a task involving a simple paired word list (or even worse, a multilingual word table) and an amazing and mysterious computational algorithm which can take this word list and use it to perform translations. In reality, the algorithms necessary for performing mechanical translation are within our reach today, but the grammar and lexicon these algorithms would require to provide flawless high-quality mechanical translation are many years from attainment. Linguists would readily admit that there does not exist a complete grammar of any major natural language. Lexicologists will likewise explain that a complete lexicon, with all the grammatical information needed to perform MT, is nowhere to be found.

Thus, unlike the naive view that the MT problem requires finding or building the marvelous algorithm, the actual task involves creating an adequately detailed description of a language in terms of its syntactical possibilities and lexical semantics. Computational lexicology is thus likely to be necessary in providing an MT system with its basis for operation.

Retrieval without translation

A very significant consequence of the increased availability of machine-readable full-text sources and the accompanying growing shortage of economical language translation capability is that a high premium will be placed on determining whether a document is worth translating before its translation is undertaken. Thus full-text sources in foreign languages such as Russian, German, French, and Spanish will become readily available via satellite telecommunications, which could put a foreign newspaper on our doorsteps in a matter of hours after its publication in its native country. However, the translation of this text would require a major effort—one which would not be undertaken unless potential readers knew there was something of interest to them in the text that could justify the cost of translating.

Therefore a situation is set up in which a full-text filtering system would be desired which operated directly on the foreign langauge full-text source material and rendered a judgment about the content before sending such material to a translator. The only real indicator of the content of a document that can be used without full grammatical parsing is an analysis based on the words used in the text. This in turn implies that a sophisticated lexicon will be the prerequisite to filtering untranslated foreign source material.

REFERENCES

1. Amsler, Robert A., and John S. White. *Final Report for NSF Project MCS77-01315, Development of a Computational Methodology for Deriving Natural Language Semantic Structures via Analysis of Machine-Readable Dictionaries*. Technical Report, University of Texas at Austin, Linguistics Research Center, 1979.
2. Amsler, Robert A. *The Structure of the Merriam-Webster Pocket Dictionary*. PhD thesis, The University of Texas at Austin, December, 1980. Also appeared as CS Technical Report TR-164.
3. Amsler, Robert A. *A Taxonomy for English Nouns and Verbs,* pages 133–138. Association for Computational Linguistics, Stanford University, June 29–July 1, 1981. *Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics,* 1981.
4. Carroll, John B., Peter Davies, and Barry Richman. *The American Heritage Word Frequency Book*. Boston: Houghton Mifflin, 1971.
5. Cherry, Lorinda. "Computer Aids for Writers." *SIGPLAN Notices,* 16 (June 1981), pp. 62–67. Special issue containing *The Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* held over June 8–10, 1981 in Portland, Oregon.
6. Cherry, Lorinda. *A Toolbox for Writers and Editors*. pages 221–227. AFIPS, Houston, Texas, 1981.
7. Francis W. N. *Manual of Information to Accompany A Standard Sample of Present-Day Edited American English, for Use with Digital Computers*. Providence, Rhode Island: Brown University, Department of Linguistics, 1964.
8. G. & C. Merriam Co. *The New Merriam-Webster Pocket Dictionary*. Springfield, Mass.: G. & C. Merriam, 1971.
9. Hornby, A. S., E. V. Gatenby, and H. Wakefield. *The Advanced Learner's Dictionary of Current English*. London: Oxford University Press, 1971.
10. Houghton Mifflin Company. WORDS. Advertising literature. September 1981.
11. Kucera, Henry, and W. Nelson Francis. *Computational Analysis of Present-Day American English*. Providence, Rhode Island: Brown University Press, 1967.
12. Procter, Paul, ed. *Longman Dictionary of Contemporary English*. London: Longman Group, 1978.
13. Macdonald, N. H., L. T. Frase, and S. A. Keenan. *Writer's Workbench: Computer Programs for Text Editing and Assessment*. Piscataway, New Jersey: Bell Laboratories 1980. Part of the documentation for the UNIX operating system.
14. Mead Data Central. *LEXIS, A Primer and LEXIS Quick Reference Manual*. New York: Mead Data Central, 1980.
15. Michiels, A., J. Mullenders, and J. Noel. *Exploiting a Large Data Base by Longman*. COLING 80, Tokyo, Sept. 30–Oct. 4, 1980. *Proceedings of the 8th International Conference on Computational Linguistics,* 1980, pp. 374–382.
16. Michiels, A. "Exploiting a Large Dictionary Data Base." PhD thesis, University of Liege, Belgium, 1981.
17. Morris, R., and Lorinda Cherry. "Computer Detection of Typographic Errors." *IEEE Transactions on Professional Communication,* PC–18 (1975) 54–64.
18. Peterson, James L. *Lecture Notes in Computer Science*. Volume 96: *Computer Programs for Spelling Correction*. New York: Springer-Verlag 1980.
19. Peterson, James L. Computer Programs for Detecting and Correcting Spelling Errors. *CACM,* December 1980, 23(12), 676–687.
20. Peterson, James R. *Use of Webster's Seventh Collegiate Dictionary to Construct a Master Hyphenation List. AFIPS, Proceedings of the National Computer Conference* (Vol. 51), 1982.
21. Sacerdoti and Bozena Thompson. Presentations at Naval Research Laboratory Workshop on Applied Computational Linguistics in Perspective, Stanford University, June 1981.
22. Simmons, Robert F., and Robert A. Amsler. *Modeling Dictionary Data*. Computer Sciences Report 7, New York University, Courant Institute of Mathematical Sciences, Computer Science Department, 1975.
23. Turba, Thomas N. "Checking for Spelling and Typographical Errors in Computer-Based Text." *SIGPLAN Notices,* 16 (1981), pp.1 51–60. Special issue containing *The Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation,* symposium June 8–10, 1981, Portland, Oregon.
24. Zamora, A. *Automatic Detection and Correction of Spelling Errors in Large Data Bases. ASIS Journal* 31 (1978), pp. 51–57.

Use of *Webster's Seventh New Collegiate Dictionary* to construct a master hyphenation list

by JAMES L. PETERSON
University of Texas
Austin, Texas

ABSTRACT

A machine-readable form of *Webster's Seventh New Collegiate Dictionary* has been obtained. After substantial processing to understand the form and content of the dictionary and to correct residual typographical errors, we have begun the task of constructing a master word-hyphenation list based on this dictionary and other sources. Substantial problems can arise in preparing the master hyphenation list because of incompatible hyphenation definitions from various sources. Some statistics are given.

MOTIVATION

Over the past decade an increasing number of documents have been prepared with the use of a computer. These computer-based systems provide certain standard functions: input, storage, editing, formatting, and output. Recently, however, a new function has been introduced: *analysis*. The idea is to have the computer analyze the text of the document to catch errors and improve the quality of the document.

Many different types of analysis are possible. Perhaps the best known is spelling checking and correcting.[1] A large number of programs currently available will check each word in a document for correct spelling. This is generally done by providing a list of words that are correctly spelled. Any word in the document that is not in the list of correctly spelled words is a candidate misspelling and is flagged for the author's attention.

Although the best-known form of document analysis is detecting spelling errors, it is far from the only form. The Writer's Workbench of PWB/UNIX[2] provides several forms of document analysis, including readability indexes, wordiness, punctuation, and style. More advanced forms of analysis would include checking grammar and syntax of documents.[3,4]

For all of these tasks two things are needed: an analysis algorithm and a suitable word list annotated with the needed properties of the words.

In our case, we are trying to evaluate several different readability formulas.[5] Many of these formulas involve counting the number of syllables in a word or sentence. Thus, we would like to be able to divide a word into its constituent syllables.

Trying to find existing algorithms for splitting a word into syllables, we noticed that this is the *hyphenation problem:* given a word, where can that word be hyphenated? A word can be hyphenated only on a syllable boundary. Thus, if we can determine hyphenation points correctly, we can use that information to define syllable boundaries and hence the number of syllables.

There are several published hyphenation algorithms,[6,7] but they tend to be either very simple (and obviously not very good) or rather complex but of unknown validity. A study of hyphenation rules leads immediately to the conclusion that the only totally correct algorithm would be to look up the word in a word list annotated with hyphenation information.[8]

At the same time, we are attempting to investigate the difficulty of checking grammar and syntax in English documents. We need to know, for each word, the possible parts of speech for that word. This, plus a partial English grammar, should allow us to create a syntax-checking program for English. A sentence with no parse in the grammar would be flagged for the author as a possible syntax error.

In searching for a word list with part-of-speech and hyphenation information, we discovered the machine-readable *Webster's Seventh New Collegiate Dictionary* (W7).[9]

WEBSTER'S SEVENTH NEW COLLEGIATE DICTIONARY

W7 exists in a computer-readable form. This is not just a word list, but a copy of the entire dictionary, including definitions, cross-references, variants, synonyms, and so on. It consists of some 12,242,868 characters, with 68,766 main entries.

The original dictionary was keyboarded onto the Q-32 computer at System Development Corporation (SDC) for a project headed by John Olney.[10] The dictionary was then heavily edited and moved onto an IBM 360. Tapes of this form, which were widely distributed, included a copy sent to the IBM T.J. Watson Research Center and further processed by C. Alberga. A copy of this was acquired by Robert Amsler.[11] We have acquired a copy of the dictionary from Amsler and have modified it in many minor ways.

Figure 1 shows a sample of the dictionary file. Each line of the file has a character in Column 1 identifying the type and format of the line. Table I shows the number and meaning of each line type.

Each line is composed of a number of fields. Fields are separated by a semicolon and are defined by their position. The first field of each line is the line type character (F, V, D, L, R, X, or S, as given above). The remaining fields depend on the type of the line. For example, the second entry on an F-line is a main-entry word, the fifth field has hyphenation information, and the seventh has part-of-speech information.

Character Codes

A major problem with the dictionary is its character set. First, the dictionary publisher did not feel constrained in use of characters, but chose whatever symbols best fit the purpose. Second, the dictionary was originally encoded in an extended BCD (for the Q-32 computer), then translated into EBCDIC (for the IBM 360/370) and now has been translated into ASCII (for our PDP-11/60). None of these character sets is completely compatible with the others, and none of them are sufficient to represent the variation found in the original printed dictionary. Hence an encoding scheme must be used to expand the set of representable characters. This expansion occurs in two independent directions: *font information* and *special characters*.

We have represented font information by use of the square brackets in ASCII to surround any special font material. Five font types are recognized: (1) italic, (2) mini-caps, (3) bold, (4) subscripts, and (5) superscripts. Each is denoted by an

```
F;Charybdis;;;33;n;;
D;0;;;n;a whirlpool off the Sicilian coast#
personified by the ancients as a female monster
F;chase;1;;;vb;;
D;1;a;;vt;to follow rapidly : [mini PURSUE]
D;1;b;;vt;[mini HUNT]
D;1;c;;vt;to follow regularly or persistently#
with the intention of attracting or alluring
L;2;;;[italic obs]
D;2;;;vt;[mini HARASS]
D;3;;;vt;to seek out
D;4;a;;vt;to cause to depart or flee : [mini DRIVE]
L;4;b;;[italic slang]
D;4;b;;vt;to take (oneself) off
D;1;;;vi;to chase an animal, person, or thing
D;2;;;vi;[mini RUSH], [mini HASTEN]
S;[bold syn] [mini PURSUE], [mini FOLLOW], [mini TRAIL]:#
[mini CHASE] implies going swiftly after and trying to#
overtake something fleeing or running; [mini PURSUE] suggests#
a continuing effort to overtake, reach, attain; [mini FOLLOW]#
puts less emphasis upon speed or intent to overtake and may not#
imply an awareness on the part of the leader that he is#
pursued; [mini TRAIL] may stress a following of tracks or#
traces rather than a visible object
F;chase;2;;;n;;
D;1;a;;n;the act of chasing : [mini PURSUIT]
D;1;b;;n;[mini HUNTING] -- used with [italic the]
D;1;c;;n;an earnest or frenzied seeking after something desired
D;2;;;n;something pursued
D;3;a;;n;a franchise to hunt within certain limits of land
D;3;b;;n;a tract of unenclosed land used as a game preserve
F;chase;3;;;vt;;
D;1;a;;vt;to ornament (metal) by indenting#
with a hammer and tools without a cutting edge
D;1;b;;vt;to make by such indentation
D;1;c;;vt;to set with gems
D;2;a;;vt;[mini GROOVE], [mini INDENT]
D;2;b;;vt;to cut (a thread) with a chaser
F;chase;4;;;n;;
D;1;;;n;[mini GROOVE], [mini FURROW]
D;2;;;n;the bore of a cannon
D;3;a;;n;[mini TRENCH]
D;3;b;;n;a channel (as in a wall) for something to#
lie in or pass through
F;chase;5;;;n;;
D;0;;;n;a rectangular steel or iron frame into which#
letterpress matter is locked for printing or plating
X;form;;;4;
```

Figure 1—Sample of the dictionary file

identifying keyword immediately after the opening (left) square bracket, followed by a space, followed by the material to be in the defined font, followed by the closing (right) square bracket. For example, an italic *was* is represented as [italic was], a mini-caps AMBIENT is [mini AMBIENT] and a bold *syn* is [bold syn]. Superscripts and subscripts may be italic, mini-caps, or bold; and a few superscripted superscripts also occur, as in 6.24 {times} 10 [sup 10 [sup 10]].

The dictionary includes a large number of special symbols that are not representable in ASCII. These include all the Greek alphabet, the Hebrew alphabet, and many miscellaneous other special symbols. All special symbols which are not available in ASCII (and some that are) have been given representations by enclosing the name in braces, as: {degrees} (for a degree symbol), {times} (for multiplication represented by a small x), and {tau} (for the lower-case Greek letter tau).

Each symbol name has been selected to exclude embedded blanks. Thus all characters between an opening right brace and its closing right brace are nonblank. Certain characters in ASCII (braces, brackets, question mark, exclamation mark, and so on) have also been represented as extended characters to allow the ASCII character to be used for other purposes (such as font and special character representation). They occur only infrequently (fewer than 100 times).

ERRORS IN W7

While processing W7 both to understand its contents and to put those contents into a usable form, we encountered a large number of errors. These errors were of several types:

1. Merged illustrations. For example, under *false* the illustration was ⟨ ~ documents ~ teeth⟩ and should have been ⟨ ~ documents⟩ ⟨ ~ teeth⟩. To correct this we searched for any line of the form ⟨. . . ~ . . . ~ . . .⟩.
2. Words containing letters with accents (236 entries). The accent field was wrong about half the time. The normal problem was that the accent was on the wrong letter. In these cases, the hyphenation information generally showed syllables that were two letters too long.
3. Incorrect values in fields. We created a list sorted by frequency of the contents of each field (as listed in the appendix of Peterson[12]). These could then be examined for rare or inappropriate values; for example, a *g* in a numeric field, or a zero in an alphabetic field.
4. Mismatched parentheses or brackets. We wrote a program to simply count parentheses, braces, and brackets. Many were found to be mismatched.

All these errors, once found and verified, were corrected by hand, using a text editor.

A last form of error analysis was an attempt to find typographical errors. The approach was simple: we extracted a list of all unique words used in the dictionary definitions. This produced a list of 54,298 words. We compared this list with the list of all words defined in the dictionary (main entries, variants, or related words). This reduced our list to 20,292 words that were *used* in definitions but not *defined*. Many of these were derived forms of defined words: past tense, plurals, and so on. Doing some simple suffix analysis, we were left with about 8,000 words. Most of these were apparently Greek or Latin botanical or zoological names. Deleting those ending in '-ia' or '-ae' and all words in italics in the dictionary left a list of 2,821 words.

These were checked by hand to produce a list of 903 incorrectly spelled words. We also found 54 words which were used, but not defined, such as *Australasian, nubby, spondunmene, seneschal.* We also found a smaller list of words with with typographical errors in the main entry in the computer files.

Of the 903 typographical errors, 543 were the result of a missing blank between two words. Of the remaining 360, 34% were a missing letter, 27% were a wrong letter, 20% were an

TABLE I—Number and meaning of line types in W7 dictionary file

Line type	Number	Meaning
F	68,766	First line, start for a new word
V	9,957	Variant
D	140,500	Definition, one per line
L	11,990	Label
R	19,123	Related word
X	4,598	Cross-reference
S	834	Synonym block

extra letter, and 13% were the result of transposed letters. The remaining errors were caused by two extra or two missing letters, or by transposing two letters around a third. The middle letter in this case was always a vowel. (For example, *min* would be typed *nim*.)

We also found 10 cases of typographical errors in the original printed dictionary. It was interesting to follow these errors through the various printings and editions of the Merriam-Webster dictionaries. Four errors were corrected in the 1970 printing of W7, one in the 1971 printing of W7, and one in the *New Collegiate Dictionary*; [13] and four errors remain in the most recent *Collegiate*:[14]

1. In *bitch, doublecross* should be *double-cross.*
2. In *vanity, knicknack* should be *knickknack.*
3. In *drift, quantitive* should be *quantitative.*
4. In *barranca, gulley* should be *gully.*

The first two errors are also in *Webster's Third New International Dictionary.* [15]

CONSTRUCTION OF THE MASTER HYPHENATION LIST

As mentioned before, each line in the dictionary is a sequence of fields, separated by semicolons. The fourth field on F-cards, the second field on R-cards, and the second field on V-cards contain hyphenation information for their respective entries. The hyphenation information is a sequence of one-digit numbers giving the number of characters between possible hyphenation points. As an example, a word such as *devilish* is hyphenated as *dev-il-ish* and would be encoded as 323. The distance from the start of the word to the first hyphenation point is 3; the next syllable is of length 2; the last syllable is of length 3. The word *ethnological,* hyphenated as *eth-no-log-i-cal,* is encoded as 32313.

The encoding of the distance between hyphenation points is one digit, from 1 to 9. If the distance exceeds 9, we continue with the upper-case letters (as with a hexadecimal representation). Thus A is 10, B is 11, C is 12, . . . , Z is 35. The most common lengths of syllables are 232, 322, and 223.

The longest hyphenation encoding is

pneumonoultramicroscopicsilicovolcanoconiosis;
4222323423123222213

pneu-mo-no-ul-tra-mi-cro-scop-ic-sil-i-co-vol-ca-no-co-ni-o-sis

There are five words with nine syllables each.

While we were processing W7, we became aware of several word lists with hyphenation data. We were able to acquire copies of three of these:

1. LONG, based on *Longman's Dictionary of Contemporary English*[16]
2. RADC, from Rome Air Development Center
3. IBM, from the Advanced Office Systems Laboratory of IBM

Each of these word lists recorded hyphenation information in a different way, generally by an embedded hyphenation indicator (either a hyphen, an equal sign, or a period). We converted all of these to a uniform encoding based on the W7 encoding.

Each source was separately checked for accuracy. We started with the four files: W7, LONG, RADC, IBM. After putting these in a common format, each file was checked for acceptable forms of words: No internal blanks, no apostrophes, no numbers, no hyphens, and no foreign characters. Each file was checked to assure that every syllable had a vowel (except *dirndl, Houyhnhnm, Niflheim, McCoy, McCarthy,* and their derivatives). Each file was checked to identify cases where the same word might have multiple hyphenations.

We then combined these lists in an attempt to create a master hyphenation list. Major advantages of this approach are the large set of words in the resulting list and the redundant verification of the hyphenation information.

One of the problems with hyphenation is that some spellings have more than one hyphenation, depending on the meaning of the word. The differing hyphenations are a result of varying pronunciations, which are related to the source of the word or its part of speech. For example, *project* used as a noun is *proj-ect*, but used as a verb it is *pro-ject.* Differing meanings are the reason for the differing hyphenation of '*add-er*' (one who adds) and '*ad-der*' (a type of snake); in both cases the word is a noun. A list of 187 of these words has been constructed.

It is sometimes difficult to determine when a word has, in fact, multiple hyphenations. The word division supplement of the *Government Printing Office Style Manual*[17] lists 100 such words; the others were found while processing the various lists. However, one also finds anomalous situations such as, in W7, the varying hyphenation of *footedness* in *slow-footedness (footed-ness)* and *flat-footedness (foot-ed-ness)* or the variation in *spoken* and *fair-spoken (spo-ken* versus *spok-en).*

In general, multiple hyphenations come in two forms: compatible and incompatible. If the hyphenation points of one hyphenation are a subset of those of another, the hyphenations are compatible, and we choose the larger set of hyphenation points. Thus, the hyphenations *footed-ness* and *foot-ed-ness* are compatible, and we choose *foot-ed-ness.*

When neither hyphenation is a subset of the other, the two are incompatible (such as *spo-ken* and *spok-en*), and other means must be used to resolve the differences. The simplest means would be reference to a universally accepted definitive source. However, no single source appears universally acceptable. We have a list of 32 words for which W7 indicates one hyphenation and all our other sources indicate a different one.

For words not on the list of known multiple hyphenations, multiple hyphenations may mean an error in at least one of the possible hyphenations. We initially had the following number of words in each file:

W7	72,983
LONG	20,506
RADC	28,689
IBM	70,355

If we combine all these files, we end with a list of 108,190

unique words, of which there are 2,011 whose hyphenation is in dispute.

To resolve these disputes, we counted the number of times each hyphenation occurs in the combined files. Where two or three sources showed the hyphenation one way and only one source showed an incompatible hyphenation, the more common hyphenation was chosen. In cases in which two sources indicated one hyphenation and two indicated another, or in which only two or three sources included the word and each indicated a different hyphenation, we had to consider each word separately.

At the moment we are attempting to get either of two additional sources: the word division list of the U.S. Government Printing Office[17] or the word-hyphenation list of the *American Heritage Dictionary.*[18] With either one of these we will have an odd number of hyphenation sources and hence may be able to use a strict majority rule to arrive at a defined hyphenation.

An interesting side point is to examine the source of hyphenations deemed incorrect. The following list shows the number of hyphenations of words deleted by our selection procedure:

LONG	817
IBM	316
W7	171
RADC	120

The high number of variant hyphenations from LONG undoubtably reflects the different pronunciation (and hyphenation) resulting from British English.

CONCLUSIONS

We are still tying up minor loose ends in the W7 dictionary and our master hyphenation list. We have recently acquired the word frequency information of the American Heritage word frequency study[19] and are proceeding to add that frequency information to our hyphenation word list information. This will allow us to evaluate hyphenation algorithms and readability formulas in ways not previously possible. We will shortly be able to determine quantitatively the accuracy of various hyphenation algorithms with respect to both a given word list and the word list weighted by frequency. This will allow us to select appropriate algorithms for various situations.

REFERENCES

1. Peterson, J. L. "Computer Programs for Detecting and Correcting Spelling Errors." *Communications of the ACM,* 23 (1980), pp. 676–687.
2. Cherry, L. "A Toolbox for Writers and Editors." *1981 Office Automation Conference Digest,* Houston, March 1981. Arlington, Virginia: AFIPS Press, 1981. pp. 221–227.
3. Miller, L., G. Heidorn, and K. Jensen. "Text-Critiquing with the EPISTLE System: An Author's Aid to Better Syntax." *AFIPS, Proceedings of the National Computer Conference* (Vol. 50), 1981, pp. 649–655.
4. Sager, N. *Natural Language Information Processing: A Computer Grammar of English and its Application.* Reading, Massachusetts: Addison-Wesley, 1981.
5. McCallum, D., and J. Peterson. "Computer-Based Readability Indexes." In Preparation.
6. Rich, R. P., and A. G. Stone. "Method for Hyphenating at the End of a Printed Line." *Communications of the ACM,* 8 (1965), pp. 444–445.
7. Knuth, D. E. *TEX and Metafont: New Directions in Typesetting,* Bedford, Massachusetts: Digital Press, 1979.
8. Reid, B. K. "Scribe: A Document Specification Language and Its Compiler." Technical Report CMU-CS-81-100, Department of Computer Science, Carnegie-Mellon University, October 1980.
9. *Webster's Seventh New Collegiate Dictionary.* Springfield, Massachusetts: G. & C. Merriam, 1965.
10. Reichert, R., J. Olney, and J. Paris. "Two Dictionary Transcripts and Programs for Processing Them. Volume I: The Encoding Scheme, PARSENT and CONIX." Technical Memorandum TM-3978/001/00, System Development Corporation, Santa Monica, California, June 1969.
11. Amsler, R. A. "The Structure of the Merriam-Webster Pocket Dictionary." Technical Report TR-164, Department of Computer Sciences, The University of Texas at Austin, December 1980.
12. Peterson, J. L. "Webster's Seventh New Collegiate Dictionary: A Computer-Readable File Format." Technical Report, Department of Computer Sciences, The University of Texas at Austin, 1981. In Preparation.
13. *Webster's New Collegiate Dictionary.* Springfield, Massachusetts: G. & C. Merriam, 1975.
14. *Webster's New Collegiate Dictionary.* Springfield, Massachusetts: G. & C. Merriam, 1981.
15. *Webster's Third New International Dictionary of the English Language Unabridged.* Springfield, Massachusetts: G. & C. Merriam, 1976.
16. *Longman Dictionary of Contemporary English.* London: Longman Group, 1978.
17. *Word Division.* Supplement to *Government Printing Office Style Manual.* 7th ed. Washington, D.C.: U.S. Government Printing Office, 1976.
18. *The Word Book.* Boston: Houghton Mifflin, 1976.
19. Carroll, J. B. *The American Heritage Word Frequency Book.* Boston: Houghton Mifflin, 1971.

Models, languages, and heuristics for distributed computing

by ROBERT E. FILMAN
Hewlett Packard
Palo Alto, California
and
Indiana University
Bloomington, Indiana
and
DANIEL P. FRIEDMAN
Indiana University
Bloomington, Indiana

ABSTRACT

We are interested in the issues surrounding computer problem solving in systems of loosely coupled processes. This paper is a compendium of ideas related to the software issues involved in programming distributed systems. We discuss two aspects of this problem: languages and models for distribution, and heuristics for organizing distributed systems. The second section of the paper discusses the nature of distributed languages and models, and presents a comparison of the attributes of several of the major proposals for distributed computing. The third section is a discussion of some heuristics for the organization of multiple-process systems.

INTRODUCTION

The marvels of miniaturized silicon are leading to a world of cheap microprocessors. These microcomputers bring with them the hope of faster and cheaper versions of the conventional, mainframe computer—an army of small automata, eager to increment and loop, ready to go out and solve our computing problems. However, as any manager of a large software project can assure you, a large collection of dumb computing agents does not add up to a working system. Microprocessors need to be told not only what to do, but how to do it. They need to cooperate and communicate in their processing task; but their cooperation must not turn into a bureaucracy, expending more energy on communication than on production.

The next generation of computer architectures will provide users with not just one but many computers to perform their tasks. But improved computational productivity is not achieved by processing power alone. Along with multiple-processor architectures must come the software facility to profitably use that computing power. We call such an integrated, multiple-processor system a *coordinated computing* system. The integration we require in coordinated computing is not merely interprocessor communication, but interprocess cooperation.

This paper discusses the issues surrounding computer problem solving in systems of loosely coupled processes. Toward this end, we have been studying models of communication, programming languages for distributed computing, and heuristics for system organization. In this paper we present a compendium of ideas related to the software issues involved in achieving coordinated computing. We discuss two aspects of the coordinated computing task: languages and models for distribution and heuristics for organizing distributed systems. The second section of the paper discusses the nature of distributed languages and models and compares the attributes of several of the major proposals for distributed computing. The third section discusses some heuristics for the organization of multiple-process systems. These issues are discussed in greater detail in *Coordinated Computing: Tools and Techniques for Distributed Software.* [1]

MODELS AND LANGUAGES

Requirements for Distribution

Every programming language or model makes assumptions about its computing environment. A programming system for coordinated computing is no exception. Particularly important are the assumptions that such a system makes about the interface it provides the programmer:

1. First, a distributed programming system has multiple, independent, concurrently computing processes or supports some activity that corresponds to doing many independent activities simultaneously.
2. The processing elements must communicate (transfer information) between themselves, though there is a cost (time delay) associated with this transfer. A system that treats interprocess communication as if it were free is a shared-memory, or tightly coupled, system. Such systems avoid many of the difficulties of distribution.
3. No process should be able to perceive the global state or global time of the system. Once again, a system that shares too much information is not truly distributed.
4. Communication should be directed: communications should have both a specific origin and a specific destination. This implies that (pseudo-) broadcast communication mechanisms are somewhat suspect. If a system wishes to use *broadcast* as a syntactic shorthand for a series of directed communications, then the cost of that broadcast should be proportional to the number of destinations.
5. If a system has an explicit notion of process, programs written in that system should be able to create processes dynamically.

These criteria are software criteria, though they have their origins in the nature of physical computing devices. Our goal is to define a distributed software system to be the programming equivalent of a multiple-processor hardware system, where the processors, though independent, share the work involved in some task. These restrictions are designed to limit discussion to languages and models that can support coordinated problem solving on such systems.

There are no mature software systems that exhibit our idea of distribution. Instead, several languages and models have been proposed for dealing with various aspects of distribution. We feel that these languages and models can be characterized by the choices they make in a multidimensional decision space. In this section we discuss the dimensions of that space and plot the location of our sample languages.

Candidate Models and Languages

Models are used to describe mathematical relationships. Programming languages are used to describe processing. Since the mathematical relationships involved in programming and modeling a processing task are similar, some of our

examples have aspects of both programming language and model.

Models are constructed to explain and analyze complex system behavior. It is important that a model abstract out the "interesting" part of the system it is attempting to model. Models of concurrent systems usually specify some properties of the interprocess communication mechanism; they are then used to prove properties of the resulting systems. For example, one can set up a model of the communication relationships among processes and use it to analyze the efficiency of an algorithm; or one can set up a model of the information transfer among processes and use it to prove an algorithm's correctness.

Informally, a computer language is a way of providing a sufficiently exact set of directions to a computer. Computer languages are characterized by their syntax and semantics. The syntax of a programming language defines the appearance of the set of legal program strings. The semantics of a programming language specifies the effects of a particular syntactic structure. The details of a programming language syntax (such as the choices of keywords and punctuation) are unimportant (except for issues of human engineering). Instead, it is the semantic actions the programming language can take that interest us.

There have been many proposals for models and languages for distributed processing. This paper contains a brief comparison of 11 of these proposals. We have selected what we feel is a representative sample of ideas from the important systems. Of these systems, we characterize four of these as pure models: Milne and Milner's "concurrent processes,"[2] Fitzwater and Zave's "exchange functions,"[3] and Lynch and Fischer's "shared variable" model[4] and "data flow," by Dennis and by Arvind et al.[5,6]* Typically, models for distribution describe only the communication relationships between processes, without placing limitations on the architecture of the remainder of a system.

Three of our examples are model-language hybrids: Hewitt's Actors,[7] Friedman and Wise's frons,[8] and Hoare's Communicating Sequential Processes.[9] Hybrids specify more of the computing process than models but are not as comprehensive as languages.

Our final four examples are programming languages: Brinch Hansen's Distributed Processes,[10] Feldman's PLITS,[11] Andrew's Synchronizing Resources,[12] and the Defense Department's Ada.[13]

Dimensions of Distributed Languages and Models

Every designer of a model or language for distributed computing chooses the facilities that that system will provide. In this section we identify several such dimensions and indicate where each model and language lies in the choice space.** Table I examines the general goals and structure of each system, Table II examines aspects of intrasystem communication, and Table III examines the system perspective of the remaining dimensions.

TABLE I—Goals and structures

Model	(A) Task domain	(B) Explicit Processes	(C) Dynamic Process Creation
Concurrent processes, Milne & Milner	Correctness	Processes	Dynamic, new
Exchange functions, Fitzwater & Zave	Pragmatics (RS)	Processes	Static
Shared variable, Lynch & Fischer	Correctness, Analysis	Processes	Static
Data flow, Dennis; Arvind et al.	Pragmatics	Tasks	
Actors, Hewitt	Pragmatics (AI), Correctness	Processes	Dynamic, new
frons, Friedman & Wise	Pragmatics, Correctness	Tasks	
CSP, Hoare	Systems, Correctness	Processes	Static
Distributed processes, Brinch Hansen	Systems	Processes	Static
PLITS, Feldman	Pragmatics (AI), Systems	Processes	Dynamic, new
Synchronizing resources, Andrews	Systems	Processes	Static
Ada, DoD	Systems, Pragmatics	Processes	Dynamic, new, lexical

1. Task domain: the most dramatic differences between these languages and models appears in their choice of problem domain. The models are directed principally at mathematical concerns, such as proofs of algorithm correctness (Correctness) and analysis of algorithmic complexity (Analysis). Some of these systems are concerned with issues of systems implementation (Systems). Some of the language proposals are pragmatic—their authors feel that the choice of constructs eases the task of programming distributed systems (Pragmatics). Two of our pragmatic systems have particular interest in programming problems from artificial intelligence (AI); one is principally concerned with the software engineering problem of requirement specification (RS). Many systems have features directed at several of these task domains.

2. Processes: Most models and languages have an explicit process entity (Process). Others view tasks as the creatures of program execution, to be solved by the system as a whole, without keeping the notion of explicit, communicating processes (Tasks).

3. Dynamics: In any system the set of processes can either be statically determined at system generation (Static), or dynamically created during system execution (Dynamic). All of the systems studied that have dynamic process creation can

*Many workers have worked on data flow systems; there are important differences between the models they have developed. Here we cite only a pair of references. The semantics of data flow models depends on which data flow model is used. Later comparisons will develop this theme.

**Other papers that have engaged in comparative discussion of distributed languages include Mohan,[14] Rao,[15] and a predecessor of this paper.[16]

allocate new processes; one can also generate them by the recursive, lexical expansion of program text (Lexical). Task-based systems can, of course, dynamically create new tasks.

Most languages that allow dynamic process creation restrict the new processes to a type determinable at system initiation (compilation). Some systems allow the creation of new varieties of processes and tasks during execution (new). A system that creates new processes invariably provides names for (or, equivalently, pointers to) these new processes. These names can be passed around the process network, creating new communication channels. Systems that rely on a static network of processes usually determine interprocess communication paths lexically.

4. Synchronization: Systems with explicit processes choose between synchronous communication, where all communicators must attend to every communication (Synch) and asynchronous communication, where processes can begin a communication and continue with other activities (Asynch). This is the difference between "call" and "send-and-forget." Following Ada,[13] we call the period of synchronization in communication *rendezvous*.

5. Buffering: A system that supports asynchronous communication can place a bound (Bounded) on the size of the communication buffer or can allow an unbounded (Unbounded) number of messages to be initiated. Systems that require synchronous communication have no use for unbounded message buffers; every message is processed when it is sent.

TABLE II—Communication

Model	(D) Synchronization	(E) Buffering	(F) Information flow	(G) Control	(H) Syntactic-Conn. Sender	Receiver
Concurrent processes	Synch	Bounded	Bi-Sim	Equal	Port	Port
Exchange Functions	Synch	Bounded	Bi-Sim	Equal	Port	Port
Shared variable	Asynch	Bounded	Uni	Equal	Port	Port
data flow[a]	Asynch	Bounded/ Unbounded	Uni	Act-Pas	Entry	none
Actors	Asynch	Unbounded	Uni	Act-Pas	Name	none
frons	Asynch	n.a.	Uni	n.a.	Port	Port
CSP	Synch	Bounded	Uni	Act-Act I/O-G	Name pattern—match	Name
Distributed processes	Synch	Bounded	Bi-Del	Act-Act I-G, Pat	Entry	none
PLITS	Asynch	Unbounded	Uni	Act-Act[b] Filter	Name	sender filter
Synchronizing resources[c]	Asynch/ Synch	Unbounded/ Bounded	Bi-Del Uni	Act-Act[d] I-G, Ex-Sr	Entry	Entry
Ada	Synch	Bounded	Bi-Del	Act-Act I-G, Tm-O	Entry	Entry

[a]Different data flow models make different choices regarding infinite buffering.
[b]PLITS processes can filter messages by sender or "transaction key."
[c]Synchronizing resources supports both synchronous (call) and asynchronous (send) mechanisms.
[d]Synchronizing resources can examine and sort calls before selecting which to serve.

TABLE III—Other Issues

Model	(I) Time & Consciousness	(J) Fairness	(K) Failure	(L) Supports Shared Memory
Concur. Proc.	Rescindable Offer	Anti		
Exch. Funct.	Instantaneous Time Outs	Strong		
Shared Var.	Always Conscious	Weak		Supports
data flow	Reactive	-n.a.-[a] /Anti		
Actors	Reactive	Weak		
frons	Reactive	Weak	Convenient redundancy	
CSP	I/O Guards[b]	Anti		
Dist. Proc.	I-Guards	Strong		
PLITS	Always Conscious	Strong		
Synch. Res.	I-Guards	Anti		Supports
Ada	I-Guard, Time Outs	Strong	Extensive mechanisms	Supports

[a]Some data flow models are deterministic. In a deterministic system, fairness is irrelevant.
[b]Hoare's earliest proposals excluded output guards. Later works on CSP have included them.

6. Information Flow: In communication, information flow can be unidirectional (Uni) (from one process to another), bidirectional simultaneous (between processes only at the synchronous time of communication) (Bi-Sim), or bidirectional delayed (where one process can compute a reply during rendezvous) (Bi-Del). None of our models or language provides for bidirectional delayed communication where both processes compute during rendezvous, though there is no theoretical reason to disallow it.

7. Control: The communication process can be initiated by an active caller to a passive receiver (Act-Pas), by an active caller to an active receiver (Act-Act), or by two equal communicators (Equal). The receiver of a request sometimes has control over the order in which the requests are processed. In the languages and models studied, this control includes input and/or output guards (I/O-G), time out on lack of response (Tm-O), choice of certain classes of requests (Choice), pattern matching on messages (Pat), and selective search and examination of all pending messages (Ex-Sr).

8. Connection: Communicants can refer to a named port or channel that is external to all processes (Port), the name of process itself (Name), or a port within a called process (Entry). The chart details the naming required of both the message sender and the message receiver for systems that do not treat communicators equivalently.

9. Time and consciousness: In synchronous communication, the process that initiates a communication may be forced

to complete that communication, or it may have some facility for aborting the communication (such as a time out). We say that a process activity that causes uninterruptable waiting is a loss of consciousness for a process. Languages and models sometimes provide mechanisms by which a calling process can regain control. These mechanisms include instantaneous time outs, time outs, input and output guards (I/O-Guards), input guards (I-Guards), and rescindable offers. A process that never loses consciousness is always conscious; a process that only awakens when invoked is reactive.

10. Fairness: Systems can strive for fairness. Sometimes this notion of fairness is a weak fairness, the idea that every attempted action eventually gets its turn (Weak). Alternatively, a system can specify a stronger notion of fairness, asserting that each process will get its "rightful" turn (Strong). Stronger fairness can usually be implemented with queues. On the other hand, a formalism may make no claims about fairness at all (Anti).

11. Failure: Most models and languages treat processes as perfect computers and communication as invariably secure (Fail). Some of the systems have some mechanisms for dealing with process and communication failure.

12. Shared Memory: Some of the systems have explicit shared-memory mechanisms, in addition to distributed ones.

HEURISTICS FOR COORDINATION

Incremental Computation

Despite the paucity of failure mechanisms in our models and languages, any real system needs mechanisms to cope with failures. On the statement level, these mechanisms need to handle the disruptions of lost messages and failing processes. On a more global level, a profitable organization of a distributed system may be performed, not as a sequential program, but as a set of computing "agents" who make progress toward solving subtasks.

The idea of useful progress may seem a foreign notion. Most conventional programming languages are a-step-at-a-time, imperative formulations. The validity of the successive steps is entirely dependent on the successful completion of the previous steps. However, there are other possible formulations for expressing computable functions. Production systems (such as Newell's[17]) are one example. A production system consists of two parts: a working memory and a set of productions. Each production has two pieces, a *pattern* and an *action.* When some part of the working memory matches the pattern of a particular production, that production fires, executing its action. Actions are programs; they typically add elements to the working memory. Elements are never removed from the working memory. Thus, the firing of a production never makes another production's firing cease to be valid. One possible organization of a distributed system is a set of productions that communicate through a working memory. The "blackboard" of the Hearsay-II model[18] is an example of such a central communication depository.

Other examples of computing systems that make progress include theorem proving[19] and suspending evaluation in Lisp-like systems.[8] In a theorem-proving system, the proof of a theorem does not invalidate the truth of any other theorem. A task expressed as a theorem to be proved can be worked on by many inference rules at the same time. In a suspending CONS system, tasks are created as the natural action of computation. When a processing element finishes a task, it "stings" that task with its value.[8] Sting is an interlock-free test-and-set primitive. If the particular object to be stung has already been stung (by someone else), the operation becomes a "no-op." Thus, if a swarm of processes are working on a problem, the first sting of the answer is permitted to succeed. Later stings have no effect; thus, the system exhibits functional behavior throughout.

Programming languages predicated on this idea of incremental discovery can be more easily distributed than systems that require the standard sequentiality.

Economic Models

Distributed computing networks are not the only organizations that require internal cooperation and communication. Human economic activity shows both some of the same requirements and some of the same goals as a distributed computing network. There are some interesting parallels between human economic systems and potential organizational models for distributed systems.

How are economies organized? One important dimension is centralization. In a centralized system, there is a master directorate (node) that sets the goals of the system and divides the task into subpieces, with each subtask specified for a particular worker. When the task is modular and well defined, it is possible to organize a distributed system in this fashion. Efficiency can be achieved in such a structure if the task is well understood, and the initial allocation of subgoals and resources can be made to reflect this understanding. However, central planning does not lend itself well to ill-defined problems. Additionally, there may be a communication overload from the planning node to the workers while most of the communication ability of the system—between the working nodes—goes unused.

It is only a small step from a fully centralized economy to a partially centralized (hierarchical) model. The central authority defines the major tasks. These are parceled out to regional subauthorities, each of whom is allotted a resource of workers. This structure can be iterated. At the limit, it resembles a corporate hierarchy tree. Hierarchical organization can respond well to local aberrations. However, its response to dramatic global changes is somewhat slower, because command must filter through several command layers. Hierarchical systems are better matched to the physical distribution of the world than systems based on pure centralized control.

An alternative approach to processor organization is a laissez-faire economy. Each task has certain goals and an allocation of currency. Currency can be used to purchase processor power and to generate new tasks. When a task has exhausted its currency, it can appeal to its own source (banker) for more. Its banker can then decide, on the basis of the results that the task presents, whether to grant that task more resources. The scheme can be applied recursively, to the

banker's banker, and so forth, back to the resources of the human being who originated the request. Such a scheme lends itself to ill-defined tasks—ones where a promising line can be recognized but not necessarily generated—and to useful-progress programming models. Though such systems are not fragile, there are difficulties both in focusing the organization in the presence of a rapidly changing environment and in terminating the activity of tasks that have ceased to be useful. A variation of this mechanism was used by the agenda priority schemes of Lenat.[20] Our development parallels a remark of Hewitt.

One could also imagine distributed systems organized as mixed economies (partially centralized and partially free market) or as indicative planned systems (with centralized goals and directives shaping a free-market economy.)

Another proposal for organizing distributed computing is Smith's contract nets.[21] Processes that have subproblems broadcast their request to the other processes. A free process, or one that has particular knowledge about that task, "bids" to obtain the contract. Contract nets are a protocol; Smith does not elaborate on how particular tasks would be organized in the contract net approach.

CONCLUSIONS

Distribution promises inexpensive and efficient computation. To realize that promise, much work needs to be done both to define the right models for distribution and to select the appropriate algorithms for apportioning computations among processes.

ACKNOWLEDGMENTS

We thank John Barnden, Jim Burns, Mitch Wand, and Dave Wise for comments on earlier drafts of this paper. Research reported herein was supported (in part) by the National Science Foundation under grants numbered MCS77–22325, MCS79–04183, and MCS81–02291.

REFERENCES

1. Filman, R. E., and D. P. Friedman. *Coordinated Computing: Tools and Techniques for Distributed Software.* To be published by McGraw-Hill, New York, 1982.
2. Milne, G., and R. Milner. "Concurrent Processes and Their Syntax." *Journal of the ACM,* 26 (1979), pp. 302–321.
3. Fitzwater, D. R., and P. Zave. "The Use of Formal Asynchronous Process Specifications in a System Development Process." *Proceedings of the Sixth Texas Conference on Computing Systems,* November 14–15, 1977. Austin, Texas: University of Texas at Austin, 1977. pp. 2B–21:2B–30.
4. Lynch, N. and M. Fischer. "On Describing the Behavior and Implementation of Distributed Systems." *Theoretical Computer Science,* 13 (1918), pp. 17–43.
5. Dennis, J. B. "First Version of a Data Flow Procedure Language." In B. Robinet (ed.), *Programming Symposium,* Paris, April 9–11, 1974. Berlin: Springer, 1974, pp. 362–376.
6. Arvind, K. Gostelow, and K. Plouffe, "The (Preliminary) ID Report: an Asynchronous Programming Language and Computing Machine." Technical Report 114, Department of Information and Computer Science, University of California, Irvine, May 1978.
7. Hewitt, C., G. Attardi, and H. Lieberman. "Security and Modularity in Message Passing." *Proceedings of the First International Conference on Distributed Computing Systems,* Huntsville, Alabama, October 1–5, 1979. Long Beach, California: IEEE Computer Society, 1979, pp. 347–358.
8. Friedman, D. P., and D. S. Wise. "An Approach to Fair Applicative Multiprogramming." In G. Kahn (ed.), *Proceedings of International Symposium on Semantics of Concurrent Computation,* Evian, France, July 2–4, 1979. Berlin: Springer, 1979, pp. 203–225.
9. Hoare, C. A. R. "Communicating Sequential Processes." *Communications of the ACM,* 21 (1978), pp. 666–677.
10. Brinch Hansen, P. "Distributed Processes: a Concurrent Programming Concept." *Communications of the ACM,* 21 (1978), pp. 934–941.
11. Feldman, J. A. "High Level Programming for Distributed Computation." *Communications of the ACM,* 22 (1979), pp. 353–367.
12. Andrews, G. "Synchronizing Resources." *ACM Transactions on Programming Languages and Systems,* 3 (1981), pp. 405–430.
13. Department of Defense. "Military Standard Ada Programming Language." Dept. of Defense Standard MIL-STD-1815, December 1980.
14. Mohan, C. "A Perspective of Distributed Computing: Models, Languages, Issues & Applications." Working Paper DSG-8001, Department of Computer Science, University of Texas, March 1980.
15. Rao, R. "Design and Evaluation of Distributed Communication Primitives." Technical Report 80–04–01, Department of Computer Science, University of Washington, April 1980.
16. Filman, R. E., and D. P. Friedman. "Inspiring Distribution in Distributed Computing." Technical Report 99, Computer Science Department, Indiana University, December 1980.
17. Newell, A. "Production Systems: Models of Control Structures." In W. Chase (ed.), *Visual Information Processing.* New York: Academic Press, 1972, pp. 463–526.
18. Erman, L. D., F. Hayes-Roth, V. R. Lesser, and D. R. Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty." *ACM Computing Surveys,* 12 (1980), pp. 213–254.
19. Loveland, D. W. *Automated Theorem Proving.* Amsterdam: North Holland, 1978.
20. Lenat, D. B. "Automated Theory Formation in Mathematics." *Proceedings of the 5th International Joint Conference on Artificial Intelligence,* MIT, Cambridge, Massachusetts, August 22–25, International Joint Conferences on Artificial Intelligence, 1977, pp. 832–842.
21. Smith, R. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver." *Proceedings of the First International Conference on Distributed Computing Systems,* Huntsville, Alabama, October 1–5, 1979. Long Beach, California: IEEE, 1979, pp. 185–192.

Weakest environment of communicating processes

by ZHOU CHAOCHEN
Institute of Computing Technology, Academia Sinica
Peking, China

ABSTRACT

As is well known, the concept of the weakest precondition[1] has played an important role in sequential programming. In this paper we introduce a similar concept for distributed programming. As far as partial correctness[2] is concerned, given an overall specification of a distributed system and of a designated part of the system, we can find a minimum specification that must be met by the rest of the system in order that the whole system meet the overall specification. This minimum specification is called the *weakest environment* of the first designated part with respect to the overall specification. In terms of weakest environment, a calculus for the partial correctness of processes with a master-slave communication mechanism is also given.

INTRODUCTION

$(P\ wp\ R)$ denotes the weakest precondition for partial correctness of program P with respect to postcondition R. $(P\ wp\ R)$ identifies the minimum condition that must be satisfied by the machine state *before* the execution of P if the machine state *after* the successful execution of P is to satisfy R. Weakest precondition is an important idea in sequential programming. It has been used as a semantics of sequential programming languages, as a proving technique for the correctness of sequential programs, and also as a rigorous approach to developing sequential programs.

This paper will define a similar concept for distributed programming. In sequential programming the sequential operator " ; " takes a peculiar part in combining program segments into a program. The weakest precondition can be understood in the following way. Given an overall specification R of a program and a last segment P of that program, we may ask what is the minimum specification that must be met by the other segment of the program in order that the whole program meet its specification R. This is nothing other than $(P\ wp\ R)$.

Let P and Q be programs, and let R be an assertion of machine states. Let "P satisfies R" stand for "Starting from any initial machine state, if P terminates, then the resultant machine state satisfies R." Then we can define $(P\ wp\ R)$ as follows: $(P\ wp\ R)$ is an assertion of machine states such that $(Q;\ P)$ satisfies R iff Q satisfies $(P\ wp\ R)$.

In a communicating process, a process may be constructed from a group of parallel subprocesses. So the parallel combinator $\|$ can combine subprocesses into a process even as the sequential operator does for sequential programming.

Let P and Q be processes. Then $(P \| Q)$ is a process constituted by P and Q in parallel. P and Q regard each other as the environment within which they fulfill a certain task. Thus a similar question arises: Given a specification R of the overall task and a subprocess P, can we formulate a minimum condition that must be met by the environment of P in order that the process constituted by P and its environment can meet R? This minimum condition is called the *weakest environment* and is denoted as

$$(P\ we\ R)$$

So $(P\ we\ R)$ is a specification such that $(P \| Q)$ satisfies R iff Q satisfies $(P\ we\ R)$.

In the following sections we give an answer to this question.

The programming notation for communicating processes presented in the first section is oriented to the master-slave communication structure. In this structure communication occurs only between a master and its slave. A master may have several slaves, but each slave belongs to only one master. A master may serve a supermaster as its slave, and a slave may employ its own slaves as well. Of course it is not allowed that a slave is also a supermaster of its master in this rank system. The denotational semantics of the notation is given in the same way as it was in Zhou and Hoare.[2]

In Section 2, *process predicate* is used as specification language. Process predicate is something like the *channel predicate* of Zhou and Hoare,[2] but it uses a process name instead of a channel name. A process name stands for a message sequence that records all messages that have passed so far to (or from) the process of this name.

In Section 3, a formal definition of *weakest environment* is introduced. Weakest environment identifies the minimum condition a master (or a slave) must meet in order to meet a given overall specification with its designated slave (or its designated master).

A calculus for the partial correctness of communicating processes is also developed in terms of weakest environment in Section 4.

The last section, Section 5, gives a proof of partial correctness in this calculus.

COMMUNICATING PROCESSES

We briefly recall the programming notation of communicating processes and its denotational semantics in this section.

A process is constructed from a group of subprocesses, intercommunicating on a network of master-slave structures. *Communication* is the atomic action of a process.

For a process there are two different kinds of communications. One occurs between the process and its slaves, and the other one between the process and its master. Since a master can have several slaves, a process calls its slaves by their names. The pair $s.e$ is used to denote a communication occurring between a process and one of its slaves, where s is the name of the slave engaged in this communication and e is the value of the message being passed in this communication. But any process can have only one master; that is, the master of a process is unique to the process. Hence it is not necessary to use names to identify masters. The pair *master.e* is generally used to denote a communication between a process and its master, where *master* is a special name that is different from any slave name, and e is the value of the message in the communication.

A communication is an atomic action of a process, so a finite sequence of communications is used to record the behavior of a process from its beginning up to some moment in time. Such a sequence of communications is called the *trace* of the behavior of a process. Finally, a *process* is defined as the set of all traces of its behavior.

Let us now take a buffer as an example, to show some possible traces of it. This buffer gets messages from its slave, named servant, and provides its master with them:

1. < > (empty sequence) records the behavior before it actually evolves.
2. < servant. 3 > is a trace recording the behavior when it has input a message of value 3 from slave.
3. < servant. 3, master 3 > records the behavior when it has input value 3 and then output it.
4. < servant. 3, servant. 4, master. 3, servant. 1, master. 4 > is another possible trace.

The programming notation presented in the following is an applicative language including the basic constructs: output, input, alternation, naming, parallelism, and recursion. The semantics of processes is defined by a function $\varsigma[\![\cdot]\!]$ that maps a process into its trace set.

1.1 STOP

The process STOP is one that communicates neither with its master nor with its slaves; i.e.,

$$\varsigma[\![\mathrm{STOP}]\!] =_{df} \{<\ >\}$$

1.2 Output

Let z be a process name (a slave name or "master") and e be a message, and let P be a process. Then $(z!e \rightarrow P)$ is also a process, which just outputs e to Process z, and then behaves like P; i.e., the possible trace of $(z!e \rightarrow P)$ is headed by $z.e$, and the rest of the sequence is a trace of P.

So $\varsigma[\![z!e \rightarrow P]\!] =_{df} \{<\ >\} \cup \{z.e\hat{}t \mid t \in \varsigma[\![P]\!]\}$.

1.3 Input

Let z be a process name, x be a variable, and M be a message set, and let $P(x)$ be a process for any x in M. Then $(z?x{:}M \rightarrow P(x))$ is a process that first inputs any message x of type M from process z, and then behaves like $P(x)$.

Thus

$$\varsigma[\![z?x{:}M \rightarrow P(x)]\!] =_{df} \{<\ >\} \cup \{z.x\hat{}t \mid x \in M \& t \in \varsigma[\![P(x)]\!]\}.$$

1.4 Alternation

Let P and Q be processes. Then $(P|Q)$ is a process that behaves either like P or like Q.

$$\varsigma[\![P|Q]\!] =_{df} \varsigma[\![P]\!] \cup \varsigma[\![Q]\!]$$

1.5 Naming

A process may employ another process as a slave by giving a slave name, say s, to the employed process. In this case the named process should be prepared to communicate with its master, which will call it by the name s. Thus, for synchronization, the communication of the named process with its master (which has the form $m.e$) must be regarded as the same event as the communication of its master with the named slave (which has the form $s.e$). This can be realized by replacing each occurrence of m by an occurrence of s in the named process.

Let P be a process, and let s be a slave name that does not occur in P. Then $(s{:}P)$ is also a process, and

$$\varsigma[\![s{:}P]\!] =_{df} \varsigma[\![P]\!]\,[s/m]$$

where m is an abbreviation of *master*.

1.6 Parallelism

Let A be the set of names of slaves that Process P wants to communicate with, and let B be the set of names of slaves that Process Q wants to communicate with, where $A \cap B = \emptyset$ (since no slave is allowed to serve more than one master). Suppose that P employs Q as its slave by giving Q a slave name, s, of A. Then $(P \|_{AB}\, s{:}Q)$ is a new process, which is constructed by P employing Q as its slave. This new process has a new list of slave names that it still wants to communicate with: $(A \cup B) - \{s\}$. Thus, in $(P \|_{AB}\, s{:}Q)$, each communication between P and its slave s must be synchronized by the communication between Q and its master. But each possible communication between $(P \|_{AB}\, s{:}Q)$ and its master or its slaves in $(A - [s])$ can be made by P itself and has nothing to do with Q. The communication between $(P \|_{AB}\, s{:}Q)$ and its slaves in B, which actually are slaves of Q, is still dominated by Q and has nothing to do with P. So the traces of $(P \|_{AB}\, s{:}Q)$, before canceling s from the slave list, should be

$$T = \{t \mid t \in (A \cup B \cup \{m\})^* \;\&\; t\upharpoonright A \cup \{m\} \in \varsigma[\![P]\!] \;\&\; t\upharpoonright B \cup \{s\} \in \varsigma[\![s{:}Q]\!]\},$$

where m is an abbreviation of *master*, X^* stands for all the finite sequences of communications prefixed by the process names in X, and $t\ X$ stands for the sequence obtained from t by canceling all the communications prefixed by a process name out of X. Then canceling s from the slave list, which $(P \|_{AB}\, s{:}Q)$ still wants to communicate with, can be realized by

$$\varsigma[\![(P \|_{AB}\, s{:}Q)]\!] =_{df} \{t/\{s\} \mid t \in T\}$$

where $t/\{s\}$ is obtained from t by canceling the communication prefixed by s. Sometime we use $T/\{s\}$ to stand for the right-hand side.

1.7 Recursion

A process expression is an expression made by process variables (p, q, etc.) and the previous six operators. Then $p \underline{\triangle} F(p)$ is a single recursion, where $F(p)$ is a process expression including process variable p. The expression $p[i{:}S] \underline{\triangle} F(p)$ is a vector form of mutual recursion, where for any $i \in S$, $p[i]$ is a process variable, and $F(p)[i]$ is a process expression including some of the process variables $p[j]$ $(j \in S)$.

The process (or array of processes) defined by recursion $p \underline{\triangle} F(p)$ (or $p\ i{:}S \underline{\triangle} F(p)$) is the least fixed point of the recursion, denoted as $\mu p\,.\,F$ (or $\mu p[i{:}S]\,.\,F$).

In Zhou and Hoare[2] a partial order of trace sets has been defined by the inclusion relation of sets, and the limit of a monotonic sequence of trace sets has been defined as the infinite union of this sequence:

$$\lim \varsigma[\![P_i]\!] =_{df} \bigcup_i \varsigma[\![Pi]\!]$$

where $\forall i.\ (\varsigma[\![P_{i+1}]\!] \supseteq \varsigma[\![P_i]\!])$.

Furthermore, we have also shown that all the operators are continuous. So the least fixed point of recursion can be defined as

$$\varsigma[\![\mu p.\ F]\!] =_{df} \bigcup_n \varsigma[\![F^n(\text{STOP})]\!]$$

and $\varsigma[\![(\mu p[i{:}S].\ F)[j]]\!] =_{df} \bigcup_n \varsigma[\![(F^n(\text{STOP}))[j]]\!]$ $(j \in S)$.

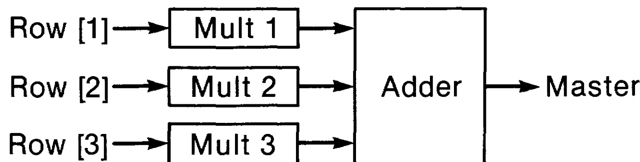
In what follows we always use p as an abbreviation of the least fixed point defined by recursion $p \triangle F(p)$, where it will not cause confusion.

Example. A Matrix Multiplier

To realize a multiplication for a matrix of three rows by a three-dimensional vector

$$(v_1, v_2, v_3)\begin{pmatrix} x_{11} & x_{12} & x_{13} & \ldots \\ x_{21} & x_{22} & x_{23} & \ldots \\ x_{31} & x_{32} & x_{33} & \ldots \end{pmatrix}$$

it is desirable to input three numbers at a time, and multiply these numbers simultaneously also. So an algorithm is suggested as



where each mult[i] (i = 1,2,3) inputs a number from its slave row[i] and multiplies it with $v[i]$ (i = 1,2,3), then sends the result to its master adder. The process adder forms the sum as required and outputs it to its master.

multiplier $\underline{\triangle}$

$$(((\text{adder} \underset{A_1B_1}{\|} \text{mult}[1]{:}\ p[1]) \underset{A_2B_2}{\|} \text{mult}[2]{:}\ p[2]) \underset{A_3B_3}{\|} \text{mult}[3]{:}\ p[3]),$$

where $B_i = \{\text{row}[i]\}$ $(i = 1,2,3)$, $A_1 = \{\text{mult}[1],\ \text{mult}[2],\ \text{mult}[3]\}$,
$A_2 = (A_1 \cup B_1) - \{\text{mult}[1]\}$ and $A_3 = (A_2 \cup B_2) - \{\text{mult}[2]\}$,

and

$$\text{adder} \underline{\triangle} \text{mult}[1]?x{:}\text{NAT} \rightarrow \text{mult}[2]?y{:}\text{NAT} \rightarrow \text{mult}[3]?z{:}\text{NAT} \rightarrow \text{master}!(x+y+z) \rightarrow \text{adder},$$
$$p[i{:}1..3] \underline{\triangle} \text{row}[i]?x{:}\text{NAT} \rightarrow \text{master}!(v[i]*\text{x}) \rightarrow p[i].$$

SPECIFICATIONS

We adopt process predicates as assertions. A process predicate is a predicate with process names as free variables, where a process name denotes the sequence of messages communicated by the process of this name up to some moment in time. For example, let u and v be strings, and let $u \leq v$ mean that u is an initial segment of v. Then the assertion

$$\text{master} \leq \text{servant}$$

means that the message sequence communicated by a process to its master is a copy of the message sequence communicated by the process to its slave, called *servant.* This assertion should always be true of a buffer process that gets messages from its slave, *servant,* and provides its maser with them; therefore we can take it as a specification of the buffer process.

Let P be a process and $R(m, z_1, \ldots, z_n)$ be a process predicate with the free process variables *master* and z_i ($i = 1, \ldots, n$). Then we define P *sat* R as meaning that R is true of any possible behavior of P. Since we use a trace to record the behavior of P, the definition of P *sat* R is

$$P\ sat\ R\ (m, z_1, \ldots, z_n) =_{df} \forall t \in \varsigma[\![p]\!].\ R(t \upharpoonright m,\ t \upharpoonright z_1, \ldots, t \upharpoonright z_n),$$

where $t\ z$ stands for a message sequence that is obtained from t by leaving only the communication of Process z and then dropping the process name z to get a pure message sequence. (Note: $t \upharpoonright z$ and $t \upharpoonright \{z\}$ are different. The first is a sequence of messages, whereas the second is a sequence of communications.)

Thus, for example, we can hope that

$$\text{buffer}\ sat\ (\text{master} \leq \text{servant}),$$

and

$$\text{multiplier}\ sat\ \forall i \in \{1..\#\text{m}\}.$$

$$\text{m}_i = \sum_{j=1}^{3} \text{v}[j] * \text{row}[j]_i \ \overset{3}{\underset{j=1}{\&}}\ \#m \leq \neq \text{row}[j],$$

where $\#t$ is the length of sequence t, and t_i is the ith member of sequence t.

One of the disadvantages of the process predicate is that we cannot use it to describe liveness of a process, just as partial correctness of a sequential program cannot deal with termination of a program. So process predicate, too, is only concerned with the partial correctness of a distributed program.

Let P be a process and $\{z_1, \ldots, z_n\}$ be the slave list of P. Then what is the most precise specification of P that we can define in terms of process predicate? This should be nothing but just all the possible message sequences to its master and slaves. Let $P(m, z_1, \ldots, z_n)$ denote the required process predicate. Then

$$P(x_o, x_1, \ldots, x_n) =_{df} \exists t \epsilon \varsigma[\![P]\!].\ x_o = t \upharpoonright m \ \underset{i=1}{\overset{n}{\&}} x_i = t \upharpoonright z_i$$

This claim can be justified by the following theorem:

Theorem.

If P is a process of slaves $\{z_1, \ldots, z_n\}$, and R is an assertion of process names $\{m, z_1, \ldots, z_n\}$, then

$$P \textit{ sat } R \text{ iff } \forall x_o, \ldots, x_n \quad P(x_o, \ldots, x_n) \Rightarrow R(x_o, \ldots, x_n)$$

Proof. ($\Rightarrow$) Suppose P *sat* R; *i.e.,*

$$R(t \upharpoonright m, \ldots, t \upharpoonright z_n)$$

is true for any t of $\varsigma[\![P]\!]$.

Then

$$P(x_o, \ldots, x_n) \Rightarrow \exists t \epsilon \varsigma[\![P]\!].\ x_o = t \upharpoonright m \ \underset{i=1}{\overset{n}{\&}} x_1 = t \upharpoonright z_i \quad \text{(def.)} \quad \Rightarrow R(x_o, \ldots, x_n)$$

($\Leftarrow$) Suppose $\forall x_o, \ldots, x_n.\ P(x_o, \ldots, x_n) \Rightarrow R(x_o, \ldots, x_n)$.

Then for any $t \epsilon \varsigma[\![P]\!]$, $R(t \upharpoonright m, \ldots, t \upharpoonright z_n)$ is true by the assumption, since $P(t \upharpoonright m, \ldots, t \upharpoonright z_n)$ is true; i.e., (P *sat* R) is true.

We now derive the structural formulas for process specification.

Theorems.

2.1 STOP

STOP is an inactive process, so it always leaves all communications with its master and slaves empty:

$$\text{STOP}(m, Z) \equiv m = <\,> \,\&\, Z = <\,>$$

where Z stands for $\{z_1, \ldots, z_n\}$, and $Z = <\,>$ stands for $\underset{i=1}{\overset{n}{\&}} z_i = <\,>$.

Proof.

$$\begin{aligned}\text{STOP}(x_o, \ldots, x_n) &\equiv \exists t \epsilon \varsigma[\![\text{STOP}]\!].\ x_o = t \upharpoonright m \ \underset{i=1}{\overset{n}{\&}} x_i = t \upharpoonright z_i \quad \text{(def.)}\\ &\equiv \exists t \epsilon \{<\,>\}.\ x_o = t \upharpoonright m \ \underset{i=1}{\overset{n}{\&}} x_1 = t \upharpoonright z_i \quad (1.1)\\ &\equiv x_o = <\,> \upharpoonright m \ \underset{i=1}{\overset{n}{\&}} x_i = <\,> \upharpoonright z_i\\ &= x_o = <\,> \ \underset{i=1}{\overset{n}{\&}} x_i = <\,>.\end{aligned}$$

$$\begin{aligned}\text{STOP}(m, Z) &\equiv \text{STOP}(x_o, \ldots, x_n)[m/x_o][z_i/x_i]\ _{i=1}^{n}\\ &\equiv m = <\,> \,\&\, Z = <\,>.\end{aligned}$$

2.2 Output

The possible message sequence of $(z!e \rightarrow P)$ is empty (before its evolution), or has value e prefixed to the sequence of z, whereas the rest is from P.

$$\begin{aligned}(z!e \rightarrow P)(m, Z) \equiv \text{STOP}(m, Z) \vee (hd(z)\\ = e \,\&\, P(m, Z)[tl(z)/z]),\end{aligned}$$

where $z \epsilon \{m\} \cup Z$, and $hd(t)$ stands for the first element of t and $tl(t)$ is the tail of t; i.e., $t = hd(t)^\wedge tl(t)$.

Proof.

$$\begin{aligned}(z!e \rightarrow P)(x_o, \ldots, x_n) &\equiv \exists t \epsilon \varsigma[\![z!e \rightarrow P]\!].\\ &\quad x_o = t \upharpoonright m \ \underset{i=1}{\overset{n}{\&}} x_i = t \upharpoonright z_i \quad \text{(def.)}\\ &\equiv \exists t \epsilon (\{<\,>\} \cup \{z.e^\wedge t | t \epsilon \varsigma[\![P]\!]\}).\\ &\quad x_o = t \upharpoonright m \ \underset{i=1}{\overset{n}{\&}} x_i = t \upharpoonright z_i \quad (1.2)\\ &\equiv (x_o = <\,> \ \underset{i=1}{\overset{n}{\&}} x_i = <\,>) \vee\\ &\quad \exists t \epsilon \varsigma[\![P]\!].\\ &\quad x_o = z.e^\wedge t \upharpoonright m \ \underset{i=1}{\overset{n}{\&}}\\ &\quad x_i = z.e^\wedge t \upharpoonright z_i\\ &\equiv \text{STOP}(x_o, \ldots, x_n) \vee \exists t \epsilon \varsigma[\![P]\!].\\ &\quad x_o = t \upharpoonright m\\ &\quad \underset{i \neq j}{\&} x_i = t \upharpoonright z_i \,\&\, x_j = e^\wedge (t \upharpoonright z_j) \text{ (say } z = z_j)\\ &\equiv \text{STOP}(x_o, \ldots, x_n) \vee \exists t \epsilon \varsigma[\![P]\!].\\ &\quad x_o = t \upharpoonright m\\ &\quad \underset{i \neq j}{\&} x_i = t \upharpoonright Z_i \,\&\, hd(x_j) = e \,\&\, tl(x_j) = t \upharpoonright z_j\\ &\equiv \text{STOP}(x_o, \ldots, x_n) \vee (hd(x_j) = e \,\&\\ &\quad P(x_o, \ldots, tl(x_j), \ldots, x_n)).\end{aligned}$$

2.3 Input

$(z?x{:}M \rightarrow P(x))$ is like $(z!e \rightarrow P)$, except that the value prefixed to the sequence z can be any one of the set M.

$$\begin{aligned}(z?x{:}M \rightarrow P(x))(m, Z) \equiv \text{STOP}(m, Z) \vee (hd(z) \epsilon M\\ \&P(hd(z))(m, Z)[tl(z)/z])\end{aligned}$$

where $z \epsilon \{m\} \cup Z$.

Proof. Omitted; it is similar to the proof of 2.2.

2.4 Alternation

The possible message sequences of $(P \mid Q)$ are the union of the message sequences of P and Q. $(P|Q)(m, Z) = P(m, Z) \vee Q(m, Z)$.

Proof. Omitted; it can be immediately obtained from the definition and (1.4).

2.5 Naming

$(s:P)$ is derived from P by replacing m with s; i.e.,

$$(s:P)(s,Z) \equiv P(m,Z)[s/m]$$

where $s \bar{\in} Z$.
Proof. Omitted.

2.6 Parallelism

$(P \underset{AB}{\|} s:Q)$ is composed from P and Q by selecting the trace of P and the trace of Q in which the message sequence of P with respect to its slave s is the same one as Q with respect to its master.

$$(P \underset{AB}{\|} s:Q)(m,Z) \equiv \exists s.\, P(m,A) \,\&\, Q(s,B),$$

where $A \cap B = 0$, $s \in A$, and $Z = (A \cup B) - \{s\}$.

Proof.
Suppose $A = \{z_1, \ldots, z_k, s\}$ and $B = \{z_{k+1}, \ldots, z_n\}$.

$$
\begin{aligned}
(\Rightarrow)\ (P \underset{AB}{\|} s:Q)(x_o, \ldots, x_n) &\equiv \exists t \in \varsigma[\![P \underset{AB}{\|} s:Q]\!].\\
&\quad x_o = t\restriction m \mathop{\&}_{i=1}^{n} x_i = t\restriction z_i \qquad \text{(def.)}\\
&\equiv \exists t \in (T/\{s\}).\\
&\quad x_o = t\restriction m \mathop{\&}_{i=1}^{n} x_i = t\restriction z_i \qquad (1.6)\\
&\equiv \exists t \in T.\\
&\quad x_o = t\restriction m \mathop{\&}_{i=1}^{n} x_i = t\restriction z_i \qquad (\text{by } s \bar{\in} Z \cup \{m\})\\
&\equiv \exists t \in T, y.\\
&\quad x_o = t\restriction m \mathop{\&}_{i=1}^{n} x_i = t\restriction z_i \,\&\, y = t\restriction s\\
&\Rightarrow \exists y.\, (\exists t_1 \in \varsigma[\![P]\!].x_o).\\
&\quad x_o = t_1\restriction m \mathop{\&}_{i=1}^{k} x_i = t_1\restriction z_i \,\&\, y = t\restriction s\\
&\qquad (\text{where } t_1 = t\restriction A \cup \{m\})\\
&\quad \& \, (\exists t_2 \in \varsigma[\![s:Q]\!].y).\\
&\quad y = t_2\restriction s \mathop{\&}_{i=k+1}^{n} x_i = t_2\restriction z_i\\
&\qquad (\text{where } t_2 = t\restriction B \cup \{s\})\\
&\equiv \exists y.\, (\exists t_1 \in \varsigma[\![P]\!].x_o).\\
&\quad x_o = t_1\restriction m \mathop{\&}_{i=1}^{n} x_i = t_1\restriction z_i \,\&\, y = t\restriction s\\
&\quad \& \, (\exists t_3 \in \varsigma[\![Q]\!].y).\\
&\quad y = t_3\restriction m \mathop{\&}_{i=k+1}^{n} x_i = t_3\restriction z_i\\
&\qquad (\text{where } t_3 = t_2[m/s])\\
&\equiv \exists y.\, P(x_o, \ldots, x_k, y)\\
&\quad \& \, Q(y, x_{k+1}, \ldots, x_n)\\
(\Leftarrow)\ \exists y.\, P(x_o, \ldots, x_k, y)&\\
&\quad \& \, Q(y_k, x_{k+1}, \ldots, x_n)\\
&\equiv \exists y.\, (\exists t_1 \in \varsigma[\![P]\!].x_0).\\
&\quad x_o = t_1\restriction m \mathop{\&}_{i=1}^{k} x_i = t_1\restriction z_i\\
&\quad \& \, y = t_1\restriction s)\\
&\quad \& \, (\exists t_2 \in \varsigma[\![Q]\!]\, y).\\
&\quad y = t_2\restriction m \mathop{\&}_{i=k+1}^{n} x_i = t_2\restriction z_i \qquad (def.)\\
&\equiv \exists y.\, (\exists t_1 \in \varsigma[\![P]\!].\, x_0).\\
&\quad x_o = t_1\restriction m \mathop{\&}_{i=1}^{k} x_i = t_1\restriction z_i\\
&\quad \& \, y = t_1\restriction s)\\
&\quad \& \, (\exists t_3 \in \varsigma[\![s:Q]\!].y).\\
&\quad y = t_3\restriction s \mathop{\&}_{i=k+1}^{n} 1 x_i = t_3\restriction z_i\\
&\qquad (\text{where } t_3 = t_2[s/m])\\
&\Rightarrow \exists y.\, \exists t \in T.\\
&\quad x_o = t\restriction m \mathop{\&}_{i=1}^{n} x_i = t\restriction z_i \,\&\, y = t\restriction s
\end{aligned}
$$

(since $t_1\restriction s = t_3\restriction s = y$, t can be obtained from t_1 and t_3 by coordinating the identical subsequences $t_1\restriction s$ and $t_3\restriction s$ and interleaving the other subsequences of t_1 and t_3.)

$$
\begin{aligned}
&\equiv \exists t \in (T/\{s\}).\\
&\quad x_o = t\restriction m \mathop{\&}_{i=1}^{n} x_i = t\restriction z_i\\
&\equiv (P \underset{AB}{\|} s:Q)(x_o, \ldots, x_n).
\end{aligned}
$$

2.7 Recursion

The possible message sequences of the least fixed point of $p \triangle F(p)$ are the infinite union of the message sequences of F^n(STOP).

$$(\mu p.F)(m,Z) \equiv \exists_n.\quad F^n(\text{STOP})(m,Z)$$

and

$$(\mu p\,[i:S].F)[j](m,Z) \equiv \exists_n.\quad F^n(\text{STOP})[j](m,Z) \qquad (j \in S)$$

Proof. Omitted, since it can immediately be obtained from the definition and (1.7).

WEAKEST ENVIRONMENT

We now have got enough to give a formal definition of weakest environment.

In designing a process with master-slave structure to meet an overall specification, we may choose the design of the master part of the process at first, and then inquire what is the minimum specification that must be met by the slave part of the process in order that the whole process meet the overall specification. The required specification is called the *weakest environment* of the designed master.

Let R be an overall specification with free process variables m and Z, and let M be a process with slaves A, and $A - Z = \{s\}$. Let M *we* R denote a process predicate with free process variables s and $Z - A$, which represents the weakest

environment of M with respect to R; i.e. for any process S with slaves $Z - A$

$$(M \underset{A\ Z-A}{\|} s{:}S)\ sat\ R \quad \text{iff} \quad (s{:}S)\ sat\ (P\ we\ R) \quad \text{(Figure 1).}$$

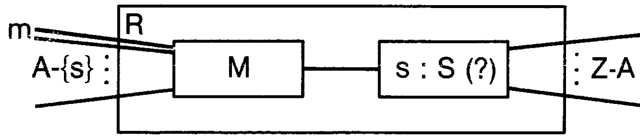


Figure 1

The formal definition can be introduced from the theorems of Section 2 as follows.

By the theorem in Section 2, since $(M \underset{A\ Z-A}{\|} s{:}S)$ has the slave list Z,

$$(M \underset{A\ Z-A}{\|} s{:}S)\ sat\ R(m, Z)$$

$$\text{iff} \quad \forall m, Z.(M \underset{A\ Z-A}{\|} s{:}S)(m, Z) \Rightarrow R(m, Z).$$

But

$$(M \underset{A\ Z-A}{\|} s{:}S)(m, Z) \equiv \exists s.\ M(m, A) \,\&\, S(s, Z - A)$$

by Theorem 2.6. So

$$\begin{aligned}
&(M \underset{A\ Z-A}{\|} s{:}S)\ sat\ R(m, Z) \\
&\equiv \forall m, Z.\,(\exists s.\ M(m, A) \,\&\, S(s, Z - A)) \Rightarrow R(m, Z) \\
&\equiv \forall m, s, Z.\,(M(m, A) \,\&\, S(s, Z - A) \Rightarrow R(m, Z)) \\
&\qquad\qquad \text{(since } s \text{ does not occur in } R\text{)} \\
&\equiv \forall m, s, Z.\ S(s, Z - A) \Rightarrow (M(m, A) \Rightarrow R(m, Z)) \\
&\quad ((B \& C \Rightarrow D) \equiv (C \Rightarrow (B \Rightarrow D))) \\
&\equiv \forall s, Z - A.\ S(s, Z - A) \Rightarrow \forall m, A \cap Z. \\
&\quad M(m, A) \Rightarrow R(m, Z) \\
&\qquad\qquad \text{(since } A \cap Z \text{ does not occur in } S\text{)} \\
&\equiv s{:}S\ sat\ (\forall m, A \cap Z.\ M(m, A) \Rightarrow R(m, Z)) \\
&\qquad\qquad \text{(since Theorem in Section 2 and 2.6).}
\end{aligned}$$

Thus $M\ we\ R$ can be suggested to be

$$\forall m, A \cap Z.\ M(m, A) \Rightarrow R(m, Z).$$

Similarly, if we design the slave part of a process first, then we can inquire about the *weakest environment* of the slave with respect to an overall specification.

Let R be an overall specification with free process variables m and Z, and let S be a process with slaves B, and $A = (Z - B) \cup \{s\}$. Let $(s{:}S\ we\ R)$ denote the weakest environment with free variables m and A such that for any process M with slaves A

$$(M \underset{AB}{\|} s{:}S)\ sat\ R \text{ iff } M\ sat\ (s{:}S\ we\ R) \qquad \text{(Figure 2).}$$

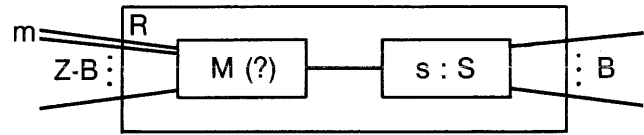


Figure 2

Similar reasoning applies; we can therefore suggest

$$(s{:}S\ we\ R) \text{ as } \forall B \cap Z.\ S(s, B) \Rightarrow R(m, Z).$$

In general for any process P with process names X and any assertion R with process variables Y, we define the weakest environment of P with respect to R as

$$P\ we\ R =_{df} \forall X \cap Y.\ P(X) \Rightarrow R(Y).$$

So $P\ we\ R$ is a predicate of process variables $X \oplus Y$, where $X \oplus Y = (X \cup Y) - (X \cap Y)$.

If M is a process with process names m and A, and R is an assertion with process variables m and Z, then

$$\begin{aligned}
&M\ we\ R \\
&\equiv \forall (\{m\} \cup A) \cap (\{m\} \cup Z).\quad M(m, A) \Rightarrow R(m, Z) \\
&\equiv \forall m, A \cap Z.\quad M(m, A) \Rightarrow R(m, Z)
\end{aligned}$$

as suggested.

If $(s{:}S)$ is a process, where S has process names m and B, and $s \bar{\in} B$; R is an assertion with process names m and Z; and $A = (Z - B) \cup \{s\}$, then $(s{:}S)$ has process names $\{s\} \cup B$, and

$$\begin{aligned}
&(s{:}S)\ we\ R \\
&\equiv \forall (\{s\} \cup B) \cap (\{m\} \cup Z).\quad (s{:}S)(s, B) \Rightarrow R(m, Z) \\
&\equiv \forall B \cap Z.\quad S(s, B) \Rightarrow R(m, Z) \qquad (2.5)
\end{aligned}$$

as suggested also.

Hence the required theorem follows:

Theorem 1.

$$(M \underset{AB}{\|} s{:}S)\ sat\ R(m, Z)$$

iff (1) $M\ sat\ (S{:}s\ we\ R)$
or (2) $(s{:}S)\ sat\ (M\ we\ R)$.

where $Z = (A - \{s\}) \cup B$.

Proof. As given above.

The intention to develop a calculus of the partial correctness of communicating processes in terms of weakest environment is based on the fact that a process satisfies an assertion iff the weakest environment of the process with respect to the assertion is a tautology.

Theorem 2.

If P is a process with process names X and R is an assertion of process variables X, then

$$P\ sat\ R \quad \text{iff } (P\ we\ R) \equiv \text{true}$$

Proof.

$$\begin{aligned}
P\ we\ R &\equiv \forall X.\quad P(X) \Rightarrow R(X) && \text{(def.)} \\
&\equiv P\ sat\ R && \text{(Theorem of 2)}
\end{aligned}$$

PROOF RULES

We now develop a set of structural rules for proving the theorems of the form of $P\ we\ R$. The validity of the proof rules is established by proving that each of them is consistent with the semantics.

In what follows we use X to represent the process names mentioned by process P, and Y to represent the process variables of assertion R.

4.0 General Rules (Healthiness Conditions)

(a) $(P\ we\ \text{true}) \equiv \text{true}$
(b) $P\ we\ \text{false} \equiv \text{false}$, provided P and false have same process names; i.e. $X = Y$
(c) If $R \Rightarrow T$ is a theorem, then $(P\ we\ R) \Rightarrow (P\ we\ T)$
(d) $(\forall y.\ \ P\ we\ R(y)) \equiv P\ we\ \forall y. R(y)$,

provided y is not a process variable and y does not occur in P freely.
The proofs for their consistency:

(a) $P\ we$ true
$\equiv \forall X \cap Y.\ P(X) \Rightarrow \text{true}$ (def.)
$\equiv$ true.
(b) If P and false have same process names X, then
$P\ we\ \text{false} \equiv \forall X.\ P(X) \Rightarrow \text{false}$ (def.)
$\equiv \forall X. \neg P(X)$

Since for any process P, the empty sequence is always one of its traces, i.e., $P(<>) \equiv \text{true}$, then

$P\ we\ \text{false} \equiv \forall X. \neg P(X)$
$\equiv$ false.

Actually, even if they mention different process names,

$(P\ we\ \text{false})(<>)$
$\equiv (\forall X \cap Y. \neg P(X))[<>/X - Y]$
$\equiv$ false.

Hence, in any case no process can satisfy ($P\ we$ false). In this sense ($P\ we$ false) is always equivalent to false.

(c) $P\ we\ R \equiv \forall X \cap Y.\ \ P(X) \Rightarrow R(Y)$ (def.)
$\Rightarrow \forall X \cap Y.\ \ P(X) \Rightarrow T(Y)$ (since $R \Rightarrow T$)
$\equiv P\ we\ T$ (def.)
(d) $\forall y.\ P\ we\ R(y) \equiv \forall y.\ \forall X \cap Y.\ \ P(X) \Rightarrow R(y, Y)$
$= \forall X \cap Y.\ \ P(X) \Rightarrow \forall y. R(y, Y)$
(y is not free in P)
$= P\ we\ \forall y.\ \ R(y)$ (def.)

4.1 STOP

$$\text{STOP}\ we\ R \equiv \forall X \cap Y.\ X = <> \Rightarrow R(Y).$$

The proof for the consistency:

$\text{STOP}\ we\ R \equiv \forall X \cap Y.\ \text{STOP}(X) \Rightarrow R(Y)$ (def.)
$\equiv \forall X \cap Y.\ X = <> \Rightarrow R(Y)$ (2.1)

4.2 Output

(a) If $z \in X - Y$,
$(z\,!e \rightarrow P)\ we\ R$
$\equiv \text{STOP}\ we\ R\ \&\ hd(z) = e \Rightarrow (P\ we\ R)[tl(z)/z]$,

where both STOP and P have the same process names X of $(z\,!e \rightarrow P)$.
The proof for its consistency:

$LHS \equiv \forall X \cap Y.\ (z\,!e \rightarrow P)(X) \Rightarrow R(Y)$ (def.)
$\equiv \forall X \cap Y.\ (\text{STOP}(X)\ v\ (hd(z)$
$= e\ \&\ P(X)[tl(z)/z])) \Rightarrow R(Y)$ (2.2)
$\equiv \forall X \cap Y.\ \text{STOP}(X) \Rightarrow R(Y)\ \&\ (hd(z)$
$= e\ \&\ P(X)[tl(z)/z]) \Rightarrow R(Y)$
$\equiv (\forall X \cap Y.\ \text{STOP}(X) \Rightarrow R(Y))\ \&\ (hd(z)$
$= e \Rightarrow (\forall X \cap Y.\ P(X)$
$\Rightarrow R(Y))[tl(z)/z])$
$(z \notin Y \cap X)$
$\equiv RHS$ (def.)

(b) If $z \in X \cap Y$, then

$$(z\,!e \rightarrow P)\ we\ R \equiv (\text{STOP}\ we\ R)\ \&\ \text{P}\ we\ R[e\hat{\ }z/z],$$

where both STOP and P take X as their process names.
The proof for the consistency:

$LHS \equiv \forall X \cap Y.\ (z\,!e \rightarrow P)(X) \Rightarrow R(Y)$ (def.)
$\equiv \forall X \cap Y.\ (\text{STOP}(X)\ v\ (hd(z)$
$= e\ \&\ P(X)[tl(z)/z])) \Rightarrow R(Y)$ (2.2)
$\equiv (\forall X \cap Y.\ \text{STOP}(X) \Rightarrow R(Y))\ \&\ \forall X \cap Y.$
$P(X)[tl(z)/z] \Rightarrow R(Y)[e\hat{\ }tl(z)/z]$
(when $hd(z) = e,\ z = hd(z)\hat{\ }tl(z) = e\hat{\ }tl(z)$)
$\equiv (\forall X \cap Y.\ \text{STOP}(X) \Rightarrow R(Y))\ \&\ \forall X \cap Y.$
$P(X) = R(Y)[e\hat{\ }z/z]$
(by renaming bound variable)

$\equiv RHS$ (def.)

4.3 Input

(a) If $z \in X - Y$,
$(z?x{:}M \rightarrow P(x)\ we\ R$
$\equiv \text{STOP}\ we\ R\ \&\ hd(z) \in M \Rightarrow (P(x)\ we\ R)[hd(z)/x]$
$[tl(z)/z]$,

where both STOP and $P(x)$ still take X as their process names.

(b) If $z \in X \cap Y$, then
$(z?x{:}M \rightarrow P(x))\ we\ R$
$= \text{STOP}\ we\ R\ \&\ \forall x \in M.\ P(x)\ we\ R[x\hat{\ }z/z]$,

where both STOP and $P(x)$ still have process names X.

The proofs for their consistency are omitted, since these are similar to the proofs of (4.2).

4.4 Alternation

$$(P \mid Q) \textit{ we } R = P \textit{ we } R \,\&\, Q \textit{ we } R,$$

where $P \mid Q$, P, and Q mention the same process names.

The proof for the consistency:

$$\begin{aligned} LHS &\equiv \forall X \cap Y.\ (P \mid Q)(X) \Rightarrow R(Y) && \text{(def.)} \\ &\equiv \forall X \cap Y.\ (P(X) \vee Q(X)) \Rightarrow R(Y) && (2.4) \\ &\equiv \forall X \cap Y.\ (P(X) \Rightarrow R(Y)) \,\&\, (Q(X) \Rightarrow R(Y)) && \\ &\equiv RHS && \text{(def.)} \end{aligned}$$

4.5 Naming

If $s \in X - Y$ and $m \bar{\in} X$, then

$$\begin{aligned} &(s{:}P) \textit{ we } R \\ &\equiv (P \textit{ we } R[z/m])[s/m][m/z] \end{aligned}$$

where $z \bar{\in} X \cup Y$ and P has the set of process names $X[m/s]$.

The proof of the consistency:

$$\begin{aligned} LHS &\equiv \forall X \cap Y.\ (s{:}P)(X) \Rightarrow R(Y) && \text{(def.)} \\ &\equiv \forall X \cap Y.\ P(X[m/s])[s/m] \Rightarrow R(Y) && (2.5) \\ &\equiv \forall X[m/s] \cap Y[z/m].\ P(X[m/s])[s/m] \Rightarrow R(Y) && \\ & \qquad \text{(since } X \cap Y = X[m/s] \cap Y[z/m]) && \\ &\equiv \forall X[m/s] \cap Y[z/m]. && \\ & \quad P(X[m/s])[s/m] \Rightarrow R(Y[z/m])[m/z] && \\ &\equiv (\forall X[m/s] \cap Y[z/m]). && \\ & \quad P(X[m/s]) \Rightarrow R(Y[z/m]))[s/m][m/z] && \\ & \qquad \text{(since } s \bar{\in} Y,\ m \bar{\in} X \text{ and } z \bar{\in} X \cup Y.) && \\ &\equiv RHS && \text{(def.)} \end{aligned}$$

4.6 Parallelism

$$\begin{aligned} &(P \parallel_{AB} s{:}Q) \textit{ we } R \\ &\quad \equiv P \textit{ we } (s{:}Q \textit{ we } R) \\ &\quad \equiv s{:}Q \textit{ we } (P \textit{ we } R), \end{aligned}$$

where P mentions process names $\{m\} \cup A$, and $s{:}Q$ mentions $\{s\} \cup B$, and $A \cap B = 0$ and $s \bar{\in} Y$.

Prove its consistency. Let $X = (A - \{s\}) \cup B \cup \{m\}$, which is the set of process names of $P \parallel_{AB} s{:}Q)$. Then

$$\begin{aligned} LHS &\equiv \forall X \cap Y.\ (P \parallel_{AB} s{:}Q)(X) \Rightarrow R(Y) && \text{(def.)} \\ &\equiv \forall X \cap Y.\ (\exists s. P(m, A) \,\&\, Q(s, B)) \Rightarrow R(Y) && (2.6) \\ &\equiv \forall X \cap Y, s.\ P(m, A) \,\&\, Q(s, B) \Rightarrow R(Y) && (s \quad Y) \\ &\equiv \forall X \cap Y, s.\ P(m, A) \Rightarrow (Q(s, B) \Rightarrow R(Y)) && \\ &\equiv \forall(\{m\} \cup A)((\{s\} \cup B) \oplus Y).\ P(m, A) \Rightarrow && \\ & \qquad \forall(\{s\} \cup B) \cap Y.\ Q(s, B) \Rightarrow R(Y) && \\ & \qquad \text{(since } (\{m\} \cup A) \cap (\{s\} \cup B) \cap Y = \emptyset) && \\ &\equiv P \textit{ we } (s{:}Q \textit{ we } R) && \end{aligned}$$

We can prove the other equivalence similarly.

Note that this rule shows that the weakest environment satisfies the usual axiom of composition as well as the weakest precondition.

4.7 Recursion

$$\mu p.F \textit{ we } R \equiv \forall n.\ F^n(\text{STOP}) \textit{ we } R$$

and $(\mu p[i{:}S].F)[j]$ *we* $R[j] \equiv \forall n.$ $F^n(\text{STOP})[j]$ *we* $R[j](j \in S)$, where $\mu p.F$ *and* $F^n(\text{STOP})$ have same process names.

The proof of the consistency is only given for single recursion:

$$\begin{aligned} LHS &\equiv \forall X \cap Z.\ (\mu p.F)(X) \Rightarrow R(Z) && \text{(def.)} \\ &\equiv \forall X \cap Z.\ (\exists n.\ F^n(\text{STOP})(X) \Rightarrow R(Z) && (2.7) \\ &\equiv \forall X \cap Z, n.\ F^n(\text{STOP})(X) \Rightarrow R(Z) && \\ & \qquad \text{(since } n \text{ does not occur in } R) && \\ &\equiv \forall n.\ F^n(\text{STOP}) \textit{ we } R && \text{(def.)} \end{aligned}$$

From (4.7) a useful structural induction rule follows:

4.7.1 If

$$T \Rightarrow (\text{STOP } \textit{we } R)$$

and for any process P

$$(T \Rightarrow (P \textit{ we } R)) \Rightarrow (T \Rightarrow (F(P) \textit{ we } R)),$$

then

$$T \Rightarrow (\mu p.F \textit{ we } R),$$

where STOP, P, and $F(P)$ are supposed to have the same process names as $\mu p.\ F$.

The proof of the consistency for this rule can be obtained as follows:

Start at $T \Rightarrow (\text{STOP } \textit{we } R)$, and repeat to use $(T \Rightarrow (P \textit{ we } R)) \Rightarrow (T \Rightarrow (F(P) \textit{ we } R))$. Then we can get for any n $T \Rightarrow (F^n(\text{STOP}) \textit{ we } R)$. Thus $T \Rightarrow \forall n.\ F^n(\text{STOP}) \textit{ we } R$. Hence $T \Rightarrow (\mu p.F \textit{ we } R)$ by (4.7).

EXAMPLE

We end this paper by showing a proof of the partial correctness of the matrix multiplier (see Section 1). We want to prove

$$\text{multiplier } \textit{sat} \quad i \quad 1..\#m \ .\ m_i = {}_{j=1}^{3} v j * \text{row} j_i$$

$$\mathop{\&}_{j=1}^{3} \# m \ \# \text{ row} j$$

By Theorem 2 of Section 4 this proposition is equivalent to

$$(\text{multiplier } we\ \forall i \in \{1..\#m\}.\ m_i = \sum_{j=1}^{3} v[j] * \text{row}[j]_i \ \mathop{\&}_{j=1}^{3} \#m \leq \#\text{row}[j]) \equiv \text{true},$$

where multiplier is a process with process names m and row $[j{:}1..3]$.
Since

$$\text{multiplier} \triangleq (((\text{adder} \mathop{\|}_{A_1 B_1} \text{mult}[1]{:}p[1]) \mathop{\|}_{A_2 B_2} \text{mult}[2]{:}p[2]) \mathop{\|}_{A_3 B_3} \text{mult}[3]{:}p[3]),$$

$$\begin{aligned}
&\text{multiplier } we\ \forall i \in \{1..\#m\}.\quad m_i = \sum_{j=1}^{3} v[j] * \text{row}[j]_i \ \mathop{\&}_{j=1}^{3} \#m \leq \#\text{row}[j] \\
&\equiv \text{adder } we\ (\text{mult}[1]{:}p[1] \\
&\qquad we\ (\text{mult}[2]{:}p[2] \\
&\qquad we\ (\text{mult}[3]{:}p[3] \\
&\qquad we\ \forall i \in \{1..\#m\}.\quad m_i = \sum_{j=1}^{3} v[j] * \text{row}[j]_i \ \mathop{\&}_{j=1}^{3} \#m \leq \#\text{row}[j])))
\end{aligned}$$

by (Parallelism).
We now need four lemmas.

Lemma 1.

$$\begin{aligned}
&\forall i \in \{1..\#m\}.\quad m_i = (\sum_{j=1}^{3} v[j] * \text{row}[j]_i + \text{mult}[3]_i \\
&\mathop{\&}_{j=1}^{2} \#m \leq \#\text{row}[j] \ \& \ \#m \leq \#\text{mult}[3] \\
&\Rightarrow (\text{mult}[3]{:}p[3]\ we\ \forall i \in \{1..\#m\}.\ m_i = \sum_{j=1}^{3} v[j] * \text{row}[j]_i \\
&\mathop{\&}_{j=1}^{3} \#m \leq \#\text{row}[j])
\end{aligned}$$

Lemma 2.

$$\begin{aligned}
&\forall i \in \{1..\#m\}.\ m_i = (v[1] * \text{row}[1]_i + \text{mult}[2]_i + \\
&\quad \text{mult}[3]_i) \ \& \ \#m \leq \#\text{row}[1] \ \mathop{\&}_{j=2}^{3} \#m \leq \#\text{mult}[j] \\
&\Rightarrow (\text{mult}[2]{:}p[2]\ we\ \forall i \in \{1..\#m\}.\ m_i = \Big(\sum_{j=1}^{2} v[j] * \\
&\quad \text{row}[j]_i = \text{mult}[3]_i) \mathop{\&}_{j=1}^{2} \#m \leq \#\text{row}[j] \\
&\quad \& \ \#m \leq \#\text{mult}[3])
\end{aligned}$$

Lemma 3.

$$\begin{aligned}
&\forall i \in \{1..\#m\}.\ m_i = \sum_{j=1}^{3} \text{mult}[j]_i \mathop{\&}_{j=1}^{3} \#m \leq \#\text{mult}[j] \\
&\Rightarrow (\text{mult}[1]{:}p[1]\ we\ \forall i \in \{1..\#m\}.\ m_i = v[1] * \text{row}[1]_i + \\
&\quad \text{mult}[2]_i + \text{mult}[3]_i \ \& \ \#m \leq \#\text{row}[1] \mathop{\&}_{j=2}^{3} \#m \leq \#\text{mult}[j])
\end{aligned}$$

Lemma 4.

$$(\text{adder } we\ \forall i \in \{1..\#m\}.\quad m_i = \sum_{j=1}^{3} \text{mult}[j]_i \ \mathop{\&}_{j=1}^{3} \#m \leq \#\text{mult}[j]) \equiv \text{true}$$

Let us only present the proof of Lemma 4, and let R stand for the assertion on the righthand side of the previous weakest environment.
Proof.
Adder is an abbreviation for the least fixed point of the recursion

$$\begin{aligned}
&\text{adder} \triangleq \text{mult 1 } ?x{:}\text{NAT} \rightarrow \text{mult 2 } ?y{:}\text{NAT} \rightarrow \\
&\quad \text{mult } 3\ ?\ z{:}\text{NAT} \rightarrow m!(x+y+z) \rightarrow \text{adder},
\end{aligned}$$

and the process names which occur in adder are $X = \{\text{mult}[1], \text{mult}[2], \text{mult}[3], m\}$.
Let us use (4.7.1) to prove this lemma.
For (STOP $we\ R \equiv$ true it is trivial, since

$$\begin{aligned}
\text{STOP } we\ R &\equiv \forall X.\ \text{m} = \langle\ \rangle \mathop{\&}_{j=1}^{3} \text{mult[j]} = \langle\ \rangle R \qquad (4.1) \\
&\equiv \text{true} \qquad\qquad (\#\langle\ \rangle = 0).
\end{aligned}$$

Now assume $(P\ we\ R) \equiv$ true. Then

$$\begin{aligned}
&(\text{mult}[1]?x{:}\text{NAT} \rightarrow \text{mult}[2]?y{:}\text{NAT} \rightarrow \text{mult}[3]?z{:}\text{NAT} \rightarrow \\
&m!(x+y+z) \rightarrow P) we\ R \\
&\equiv \text{STOP } we\ R\ \&\ \forall x \in \text{NAT}. \\
&(\text{mult}[2]?y{:}\text{NAT} \rightarrow \text{mult}[3]?z{:}\text{NAT} \rightarrow \\
&\qquad m!(x+y+z) \rightarrow P) we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]] \\
&\qquad\qquad (4.3\,(b)) \\
&\equiv \forall x \in \text{NAT}.\ \text{STOP } we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]] \\
&\qquad \&\ \forall y \in \text{NAT}.\ \text{mult}[3]?z{:}\text{NAT} \rightarrow m!(x+y+z) \rightarrow P \\
&\qquad\quad we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]][y\hat{\ }\text{mult}[2]/\text{mult}[2]] \\
&\qquad\qquad (\text{STOP } we\ R \equiv \text{true and } 4.3\,(b)) \\
&\equiv \forall x, y \in \text{NAT}.\ \text{STOP } we\ R \\
&\qquad [x\hat{\ }\text{mult}[1]/\text{mult}[1]][y\hat{\ }\text{mult}[2]/\text{mult}[2]] \\
&\qquad \&\ \forall z \in \text{NAT}.\ m!(x+y+z) \rightarrow P \\
&\qquad\quad we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]][y\hat{\ }\text{mult}[2]/\text{mult}[2]] \\
&\qquad\quad [z\hat{\ }\text{mult}[3]/\text{mult}[3]] \\
&\qquad\qquad (\text{STOP } we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]] \equiv \text{true and } 4.3\,(b)) \\
&= \forall x, y, \in \text{NAT}.\quad \text{STOP } we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]] \\
&\qquad [y\hat{\ }\text{mult}[2]/\text{mult}[2]][z\hat{\ }\text{mult}[3]/\text{mult}[3]] \\
&\qquad \&\ P\ we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]][y\hat{\ }\text{mult}[2]/\text{mult}[2]] \\
&\qquad [z\hat{\ }\text{mult}[3]/\text{mult}[3]][(x+y+z)\hat{\ }m/m] \\
&\qquad (\text{STOP } we\ R[x\hat{\ }\text{mult}[1]/\text{mult}[1]][y\hat{\ }\text{mult}[2]/\text{mult}[2]] \\
&\equiv \text{true and } 4.2\,(b)).
\end{aligned}$$

But

$R[x\hat{\ }\text{mult}[1]/\text{mult}[1]][y\hat{\ }\text{mult}[2]/\text{mult}[2]][z\hat{\ }\text{mult}[3]/\text{mult}[3]][(x+y+z)\hat{\ }m/m \equiv R$, and STOP $we\ R\ [x\hat{\ }\text{mult}[1]/\text{mult}[1]][y\hat{\ }\text{mult}[2]/\text{mult}[2]][z\hat{\ }\text{mult}[3]/\text{mult}[3]] \equiv$ true by (4.1), hence the previous proposition is equivalent to truth as required.

ACKNOWLEDGMENTS

I am deeply grateful to Tony Hoare. The original idea of weakest environment appears in a joint work with him on the partial correctness of communication protocols,[3] and the suggestion for writing this paper also came from him.

I also thank the Institute of Computing Technology, Chinese Academy of Sciences, for awarding me two years' stay in the United Kingdom, which made possible the cooperative works I published with British scholars.

REFERENCES

1. Dijkstra, E. W. "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs." *Communications of the ACM,* 18 (1975), p. 8.
2. Zhou, Chaochen, and C. A. R. Hoare. "Partial Correctness of Communicating Sequential Processes." In *Proceedings of the Second International Conference on Distributed Computing Systems,* April 1981.
3. Zhou, Chaochen, and C. A. R. Hoare. "Partial Correctness of Communication Protocols." Paper presented at INWG/NPL Workshop, "Protocol Testing—Towards Proving?" May 1981.

Adaptive structuring of distributed databases

by K. DAN LEVIN
Tel Aviv University
Tel Aviv, Israel*
and
The Wharton School
Philadelphia, Pennsylvania

ABSTRACT

This paper reports the study of an adaptive strategy for allocating programs and data files in computer networks when only imperfect information on access request rates is available. The adaptive system is based on a set of computer procedures to collect access statistics, to estimate frequencies, to find optimal file assignment based on expected improvements in performance, and to redistribute the files over the network nodes. These computer modules are synchronized with each other in order to predict the best assignment policy on the basis of where access requests are expected to arise in the future; and as new information becomes available, the forecast is revised and the assignment policy adapts itself to the new information. Transition costs incurred by file transmission from one node to another are taken into account, and the decision to reassign files is triggered only when net gains can be realized.

The paper also points out the departure from optimality due to the complexity of a multistage self-adapting system. Accordingly, a second-best solution is suggested.

*This paper was written while the author was affiliated with the Department of Decision Sciences, The Wharton School, University of Pennsylvania.

I. INTRODUCTION

The operating costs and the response time of a database shared by a community of users depend on its physical structure and usage patterns. Since usage patterns vary over time, a static physical structure may result in performance degradation. Under these circumstances performance efficiency can be restored with the ability to choose a good database structure and to adapt this structure to changing requirements. Many researchers have already recognized the importance of an adaptive capability, and some solutions applicable to centralized databases have been offered.[3–6, 10, 14, 15, 19, 20] With growing interest in the performance of distributed databases it is evident that such capability is especially important in that environment.[7, 12]

Several models have been suggested for determining the optimal physical locations of files in distributed databases, both in a static environment[1, 2, 9, 12] and in a dynamic environment where usage patterns change over time.[8] The major deficiency is that perfect information on usage patterns was assumed in all these models. This is seldom the case in practical applications, where access request rates are not perfectly known in advance. In this case it is necessary to estimate these request patterns and to incorporate these estimates in the distributed database design. The purpose of this paper is to detail the various functions of an adaptive distributed-database management system. Hammer's guidelines[5] for self-adaptive database design are adopted. Accordingly, the system includes an information-gathering module to collect statistics, an estimator module to estimate future access patterns, a performance measure to evaluate different physical structures, and an optimization module to determine the optimal physical locations of files on the basis of expected performance improvements. Clearly, there are some costs associated with the physical redistribution of the database; therefore, the problem is to establish the optimal tradeoff between efficient system operation and reorganization costs.

In Section A a performance measure for evaluating alternative physical database structures is described briefly. The performance measure is formulated as a cost function to be minimized over the set of feasible distributed structures. This cost function, together with an efficient search procedure, defines the optimization model in our adaptive system. This model has been developed in our previous research,[7, 8, 9] both for static and time-varying access requirements. The parameters of the model—namely, the access request rates from each user to each program and file—are to be estimated in the estimator module, on the basis of data collected in the information-gathering module.

In Section B the estimator module is described and its statistical properties are investigated. The input requirements of this module are defined in order to determine the structure of the information-gathering module. Implementation considerations are discussed at the end of this section.

Section C concludes the paper by pointing out the departure from optimality when one takes into account the additional information on actual access requests that can be collected as the system evolves. This possibility transforms the optimization problem to one of finding an optimal stochastic control—a problem which, in general, has infinite dimensions. Realizing this complexity, some guidelines for second-best solutions are suggested at the end of the paper.

A. The Performance Measure and the Optimization Module

The distributed database considered here is shared by a community of users interconnected through a computer communication network. The network has N nodes and a database consisting of F files and P programs. Every node in the network demands the services of some programs and files. This demand is generated through transactions originated in each node and falls into one of two classes, query traffic and update traffic. A transaction is first routed to its relevant program; from this program a query is transmitted to the nearest file copy while an update message is transmitted to every copy of the file. The rate of transactions originated at each node varies over time. It has been shown[9] that the multiple-file problem can be decomposed into individual-file assignment problems, so we can proceed with the following notations:

λ_{ipt} = query traffic from node i via program p at period t
λ'_{ipt} = update traffic from node i via program p at period t
C_{ij} = communication cost per query unit from i to j
C'_{ij} = communication cost per update unit from i to j
σ_k=storage cost at node k
α = expansion factor for query message
β = expansion factor for update message

The expansion factors are included in the model because the length of the messages traveling between the programs and the data files differs from the length of the messages between the users' terminals and the processing programs.

Throughout the paper subscript i indicates a user node, subscript j indicates a node where a program resides, and subscript k indicates a node where a file resides. In addition, p denotes the program index, $p = 1, \ldots, P$; and J, Q, and U are sets of programs. These letters subscripted, e.g., J_p are sets of permissible nodes for program p (i.e., places where p can execute).

Let K_t be the set of nodes where a copy of the file is stored

in period t. Let N be the set of nodes in the network, so that $K_t \leq N$ for every t.

It has been shown[7, 8] that the costs associated with an arbitrary assignment for any given period, t, can be formulated as

$$C(K_t) = q(K_t) + u(K_t) + s(K_t)$$

where

$q(K_t) = \Sigma_{p \in Q} \Sigma_i \lambda_{ipt} \min_{j \in Qp} (C_{ij} + \alpha \min_{k \in K_t} C_{jk})$
= communication costs of queries

$u(K_t) = \Sigma_{p \in U} \Sigma_i \lambda'_{ipt} \min_{j \in Up} (C'_{ij} + \beta \Sigma_{k \in K_t} C'_{jk})$
= communication cost of updates

$s(K_t) = \Sigma_{k \in K_t} \sigma_t$

Q = the set of query programs
U = the set of update programs
Q_p = set of nodes at which query program p can be processed
U_p = set of nodes at which update program p can be processed

With varying access request rates from period to period, it is conceivable that an optimal assignment at one period is nonoptimal in the next period. In this case it is not sufficient to optimize the cost function above, since reassignment costs have to be taken into account. These costs express the cost of transmitting the files (over the communication link) from an assignment in period $t-1$ to its assignment in period t.

Let b denote the number of messages to be transmitted when a file is assigned to a different location.

Let $d(K_{t-1}, K_t)$ denote the cost of reassigning the file from an initial assignment K_{t-1} to an arbitrary assignment K_t. Then:

$$d(K_{t-1}, K_t) = \Sigma_{j \epsilon k t} b \min_{i \epsilon k_{t-1}} C_{ij}$$

The term $\min_{i \epsilon K_{t-1}} C_{ij}$ reflects the assumption that file transmission is carried out in the most economical way.

Thus, the resulting cost function to be minimized is:

$$G(K_t) = q(K_t) + u(K_t) + s(K_t) + d(K_{t-1}, K_t)$$

It was shown by Levin and Morgan[8] that the same efficient branch and bound procedure that was suggested for the static model[9] can be applied here. Furthermore, the inclusion of the reassignment cost tightens the bounds generated in the static model, and the search procedure for the problem above is *at least* as efficient as the one for the static model.

When the estimator module generates estimates for the access requests in the next period, these estimates are used in evaluating the above function in the optimization module. If the resulting optimal assignment is different from the current assignment, the files have to be reassigned, since the reassignment cost has already been taken into account.

B. The Estimator Module

The optimization module described in Section A requires the input of the access request rates from users to files. When these request rates are known with certainty, a cost minimization is performed to determine the optimal assignment of files in the next period, taking into account the reorganization cost. In most practical cases, the exact number of users' requests in the next period is unknown and can be treated as random variables. Since λ_{ipt} and λ'_{ipt} are random variables, the associated cost $G(K_t)$ is in itself a random variable. Under these circumstances the cost minimization objective should be modified. The most common approach in this case is to minimize the expected cost, i.e.,

$$\text{Find } K_t^* \text{ such that } E[G(K_t^*)] = \text{Min } \{E[G(K_t)]\}.$$

A straightforward application of basic statistics results in the problem of estimating the expected request rates for each node, program, and file. These request rates can be estimated (following Hammer's approach[5]) by applying an exponential smoothing procedure that would be sensitive to changes in trends.

Let

- $\lambda_{ip}(t)$ denote the access request rates realized in period t (the same procedure can be used for updates $\lambda'_{ip}(t)$).
- $\lambda_{i\hat{p}}(t)$ denote the corresponding estimated access request rates for period t.

Let $e(t)$ be an estimated trend factor to adjust the prediction of access request rates for period $(t+1)$.

And let γ and δ denote the smoothing coefficients for exponential forecasting.

Then the adaptive estimator module is defined by the following procedure:

At the initialization step $t=0$ observe $\lambda_{ip}(0)$.

Let

$$\lambda_{i\hat{p}}(0) = \lambda_{i\hat{p}}(0)$$
$$e(0) = 0$$

At $t=1$ set

$$\lambda_{i\hat{p}}(1) = \gamma[\lambda_{ip}(1)] + (1-\gamma)[\lambda_{i\hat{p}}(0)]$$
$$e(1) = \delta[\lambda_{i\hat{p}}(1)] - (1-\delta)[\lambda_{i\hat{p}}(0)]$$

At $t=2$ set

$$\lambda_{i\hat{p}}(2) = \gamma[\lambda_{i\hat{p}}(2)] + (1-\gamma)[\lambda_{i\hat{p}}(1) + e(1)]$$
$$e(2) = \delta[\lambda_{i\hat{p}}(2) - \lambda_{i\hat{p}}(1)] + (1-\delta)[e(1)]$$

And for an arbitrary period t:

Set

$$\lambda_{i\hat{p}}(t) = \gamma[\lambda_{ip}(t)] + (1-\gamma)[\lambda_{i\hat{p}}(t-1) + e(t-1)]$$
$$e(t) = \delta[\lambda_{i\hat{p}}(t) - \lambda_{i\hat{p}}(t-1)] + (1-\delta)[e(t-1)]$$

For this procedure the estimated access request rates for period $t+1$ are

$$\lambda_{i\hat{p}}(t+1) = \lambda_{i\hat{p}}(t) + e(t)$$

with a prediction error defined by $\lambda_{i\hat{p}}(t) - \lambda_{ip}(t)$.

The statistical properties of these estimates can be analyzed by formulating an underlying stochastic process that represents the adaptive nature of the estimator.

Let

$\lambda_{i\hat{p}}(t)^*$ be the true mean of the random variable $\lambda_{ip}(t)$.
$\eta(t)$ be the trend change from period $t-1$ to period t (estimated by $e(t)$.

$$\lambda_{ip}(t) = \lambda_{ip}(t)^* + U_t$$
$$\lambda_{ip}(t)^* = \lambda_{ip}(t-1)^* + \eta(t)$$
$$\eta(t) = \eta(t-1)\,V_t$$

where U_t and V_t are time series with zero mean, constant variances (σ_u^2, σ_v^2), and covariances of all kinds:

$$\begin{aligned} E(U_t) &= E(V_t) = 0 \\ E(U_t, U_t') &= \sigma_u^2 \quad \text{if } t = t' \\ &= 0 \quad \text{if } t = t' \\ E(V_t, V_t') &= \sigma_v^2 \quad \text{if } t = t' \\ &= 0 \quad \text{if } t = t' \\ E(U_t, V_t') &= 0 \quad \text{for } \textit{all} \text{ pairs } (t, t') \end{aligned}$$

For this process, it was shown by Theil and Wage[17] that the mean-square prediction error can be minimized by determining the smoothing coefficients γ and δ as a function of the ratio of variances. Clearly, finding optimal values for γ and δ requires estimates of the variances. However, a sensitivity analysis of the consequences of error in estimating the variances ratio was performed by Nerlove and Wage[11] and showed relatively little sensitivity. Even a 50% error in estimating this ratio produces an increase of less than 1.5% in the mean square error achieved. This last result permits us to start with a rough estimate of the variances ratio, determine the values of γ and δ, and update these coefficients when additional information is available. This information can be collected by recording the current mean-square prediction error, i.e.,

$$P_e(t) = \sum_{i=1}^{t} [\lambda_{i\hat{p}}(t) - \lambda_{ip}(t)]^2$$

Thus, the information items to be recorded and transferred from one period $t-1$ to period t are $\lambda_{i\hat{p}}(t-1)$, $e(t-1)$ and $P_e(t-1)$ for every node, program, and file. Since the data on access request rates can be most easily collected at the originating node, information-gathering modules should reside in each node. Storage requirements for these modules consist of three storage units (for the three information items) for each program and file accessed from the node.

Similarly, the simplicity of the estimation process in terms of computation and storage favors a distributed organization; i.e., information of access request rates from each node should be performed locally. The advantages are twofold: estimation can be performed in parallel (whereas a centralized module would compute the estimates sequentially), and the volume of messages transmitted over the communication link is reduced. Only the revised estimates of access request rates have to be transmitted to the centralized optimization module. This volume can be further reduced by filtering out relatively small changes in the estimates.

C. Departure from Optimality with a Multiperiod Adaptive System

Sections A and B have detailed the components of a self-adaptive distributed database design. Implicitly, we were dealing there with a one-period lookahead system. Changes in access request rates were identified and estimated for the next period, then the physical distribution of files was changed in anticipation of the usage patterns of the next period. However, optimal assignment of files in a multiperiod planning horizon depends also on the access requests beyond the next period. This has already been shown in previous research,[8] where a T-period cost function has been developed for dynamic optimization of distributed databases; i.e., the global T-period cost function for any arbitrary T-period assignment, A, is given by

$$G(A) = \sum_{t=1}^{T} [C(K_t) + d_t(K_{t-1}, K_t)]$$

where file assignment at time 0 is given. The problem is to find that sequence of file assignments A^* that will minimize the global T-period costs,

$$G(A^*) = \min G(A)$$

For this problem assignment, decisions should be made in each period on the basis of the information available at that period. That information typically consists of realized values pertaining to the previous periods and expectations pertaining to future periods. Section B addressed the problem of finding K_t^* such that

$$E[G(K_t^*)] = \text{Min}\{E[G(K_t)]\}$$

However, for a multistage problem it is in general not optimal to adopt the following not uncommon procedure: at each stage j, compute the *decisions* that are optimal for Stages j through $j+n$ on the basis of the information and expectations available at Stage j, enact immediately the subset of decisions that pertains to Stage j, and repeat the procedure at Stage $j+1$. The departure from optimality is of course due to the fact that the decisions enacted at each stage ignore the possibility of reconsideration at the ensuing stages—a possibility of which advantage can be taken. If a two-stage decision problem calls for a decision d_1, followed by an observation x, and then by a second-stage decision d_2, and if d_1^*, d_2^* solve

$$\underset{d_1, d_2}{\text{Max}} \; \underset{x}{E}\, F(d_1, d_2, \tilde{x})$$

then the following well-known inequalities illustrate both the desirability of reconsidering and the need to take that possibility into account when choosing d_1:

$$\underset{d_1}{\text{Max}}\; \underset{x}{E}\; \underset{d_2}{\text{Max}}\; F(d_1, \tilde{d}_2, \tilde{x}) \geq \underset{x}{E}\; \underset{d_2}{\text{Max}}\; F(d_1^*, \tilde{d}_2, \tilde{x}) \geq \underset{x}{E}\; F(d_1^*, d_s^*, \tilde{x})$$

In that case the multiperiod adaptive problem is one of finding the optimal stochastic control which is, in general, of infinite dimensions. The enormous complexity of constructing a self-adaptive, self-learning system[16,18] forced the researchers to proceed with most restricting assumptions. When only *one copy* of each file is allowed to exist in the system at any given time, Segall[13] was able to show (under additional restricting assumptions) that a "separation" property holds in the sense that the estimates of the states of the rates are sufficient statistics for the optimal control.

In most practical situations redundant copies of the files exist in the network, and for these cases no generally applica-

ble algorithm (for optimal control) has, to the best of my knowledge, yet been developed. A plausible second-best solution is to adopt the nonoptimal procedure: at each period t estimate the access request rates for periods $t+1, t+2, \ldots, t+T$; find the optimal assignment of files in the next T periods; and assign the files in accordance with the recommended assignment for Period $t+1$. Repeat that procedure at the end of Period $t+1$ by reestimating the future access rates. The estimator module can generate access rate predictions for lead time $t'=1$ to T by setting

$$\lambda_{i\hat{p}}(t+t') = \lambda_{i\hat{p}}(t) + t'[e(t)]$$

These access rates estimates for the next T periods can then be transmitted to the optimization module that would consist of the multiperiod dynamic optimization model.[8]

SUMMARY

The cost and performance of operating a large shared distributed database are functions of its physical structure and usage patterns. The usage patterns are characterized by the access request rates from each user to each program and file. These access request rates vary over time in accordance with the life cycle of the database; thus, an optimal physical structure at one period is nonoptimal at another period.

In this paper we have studied an adaptive strategy for allocating programs and data files in computer networks when only imperfect information on access request rates is available. The adaptive system is based on a set of computer procedures to collect access statistics, to estimate frequencies, to find optimal file assignment based on expected improvements in performance, and to redistribute the files over the network nodes. These computer modules are synchronized with each other in order to predict the best assignment policy on the basis of where access requests are expected to arise in the future; and, as new information becomes available, the forecast is revised and the assignment policy adapts itself to the new information. Transition costs incurred by file transmission from one node to another are taken into account, and the decision to reassign files is triggered only when net gains can be realized.

We have also pointed out the departure from optimality due to the complexity of a multistage self-adapting system. Accordingly, a second-best solution is suggested.

ACKNOWLEDGMENT

Research for this paper was supported by the U.S. Office of Naval Research, ONR Grant N00014-75-C-0462.

REFERENCES

1. Chu, W. W. "Optimal File Allocation in a Multiple Computer System." *IEEE Trans. on Computers,* 10 (1969), pp. 885–889.
2. Casey, R. G. "Allocation of Copies of a File in an Information Network." *AFIPS, Second Joint Computer Conference* (Vol. 40), 1972, pp. 617–625.
3. Hammer, M., and A. Chan. "Index Selection in a Self-Adaptive Data Base Management System." *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data,* Washington, D.C., June 1976, pp. 1–8.
4. Hammer, M., and A. Chan. "Acquisition and Utilization of Access Patterns in Relational Data Base Implementation." In C. H. Chen (ed.), *Pattern Recognition and Artificial Intelligence.* New York: Academic Press, 1976.
5. Hammer, M. "Self Adaptive Automatic Data Base Design." *AFIPS, Proceedings of the National Computer Conference* (Vol. 46), 1977, pp. 123–129.
6. Kennedy, S. R. "The Use of Access Frequencies in Data Base Organization." Ph.D. dissertation, Department of Operations Research, Cornell University, 1973.
7. Levin, K. D., and H. L. Morgan. "Optimizing Distributed Data Bases: A Framework for Research." *AFIPS, Proceedings of the National Computer Conference* (Vol. 44), 1975, pp. 473–478.
8. Levin, K. D., and H. L. Morgan. "Dynamic Optimization Model for Distributed Data Bases." *Operations Research,* 26, (1978), pp. 824–835.
9. Morgan, H. L., and K. D. Levin. "Optimal Programs and Data Locations in Computer Networks." *Communications of the ACM,* 20 (1977), pp. 315–322.
10. Morgan, H. L., and S. R. Kennedy. "An Adaptive File System." Technical Report No. 4, Information Science Department, California Institute of Technology, 1972.
11. Nerlove, M., and S. Wage. "On the Optimality of Adaptive Forecasting." *Management Science,* 10 (1964), pp. 207–223.
12. Ramamoorthy, C. V., and B. W. Wah. "Data Management in Distributed Data Bases." *AFIPS, Proceedings of the National Computer Conference* (Vol. 48), 1979, pp. 667–680.
13. Segall, A. "Dynamic File Assignment in a Computer Network." *IEEE Transactions on Automatic Control,* 21, (1976) pp. 161–173.
14. Schneiderman, B. "Optimum Data Base Reorganization Points." *Communications of the ACM,* 16 (1973) pp. 362–365.
15. Stocker, P. M., and P. A. Dearnley. "A Self Organising Data Management System." In J. W. Kimbie and K. L. Koffeman (eds.), *Data Base Management.* Amsterdam: North-Holland.
16. Stratonovich, R. L. "Does There Exist a Theory of Synthesis of Optimal, Self-Learning and Self-Adaptive Systems?" *Automat. Telemekh,* 29 (1968), pp. 83–92.
17. Theil, H., and S. Wage. "Some Observations on Adaptive Forecasting." *Management Science,* 10 (1965), pp. 198–206.
18. Tsypkin, Y. Z. "All the Same, Does a Theory of Synthesis of Optimal Adaptive Systems Exist?" *Automat. Telemekh,* 29 (1968), pp. 93–98.
19. Tuel, W. G. "Optimum Reorganization Points for Linearly Growing Files." *ACM Transactions on Data Base Systems,* 3 (1978), pp. 32–40.
20. Yao, S. B., K. S. Das, and T. J. Teorey. "A Dynamic Data Base Reorganization Algorithm." *ACM Transactions on Data Base Systems,* 2 (1976), pp. 159–174.

Distributed scheduling of resources on interconnection networks

by BENJAMIN W. WAH and ANTHONY HICKS
Purdue University
West Lafayette, Indiana

ABSTRACT

In this paper, we have studied the distributed scheduling of resources on interconnection networks. The resource scheduling problem is different from the conventional address mapping problem on interconnection networks because a request is not directed towards a particular destination address but to any one of a pool of destination addresses for free resources. To design an algorithm with the minimum transfer of control signals, priority is associated with the scheduling of multiple requests. This is illustrated by the distributed cross-bar switch which has one signal line in each direction of a switch node. For complete asynchronous operation, more signal lines are needed. This is illustrated by the distributed Omega and binary n-cube networks. Each exchange box in the network operates independently to resolve conflicts. The performance of the distributed scheduling algorithm for the Omega and cube networks is compared against the optimal centralized scheduling algorithm which has about 1% average blocking probability. The performance degradation is less than 20% in all cases. The theory of the design can be applied to other interconnection networks.

*This research was supported by the National Science Foundation Grants ECS 80-16580 and ECS 81-05968.

1. INTRODUCTION

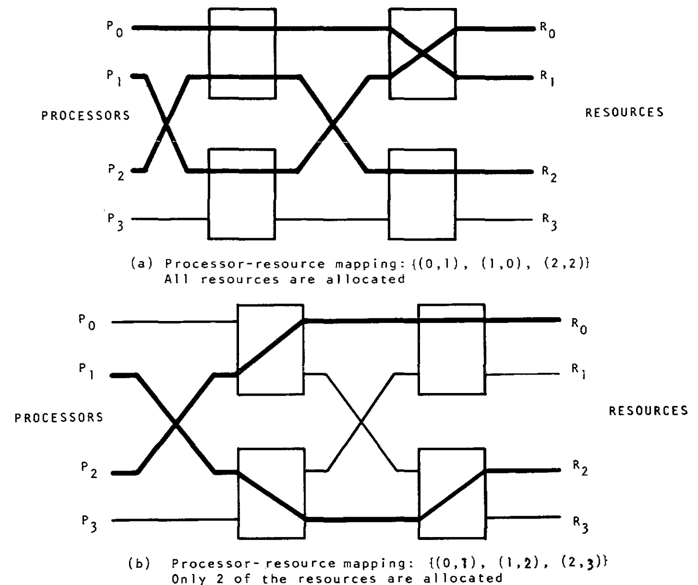
The recent advances in large-scale integrated logic and communication technology, coupled with the explosion in size and complexity of new applications have led to the development of parallel processing systems with a large number of general and special purpose processing units. An interconnection network is an essential element of a parallel processing system which interconnects processors and resources together. The function of the interconnection network is to route requests initiated from one point to another point connected on the network.[5,8,11,14,15,17,21] The notable characteristics in these networks is that routing is done by addresses. That is, a request is initiated with a specific destination or a set of destinations and the requests are supposed to be routed to the correct destinations. Examples of these networks are the Banyan,[7] binary n-cube,[15] cube,[18] perfect shuffle,[20] flip,[3] Omega,[11] data manipulator,[5] augmented data manipulator,[19] delta,[14] and baseline.[21] Examples of systems designed with interconnection networks are TRAC,[17] STARAN,[2] C.mmp,[22] Numerical Aerodynamic Simulation Facility (NASF)[1,4] and the Ballistic Missile Defense testbed.[12]

In general, an interconnection network routes requests from a set of source points to a set of destination points (they may coincide with each other). In a *resource sharing interconnection network (RSIN),* the destination points are identical (or sets of identical) resources to which requests or tasks can be delegated to. Examples of these resources are special purpose VLSI chips. In this respect, jobs initiated at source processors can be sent to any one of the free resources of a given type at the destination. This is the important point that differentiates RSIN from interconnection networks using address mapping. Another application is to map the set of destination points directly into the source points. In this case, we have a load balancing network which can re-balance the load in the system dynamically. RSIN can also be applied in dataflow computers to distribute tasks to processing units.

Since the system operates continuously, requests from source processors can be initiated at random times. At any time, a set of processors may be making requests and a set of resources are free. It is the function of a scheduler to set the RSIN in order to connect the maximum number of resources to the processors, that is, to have the maximum resource utilization.

As an example, consider a 4 by 4 Omega network (see Figure 1). Assume processors 0, 1, 2 are making requests and resources 0, 1, 2 are available. Processor 3 is not making a request and resource 3 is busy. Further, the network is completely free.* All the resources will be allocated if the following processor-resource mappings are used: {(0,0), (1,1), (2,2)}, {(0,1), (1,0), (2,2)}, {(0,2), (1,0), (2,1} or {(0,2), (1,1), (2,0)}. But if the following processor-resource mappings are used: {(0,0), (1,2), (2,1)} or {(0,1), (1,2), (2,0)}, then a maximum of 2 resources can be allocated without blocking. This gives a resource utilization of 66%. A similar example can be generated for the indirect binary n-cube network. This illustrates that the scheduler must be designed properly to give the maximum resource utilization.

*Processor 3 could have made a request earlier and have sent a job to resource 3 in a block transfer mode. The network will, therefore, be free at the present time.



(a) Processor-resource mapping: {(0,1), (1,0), (2,2)}
All resources are allocated

(b) Processor-resource mapping: {(0,1), (1,2), (2,3)}
Only 2 of the resources are allocated

Figure 1—A RSIN using 4 by 4 Omega network

The earliest study of RSIN has been done with respect to centralized computer systems. A uni-bus is used in a time-shared fashion for connecting peripheral I/O devices to the CPU. Multiple time-shared busses have been used in the PLURIBUS minicomputer multiprocessor.[13] A cross-bar switch has been used in C.mmp[22] although the network is mostly used in address mapping mode. This network permits full interconnection capability between any source and destination ports. As long as each source port addresses a unique destination port, there is no blocking in the network and all messages can be routed through the network simultaneously. The single or multiple busses is a source of bottleneck, and is the least expensive design. The cross-bar switch is the most expensive network but has the least degree of blocking. A compromise is to use a less expensive network than the cross-bar switch that has a lower blocking probability than the single bus systems. This has been studied with respect to the Banyan network.[9,16] In these studies, it is shown that when a processor makes a request for multiple resources, by allocating re-

sources with smaller distance functions, the amount of network blockage caused by the allocation of these resources is reduced.[8] A tree network is proposed to aid the scheduler in choosing a resource to allocate. The tree network has a delay of $0(\log_2 n)$ in selecting a free resource (n is the total number of resources), and most notably, the scheduling of multiple requests is done sequentially.

A few comments can be made about the previous studies. First, the scheduling algorithms are centralized. For mapping n requesting processors to n resources, the scheduling algorithm has a worst case complexity of $0(n*\log_2 n)$. This complexity depends on the number of requesting processors, which is practical when n is small or when requests are not very frequent. Second, for scheduling requests on interconnection networks with logarithmic delays such as binary n-cube, Banyan and Omega, no optical scheduling algorithm has been established.

The objective of this paper is (a) to study the performance of distributed scheduling algorithms with a scheduling time that is independent of the number of requests made in the network but only dependent on the delay in the network and compare the performance against centralized scheduling algorithms; (b) to design interconnection networks to support distributed scheduling.

The basic assumption made in this study is that each processor makes a request for one resource, although there may be multiple requests outstanding. The extension to the case when multiple resources are requested simultaneously is discussed in Section 5. Two types of request characteristics are identified. First, the resource requested by the processor must be continuously connected to the processor for the duration of the request. In this case, the RSIN may not be completely free when a set of new requests are initiated. Networks with logarithmic delays such as the Omega and binary n-cube may have a high blocking probability. A distributed cross-bar RSIN with no blocking is preferable in this case. This is discussed in Section 2. Second, requests are made in a block transfer mode. That is, a free resource is connected to a requesting processor for a short duration of time. After the request is sent to the resource, the connection between the processor and the resource is broken. The resource will continue to service the task, and the processor is free to generate new requests in the future. In this case, the network is almost or completely free when a set of new requests is initiated. We present in Sections 3 and 4 the centralized and distributed scheduling of request on Omega and binary n-cube networks with the assumption that the network is completely free when a set of requests is initiated. The performance of partially busy networks is presented elsewhere.[23]

2. DISTRIBUTED SCHEDULING ALGORITHM FOR THE CROSS-BAR SWITCH

We present in this section the design of a cross-bar switch to support distributed scheduling algorithms. The motivations behind studying cross-bar switches are that it is non-blocking and it is very suitable for VLSI implementation. It has been shown that cross-bar communication networks are favorable as compared to Banyan networks for VLSI implementation provided that the whole network is implemented on one chip.[6]

Figure 2 shows the overall structure of a cross-bar network. Processor i, $0 \leq i < n$, initiates a request by sending a request signal to the switch along the i-th row. Resource j, $0 \leq j < m$, indicates that it is free by sending a resource signal along the j-th column. At cell $C_{i,j}$ where there are request and resource signals, the switch is set on and data transfer can begin. The request signal is removed from any further cells along the i-th row. Similarly, the resource signal is removed from any further cells along the j-th column. Each cell in the switch has enough intelligence to resolve the conflicts and to route the requests. There is a control latch in each cell to indicate the state of the switch. It is obvious that there is no centralized control for the routing of requests.

Because requests can appear and disappear at any time, it is important that a change in request state for one processor does not affect the state of allocation of other processors. To illustrate this referring to Figure 2, if the request signal to cell $C_{i,j}$ is removed, then the latch in $C_{i,j}$ is reset and the resource becomes free. The resource signal will again propagate down the j-th column. Processor k may have made a request previously. Since resource j was busy, it tried to search for another resource and found one. The new resource signal passed along the j-th column should be ignored in cell $C_{k,j}$ in order not to upset the state of a previous allocation. To solve this problem, we assume that the system operates in two modes: request mode and reset mode. In the request mode, processors can make requests for free resources. In the reset mode, processors can relinquish previously acquired resources. This method degrades performance because requests and resets cannot operate concurrently. However, a single signal line suffices to indicate which mode is active. Other alternatives which allow concurrency in requests and resets include (a) the use of state saving latches in each cell and (b) the use of separate request and reset control lines. These alternatives require more hardware and will be investigated in the distributed Omega and binary n-cube networks.

Referring to Figure 2(b), the inputs and outputs of cell $C_{i,j}$ which connects processor i and resource j have the following meaning:

(request mode)
$$X_{i,j} = \begin{cases} 0 & \text{processor } i \text{ is not searching for a free resource} \\ 1 & \text{processor } i \text{ is searching for a free resource} \end{cases}$$

(reset mode)
$$X_{i,j} = \begin{cases} 0 & \text{processor } i \text{ does not want to change the state of allocation} \\ 1 & \text{processor } i \text{ wishes to relinquish the allocated resource} \end{cases}$$

$X_{i,j}$ always returns to 0 at the end of each mode

$$Y_{i,j} = \begin{cases} 0 & \text{resource } j \text{ is busy and cannot accept any request} \\ 1 & \text{resource } j \text{ is free and can accept a new request} \end{cases}$$

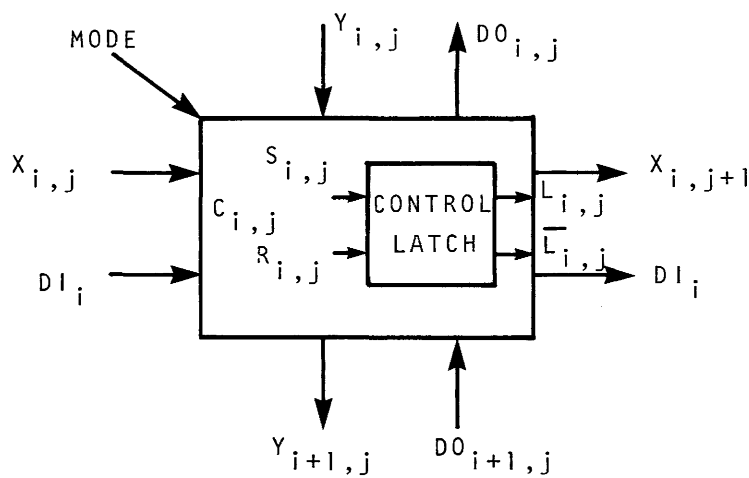
DI_i—data line to send data from the i-th processor

RESOURCES

PROCESSORS

PROPAGATION OF SIGNALS

(a) Structure of a cross-bar switch



(b) Structure of a cell

Figure 2—A cross-bar switch to support decentralized scheduling

$DO_{i,j}$—data line for the j-th resource to receive data from the i-th processor

$$L_{i,j} = \begin{cases} 0 & \text{switch is off; any request made by processor } i \text{ is passed to the next cell, } C_{i,j+1} \\ 1 & \text{switch is on; processor } i \text{ is connected to resource } j \end{cases}$$

$S_{i,j}/R_{i,j}$—the set/reset signal to the control latch in cell $C_{i,j}$
MODE—controls the cell to be in request or reset mode.
The input/output relationship of the control signals is shown in the truth table in Table I.

In the request mode, the latch is set ($S_{i,j} = 1$) if processor i is making a request and resource j is available. If resource j is not available ($Y_{i,j} = 0$), then the request signal is passed to the next cell ($X_{i,j+1} = X_{i,j}$). The resource signal to the next cell ($X_{i+1,j}$) depends on the state of control latch in the cell. If $Y_{i,j} = 0$, then $Y_{i+1,j} = 0$. If $Y_{i,j} = 1$ and $X_{i,j} = 1$, then the control latch is set and $Y_{i+1,j} = 0$. Since the $X_{i,j}$ signal returns to 0 at the end of the request mode, but the $Y_{i,j}$ signal may still be kept at 1, so $Y_{i+1,j}$ equals the output of the control latch ($\overline{L}_{i,j}$) when $X_{i,j} = 0$ and $Y_{i,j} = 1$. For those processors which have made requests previously, the state of allocation is not disturbed in the current request mode and data transmission can continue. In the reset mode, if processor i issues a reset signal, all the control latches in row i of the switch are reset. The logic equations for the controls and outputs are also shown in Table I. The design of cell $C_{i,j}$ is shown in Figure 3.

The boundary connections for the switch are as follows. Each $X_{i,m}$ signal is connected directly back to P_i. Similarly, each $Y_{n,j}$ signal is connected back to R_j. Suppose P_i makes a request by setting $X_{i,0} = 1$ and it receives at the end of the request cycle, $X_{i,m} = 1$, this means that the request is not satisfied and P_i should resubmit its request in the next request cycle. Likewise, resource R_j indicates that it is free by setting $Y_{0,j} = 1$. If at the end of the request cycle, $Y_{n,j} = 1$, this means that the resource is not allocated and R_j should send out the $Y_{0,j} = 1$ signal continuously. Otherwise, it will set $Y_{p,j} = 0$ to indicate that it is allocated.

Requests and resets are accepted at the beginning of the corresponding cycle. They are not accepted in the middle of a cycle because the next cycle cannot start until all the signals in the current cycle have settled. In each cycle, the signals propagate from the top left corner at 45° to the bottom right corner (Figure 2) on a wave-like motion. The maximum time for signal propagation is, therefore, proportional to $n + m$. In the request cycle, the maximum gate delays in each cell is 4 because of two gate delays in the control latch. The maximum length of the request cycle is $4(n + m)$ gate delays. In the reset cycle, the maximum delay in each cell is 1 due to the mode control gate. The maximum length of the reset cycle is $(n + m)$ gate delays.

A final remark about the scheduling algorithm is that it is asymmetric. That is, it favors processors with lower index numbers. In order to design an algorithm that is symmetric and to allow requests and resets to be initiated dynamically, more control lines are needed. Resources that are available

TABLE I—Truth table and control signals for cell $C_{i,j}$

Inputs		Outputs			
$X_{i,j}$	$Y_{i,j}$	$X_{i,j+1}$	$Y_{i+1,j}$	$S_{i,j}$	$R_{i,j}$
0	0	0	0	0	0
0	1	0	$\overline{L}_{i,j}$	0	0
1	0	1	0	0	0
1	1	0	0	1	0

$X_{i,j+1} = X_{i,j}\,\overline{Y}_{i,j}$
$Y_{i+1,j} = \overline{X}_{i,j}\,Y_{i,j}\,\overline{L}_{i,j}$
$S_{i,j} = X_{i,j}\,Y_{i,j}$
$R_{i,j} = 0$
$DO_{i,j} = L_{i,j}\,DI_i + DO_{i+1,j}$
(a) Request mode

Inputs		Outputs			
$X_{i,j}$	$Y_{i,j}$	$X_{i,j+1}$	$Y_{i+1,j}$	$S_{i,j}$	$R_{i,j}$
0	0	0	0	0	0
0	1	0	1	0	0
1	0	1	0	0	1
1	1	1	1	0	1

$X_{i,j+1} = X_{i,j}$
$Y_{i+1,j} = Y_{i,j}$
$S_{i,j} = 0$
$R_{i,j} = X_{i,j}$
$DO_{i,j} = L_{i,j}\,DI_i + DO_{i-1,j}$
(b) Reset mode

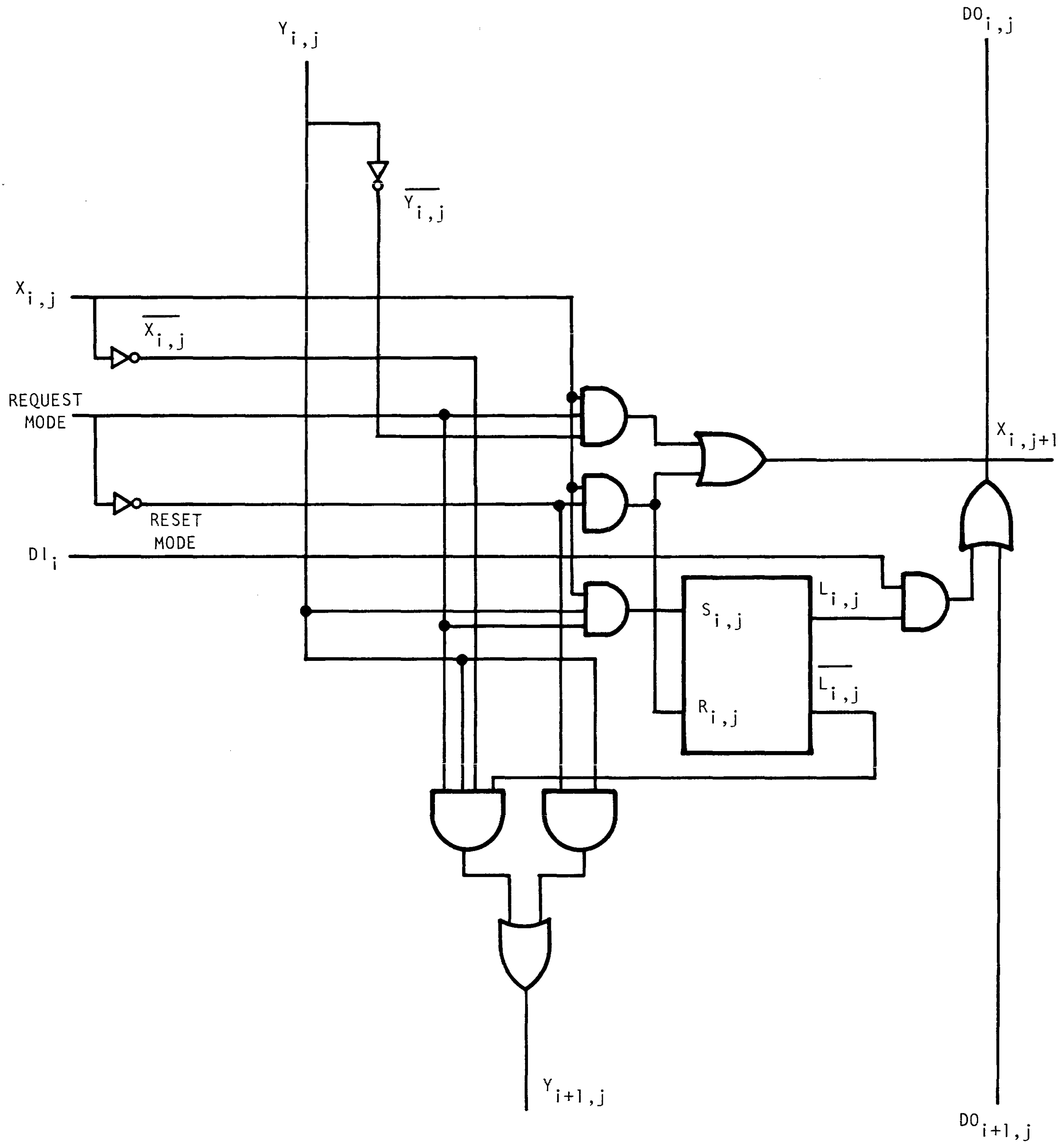


Figure 3—A cell in the distributed cross-bar switch

can send a pulse of a short duration along a column. Only processors that receive a pulse will be assigned the resource. In this sense, the pulse behaves like a token. When different resources issue tokens randomly, the algorithm is symmetric.

3. CENTRALIZED RESOURCE SCHEDULING ON NETWORKS WITH LOGARITHMIC DELAYS

In the remainder of this paper, we present the resource scheduling algorithms for interconnection networks with logarithmic delays. Specifically, we study the Omega and binary n-cube networks and first establish the optimal scheduling algorithm of these networks. Based on the optimal behavior, we compare the performance degradation of other heuristics and the distributed algorithm.

The Omega[11] and binary n-cube[15] networks are chosen for their simplicity and versatility. The basic element in these networks is a 2 input, 2 output interchange box which allows a straight or a diagonal connection. For a network connecting N inputs to N outputs (N is a power of 2), there are $\log_2 N$ stages and $\frac{N}{2} * \log_2 N$ interchange boxes. The delay in the net-

work is therefore O ($\log_2 N$). Figure 8 shows an example of an Omega network with $N = 8$.

The Omega network is equivalent to the binary n-cube network with the difference that it operates in the reverse direction. Using these networks as RSINs, they are statistically identical. The performance of these networks is evaluated by selecting a random subset of processors and resources and finding the maximum number of resource allocations. If the Omega network can be rearranged into a binary n-cube network, then their performance as RSINs are identical. This rearrangement is exemplified in the Omega network in Figure 8. If $B_{0,1}$ and $B_{1,1}$ are moved so that they are adjacent to $B_{0,3}$ and $B_{1,3}$, and with proper relabeling of processors and resources, the Omega network is transformed into a cube network. In the following discussion, we will only present the result on the Omega network. The performance of the binary n-cube network is equivalent.

(a) *Optimal Scheduling Algorithm*

Simulation results presented in Franklin[6] show that with $N = 8$, there is a message blocking probability of about 30% using address mapping. We show in this section that there is virtually no blocking when the Omega and binary n-cube networks operate as RSINs.

The results are obtained by exhaustive enumeration over all the possible combinations of connections for a subset of requesting processors and free resources. Because of the large number of combinations, only networks with $N = 8$ can be studied. Even in this case, the total number of possible combinations is slightly under 600 million. The large number of combinations is attributed to the fact that the order of connections is important. For a set of i requesting processors and j free resources, there are $i! * j!$ possible ordered connections.

A faster method is developed by observing that each box can be set in 2 states. With 12 interchange boxes, there are $2^{12} = 4096$ states or possible connections. These 4096 possible connections are arranged into multiple trees so that the maximum number of connections can be found efficiently. Using this method, the enumerations were completed using 10 hours of CPU time and 64 K bytes of memory on a VAX 11/780.

A selected set of the simulation results are plotted in Figures 4 and 5 for the blocking probability and standard deviation of processor allocations when the number of requesting processors equals the number of free resources. These results are based on the assumption that the network is completely free before the allocations. The average processor blocking probability is defined as:

$$\text{processor blocking probability} = \frac{\text{Number of allocated processors}}{\text{Number of requesting processors}}$$

The blocking probability must be interpreted correctly. For example, if there are 4 processors and 2 resources, then at most 2 processors can be allocated resources and the minimum blocking probability is 50%. If there are 2 processors and 4 resources, the minimum blocking probability is 0%.

It is seen that the blocking probability and the standard deviation of processor allocations are very small. We can conclude that with a good scheduling algorithm, the Omega and binary n-cube network serve almost equally as well as the distributed cross-bar switch for resource sharing.

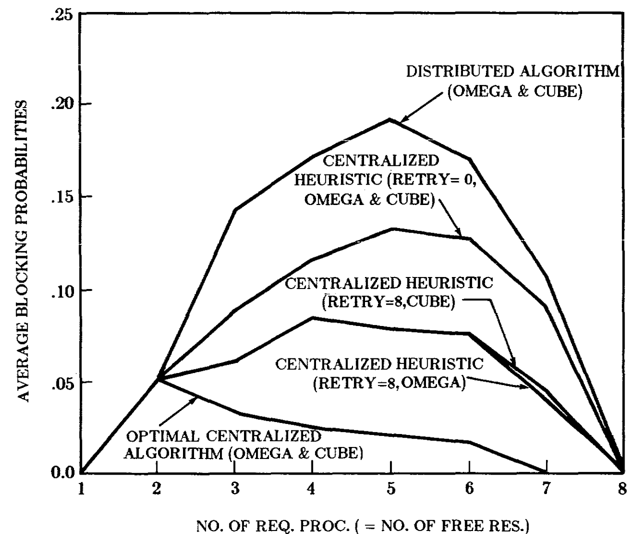


Figure 4—Blocking probability for resource allocations on Omega and Cube networks ($N = 8$)

(b) *Centralized Scheduling Heuristic*

As a comparison, we present a centralized heuristic and compare its performance against the optimal algorithm. Let

$$P_R = \text{Set of requesting processors} = \{P_i, P_{ii}, P_{iii}, \ldots P_x\}$$
$$R_A = \text{Set of free resources} = \{R_i, R_{ii}, R_{iii}, \ldots R_y\}$$

We assume that the processors and resources in P_R and R_A are ordered by their index numbers. A parameter of the heuristic

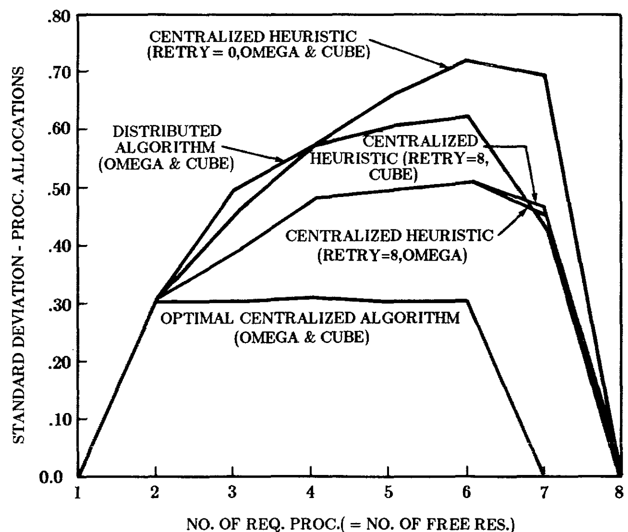


Figure 5—Standard deviation of number of requesting processors allocated for Omega and Cube networks ($N = 8$)

is the number of retries ($0 \leq \text{RETRY} \leq y-1$). Supposed P_i fails to be connected to R_j due to a blocked connection, then the heuristic successively retries the next set of RETRY free resources to see if a connection can be made. Whether a connection can be made within the fixed number of retries or not, the next processor in P_R is always matched with the first free resources in R_A immediately following the resource matched for the current processor. The heuristic, written in pidgin Algol, is shown in Figure 6.

```
PROCEDURE cent_heuristic

/*  Assume that match (P_i, R_j) is a boolean procedure which
    returns TRUE if P_i can be connected to R_j and FALSE
    otherwise.  If TRUE is returned, the connection is actually made.

    i - index of a requesting processor (i = φ means there is no
        requesting processor)

    j - index of a free resource (j = φ means there is no free resource)

    r - variable indicating the number of retries
*/

i=1; j=1;  /* initialization */

WHILE (i ≠ φ AND j ≠ φ) DO
    BEGIN
    done = FALSE; r=0;

    WHILE (NOT match (P_i, R_j) AND done .EQ. FALSE) DO
        BEGIN
        r = r+1;
        IF (r > RETRY)  /* Test for # of retries */
           THEN  done = TRUE
           ELSE  j ← next free resource in R_A;
        END;

    i ← next requesting processor in P_R;
    j ← next free resource in R_A;
    END
END
```

Figure 6—Centralized heuristic for resource allocation

To illustrate the heuristic, consider an 8 by 8 Omega network with $P_R = \{0, 3, 4, 5\}$, $R_A = \{0, 1, 3, 4\}$ (see Figure 8),

1. The algorithm connects P_0 to R_0 and is successful.
2. The algorithm connects P_3 and R_1 and is successful.
3. The algorithm tries to connect P_4 to R_3, but is blocked.

If $\text{RETRY} = 0$, then the algorithm connects P_5 to R_4 and is successful.

If $\text{RETRY} = 1$, then the algorithm tries to connect P_4 to R_4 and is successful. It continues to connect P_5 to R_3 and is successful. For this example, the resource utilization is 100% if $\text{RETRY} \geq 1$, otherwise, it is 75% for $\text{RETRY} = 0$.

The procedure *match* in Figure 6 has a complexity of $0(\log_2 N)$. (It is proportional to the number of stages in the network). The worst case complexity of the heuristic for x requesting processors and y free resources ($0 < x, y \leq N$) is, therefore, $0(N(\text{RETRY}+1)\log_2 N)$. If $\text{RETRY} = 0$, the heuristic has complexity $0(N^* \log_2 N)$. If $\text{RETRY} = N-1$, the heuristic has complexity $0(N^2 \log_2 N)$.

Since the heuristic assumes a predetermined sequence of allocations and no backtracking is provided if a wrong decision is made, the heuristic is sub-optimal. The performance of the heuristic with $\text{RETRY} = 0$ and $\text{RETRY} = 8$ are shown in Figures 4 and 5. It is seen that the blocking probability is higher than the optimal case (around 7%). As the number of retries is increased, the blocking probability reduces. Further, the Omega and the binary n-cube networks have different performance on the centralized heuristic. This is due to the fact that the order in which resources are tried are different in the two networks. Although the Omega and binary n-cube networks seem to have identical performance for the centralized heuristic with $\text{RETRY} = 0$, the Omega network has worse performance when the number of processors and resources are different.

4. DISTRIBUTED RESOURCE SCHEDULING ON NETWORKS WITH LOGARITHMIC DELAYS

The centralized scheduling algorithm has a high overhead when the number of processors and resources to be scheduled is large since every requesting processor has to be scheduled sequentially. In a distributed algorithm, all the requesting processors are scheduled in parallel. The resource scheduling overhead is, therefore, proportional to the delay time in the network ($0(\log_2 N)$) and independent of the number of requesting processors.

The distributed algorithm is implemented by distributing the scheduling intelligence into the interconnection network so that there is no centralized control. Each exchange box can resolve conflicts and route requests to the appropriate destination. If a request is blocked, it will be sent back to the originating exchange box in the previous stage. Request routing is, thus, dynamic and all the exchange boxes operate independently.

Before the algorithm is described, some symbols must be defined. The information paths for exchange box j in stage i, $B_{i,j}$, is shown in Figure 7. There are four types of signals, S, Q, J and D:

S—carries information about the number of resources reachable from this link

$$Q = \begin{cases} 1 & \text{a request of a free resource is made on this link} \\ 0 & \text{otherwise} \end{cases}$$

$$J = \begin{cases} 1 & \text{a block has been detected in stages after the current stage and the request along this link is rejected} \\ 0 & \text{otherwise} \end{cases}$$

D—data transmission links

RA—resource availability register which stores the number of resources reachable from an output terminal of $B_{i,j}$.

For simplicity of representation, subscripts of symbols for signals incident upon and originating from $B_{i,j}$ are set to be i,j. The index of the box that they are connected to is not included in the representation as a mapping function. There are four types of superscripts, UL (upper left), UR (upper right), LL (lower left) and LR (lower right) and they indicate the corner of the exchange box that the signal links are connected to. The distributed scheduling algorithm utilizes these signals to connect the data paths from the $i-1$-*st* stage ($D^{UL}_{i-1,j}$, $D^{LL}_{i-1,j}$) to the $i+1$-*st* stage ($D^{UR}_{i,j}$, $D^{LR}_{i,j}$).

Consider a situation when the network is completely free,

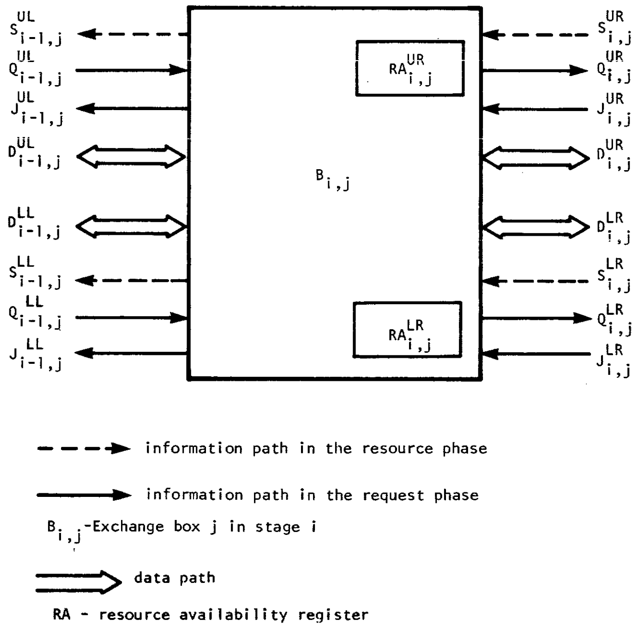


Figure 7—The information paths of an exchange box in the distributed scheduling algorithm

and there is a set of requesting processors and free resources. All the resource availability registers are set to be zero initially. We will generalize later to the situation in which requests can be initiated dynamically.

The algorithm consists of two phases. In the first phase *(Resource Phase)*, information concerning the number of free resources is passed from the resource side to the processor side. Specifically, each resource that is free sends a "+1" along the S link to the exchange box connected to it. Referring to Figure 7, which shows exchange box $B_{i,j}$, the dashed lines represent the information flow in the resource phase. The exchange box receiving this information increments the corresponding resource availability registers. It then adds the two numbers stored in the two resource availability registers and sends the result to the two exchange boxes connected in stage $i-1$. Conceptually, the numbers $S_{i,j}^{UR}$ and $S_{i,j}^{LR}$ represent the number of resources reachable from the upper and lower output terminals of $B_{i,j}$. Therefore, the total number of resources reachable from this box is $S_{i,j}^{UR} + S_{i,j}^{LR}$ and this information is passed to the two exchange boxes connected in the previous stage along links $S_{i-1,j}^{UL}$ and $S_{i-1,j}^{LL}$. The delay for this phase is proportional to the number of stages of the network.

As an example, refer to Figure 8. Suppose processors P_0, P_3, P_4 and P_5 are making requests and resources R_0, R_1, R_4 and R_5 are free. Resource availability information is passed from the resource side to the processor side starting with stage 2. Box $B_{2,0}$ receives "+1" from R_0 and R_1. Therefore, it passes a "+2" to boxes $B_{1,0}$ and $B_{1,1}$. Likewise, box $B_{2,2}$ receives "+1" from R_4 and R_5 and passes this information to boxes $B_{1,2}$ and $B_{1,3}$. The propagation of this information is similar in stages 1 and 0. At the end of the resource phase, P_0, P_3, P_4 and P_5 know that there are 4 resources available.

In the second phase *(Request Phase)*, the network propagates the requests from the processors to the resources. This uses the information that is obtained in the resource phase. The maximum total number of request and rejection signals pending in each exchange box is 2 since the exchange box can only make two connections at any time. For example, it is impossible to have two rejection signals received together with a request signal, because in order for the rejection signals to be received, two request signals must have been received earlier. A new request cannot be received until the two previous requests have been rejected. Therefore, we can have any one of the following six combinations of signals pending in an exchange box: 2 Qs, 2 Js, 1 Q and 1 J, 1 J, 1 Q, or no signal pending.

When multiple signals are pending in a box, priority must be set to determine the order of servicing these requests. Two priority rules are used:

(P1) For two request signals received ($Q_{i-1,j}^{UL} = 1$, $Q_{i-1,j}^{LL} = 1$), the request originating from the top input terminal ($Q_{i-1,j}^{UL}$) has priority over the other ($Q_{i-1,j}^{LL}$).

(P2) For one request and one reject signal received, the reject signal has priority over the request signal in service.

In servicing a request or a reject, two service rules are applied.

(S1) To service a request ($Q_{i-1,j}^{UL}$ or $Q_{i-1,j}^{LL}$), find a free output link where free resources can be accessed (contents of resource availability register is greater than zero). If both output links are free, then $S_{i,j}^{UR}$ is checked before $S_{i,j}^{LR}$. If such an output link is found, the output link is marked busy so that no further request can be made along this link and a request is sent to stage $i+1$. If the free output links do not lead to any free resources, a reject signal is sent from the original input terminal to stage $i-1$.

(S2) To service a reject ($J_{i,j}^{UR}$ or $J_{i,j}^{LR}$), the corresponding resource availability register ($RA_{i,j}^{UR}$ or $RA_{i,j}^{LR}$) is set to zero to indicate that no free resource is reachable from this output terminal. The output terminal is marked free and service rule (S1) is applied to search for another available output terminal where free resources can be reached.

For the six possible input combinations of signals pending in $B_{i,j}$, the sequence of priority and service rules applied is shown in Table II.

If a request successfully reaches a free resource, the resource sends a "-1" along the S link to the exchange box connected to it. For exchange box $B_{i,j}$ receiving a "$-k$" ($k = 1, 2, \ldots$) along the S link ($S_{i,j}^{UR} = -k$ or $S_{i,j}^{LR} = -k$), if the content of the corresponding resource availability register is zero, then nothing is done. If not, the corresponding resource availability register is decremented and the "$-k$" information is passed to stage $i-1$ along $S_{i-1,j}^{UL}$ and $S_{i-1,j}^{LL}$. If both $S_{i,j}^{UR}$ and $S_{i,j}^{LR}$ are negative ($S_{i,j}^{UR} = -k_1$ and $S_{i,j}^{LR} = -k_2$), then both $RA_{i,j}^{UR}$ and $RA_{i,j}^{LU}$ are decremented and "$-(k_1 + k_2)$" is sent along $S_{i-1,j}^{UL}$ and $S_{i-1,j}^{LL}$ to stage $i-1$.

Referring to the example in Figure 8, $B_{1,1}$ in stage 1 receives two requests. Since only one output terminal leads to free resources, the request originating from $B_{0,3}$ is rejected. This

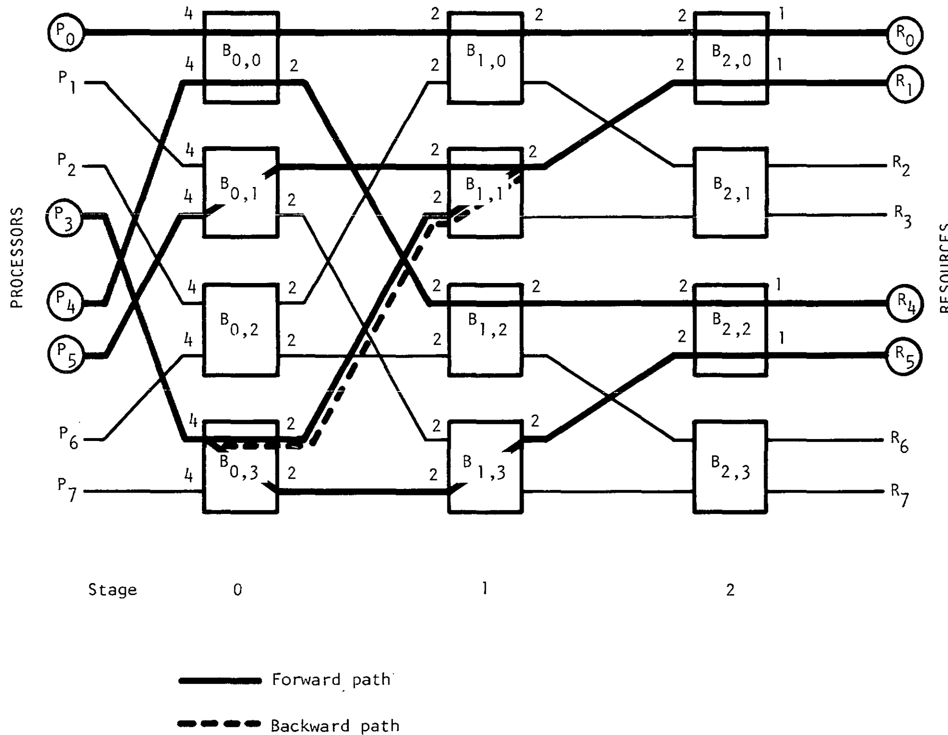


Figure 8—Example of Omega network with four requesting processors and four free resources (25% of requests are blocked and backtracked; 100% resource allocation; average delay = 4.67 units)

request, subsequently, finds another route via $B_{1,3}$ and $B_{2,2}$ to R_5. The average delay time is 4.67 units in this example (a unit is the time to pass through an exchange box).

The algorithm described above does not preclude dynamic operation. In fact, requests can be initiated at random times and they will be routed to a free resource or be rejected. The operation of the exchange box can be completely asynchronous. An accepted request is known to a processor when an acknowledgement is received along the data link. A request is rejected when a rejection signal is received by the processor along the J signal link. A rejected request can be retried later.

The performance of the distributed algorithm is again plotted in Figures 4 and 5 and it is identical for the Omega and binary n-cube networks. It is seen that the blocking probability is less than 20% in all cases and compares favorably with the optimal algorithm and centralized heuristic. The standard deviation is approximately doubled as compared with the optimal case. The average delay time for a request to access a free resource or be rejected is shown in Figure 9. The delay is never greater than 4.2 units of time in which the delay through an exchange box is 1 unit. The delay time of the algorithm is dependent on the delay in the network and not on the number of requesting processors.

TABLE II—Sequence of priority and service rules applied for the six possible combinations of signals pending in $B_{i,j}$

Combinations of signals pending in $B_{i,j}$	Sequence of priority and service rules applied
2 Q	P1, S1, S1
2 J	S2, S2
1 Q, 1 J	P2, S2, S1
1 Q	S1
1 J	S2
0 Q, 0 J	no action

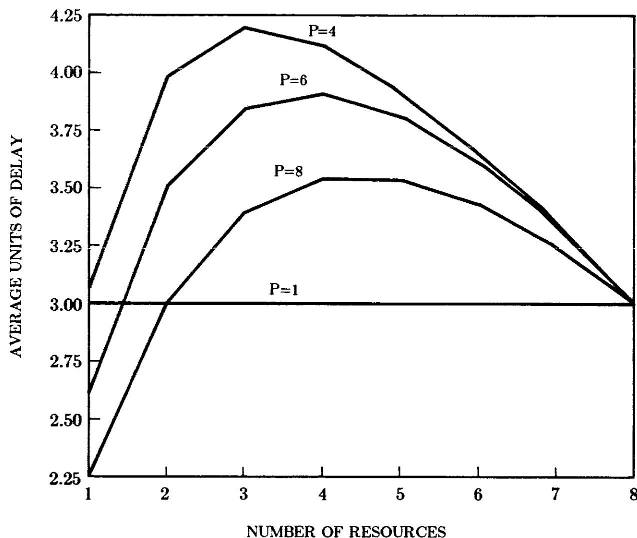


Figure 9—Average delay time for distributed algorithm in 8 by 8 Omega and binary n-cube networks (a unit is the time to pass through an exchange box)

5. CONCLUSION AND EXTENSIONS

In this paper, we have presented the scheduling of resources on interconnection networks. The resource scheduling problem is different from the conventional address mapping problem on interconnection networks because a request is not directed towards a particular destination address but to any one of a pool of destination addresses for free resources. A broadcast technique does not work effectively because it precludes other processors in making requests when one of the processors is making a request. Centralized scheduling algorithms are inefficient because all the requesting processors must be serviced sequentially. A distributed scheduling algorithm allows the scheduling of all the processors to be performed in time proportional to the delay in the network.

An interconnection network for resource sharing may operate in two ways. First, the network is "circuit switched" and the processor and resource are continuously connected for the duration of use. In this case, the network may be partially busy when a new request is initiated. To avoid excessive blocking, the network should provide conflict-free access even when other connections are present. A distributed scheduling algorithm is designed for the cross-bar switch and the implementation of a cross-bar switch cell is presented. Each cell can be implemented with 12 gates and a flip-flop when the data path is one bit wide. The cell is designed with the minimum number of signal lines. As a result, the scheduling algorithm is divided into two phases and the algorithm is asymmetric, that is, priority is induced into the scheduling of multiple requests. In order to allow the algorithm to operate in one phase, more signal lines are needed between adjacent cells as shown in the distributed Omega and binary n-cube networks. To remove the assymmetry in the scheduling, each resource can send a short pulse along the resource availability line. This pulse acts like a token and one of the requesting processors receiving this pulse will be allocated. When different resources issue tokens randomly, the algorithm is symmetric.

Second, the network is "packet switched" or operates in block-transfer mode. In this case, the resources are connected to the processors for a short duration of time and can relinquish the network after tasks are assigned. When a new set of requests are initiated, the network is almost or completely free. Networks with logarithmic delays are suitable for this application. An optimal centralized scheduling algorithm has been studied for the Omega and binary n-cube networks. It is shown that there is an average blocking probability of 1%. This means that these networks have behavior close to the cross-bar switch for resource sharing in a block transfer mode.

The centralized optimal algorithm has exponential time complexity. We studied, respectively, two centralized heuristics (with time complexities $O(N^2 \log_2 N)$ and $O(N \cdot \log_2 N)$) and a distributed algorithm (with time complexity $0(\log_2 N)$). In the distributed algorithm, each exchange box in the network operates asynchronously and is responsible for resolving multiple requests directed to it. Resource availability information is also passed along the network to the processors. The control of the network can be hardwired or microprogrammed. The blocking probability increases as the time complexity decreases. In the worst case (distributed algorithm), the blocking probability is around 19%.

Several extensions can be included in the design. We discuss them briefly here.

1. The resources connected on the network do not have to be identical. In a general system, there are multiple types of resources, each type is made up of a set of identical resources. The algorithms discussed have to be modified by identifying the type of resource requested by a processor and the type of resource available on a resource availability line. This can be done by sending a binary request code (instead of 1 bit) in the distributed algorithms. In the distributed Omega and binary n-cube networks, multiple resource availability registers have to be used in each exchange box.
2. There is a tradeoff between the time complexity of the algorithm and the number of signal lines between two adjacent cells in the distributed RSINs. A one-bit data paths is assumed in the distributed cross-bar switch. In the distributed Omega and binary n-cube networks, parallel data paths are assumed. This can be reduced by appropriate multiplexing at the external chip interface level.
3. The scheduling algorithms can be extended to the case when multiple resources are requested by a processor to operate in a broadcast mode. In the distributed cross-bar switch, a count can be sent with a request signal. Each time a free resource is allocated to this request, the count is decremented by one before it is sent to the next cell. In the centralized heuristic for the Omega and binary n-cube networks, request for multiple resources is serviced by searching for a free resource and allocating it until the required number of resources are allocated. In the distributed Omega and binary n-cube networks, the exchange boxes are extended so that they can operate in broadcast mode. That is, an input terminal to an exchange box can be connected to the two output terminals simultaneously. The algorithm is modified to proceed as

follows. A count K is sent with each request or reject signal. This count indicates the number of additional resources needed by the request or reject. Referring to Figure 7, a request is sent to the upper request line ($Q_{i,j}^{UR}$) if the content of the resource availability register, ($RA_{i,j}^{UR}$), is greater than K. Otherwise, the request is sent along both the upper and lower request lines. Of course, if the content of any resource availability register is zero, a request is not sent along the corresponding request line. If $(RA_{i,j}^{UR}) + (RA_{i,j}^{LR}) > K$, that is, the request cannot be satisfied completely, then a reject signal with count $= K - (RA_{i,j}^{UR}) - (RA_{i,j}^{LR})$ is sent from the original input terminal to stage $i - 1$ to search for additional resources. It is also assumed in the generalized distributed algorithm, that the two priority rules (P1 and P2) are followed. To avoid deadlock in the generalized distributed algorithm, a requesting processor should relinquish its allocated resources if it cannot find the required number of resources and resubmit its request again later.

The distributed algorithm implemented in each exchange box does not preclude operation under the address mapping mode. Further, the theory underlying the design of the distributed Omega and binary n-cube networks can be applied to other interconnection networks such as the Banyan,[7] and Delta.[14] In these networks, the number of processors and the number of resources can be different. The performance will be evaluated in the future. Future studies also include the performance evaluation of the algorithm under dynamic operations.

REFERENCES

1. Barnes, G. H., and S. F. Lundstrom. "Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems." *IEEE Computer,* 14 (1981) pp. 31–41.
2. Batcher, K. E., "STARAN Parallel Processing System Hardware," *Proc. of AFIPS 1974 National Computer Conf.,* Vol. 43, pp. 405–410, May 1974.
3. K. E. Batcher, "The Flip Network in STARAN," *Proc. of 1976 Int'l Conf. on Parallel Processing,* Michigan, pp. 65–71, 1976.
4. Burroughs Corp., *Final Report, Numerical Aerodynamic Simulation Facility Feasibility Study,* NASA Contractor Reports CR152284 and CR152285, Burroughs Corp., Paoli, PA, March 1979.
5. Feng, T. "Data Manipulating Functions in Parallel Processors and Their Implications." *IEEE Trans. Computer,* Vol. C-23, No. 3, pp. 309–318, Mar. 1974.
6. Franklin, M. A. "VLSI Performance Comparison of Banyan and Cross-bar Communication Networks." *Proc. of Workshop on Interconnection Networks,* pp. 20–28, Apr. 1980.
7. Goke, L. R., and G. J. Lipovski. "Banyan Networks for Partitioning Multiprocessor Systems," *Proc. 1st Annual Comp. Architecture Conf.,* pp. 21–28, Dec. 1973.
8. Goke, L. R., *Banyan Networks for Partitioning Multiprocessor Systems* Ph.D. Thesis, Univ. of Florida, 1976.
9. Jenevein, R., D. Degroot and G. J. Lipovski. "A Hardware Support Mechanism for Scheduling Resources in a Parallel Machine Environment." *Proc. of 8th Annual Symposium on Computer Architecture,* pp. 57–66, May 1981.
10. Kuck, D. J. "ILLIAC IV Software and Application Programming." *IEEE Trans. on Comp.,* Vol. C-17, pp. 746–757, Aug. 1968.
11. Lawrie, D. "Access and Alignment of Data in an Array Processor." *IEEE Trans. Computers,* Vol. C-24, No. 12, pp. 215–255, Dec. 1975.
12. McDonald, W. C. and J. M. Williams, "The Advanced Data Processing Test Bed." *Proc. of COMPSAC 78,* pp. 346–351, March 1978.
13. Ornstein, S. M., et al., "Pluribus—A Reliable Multiprocessor." *Proc. AFIPS 1975 National Computer Conference,* AFIPS Press, Montvale, N.J., pp. 551–559, 1975.
14. Patel, J. H. "Performance of Processor-Memory Interconnections for Multiprocessors." *IEEE Trans. on Computers,* Vol. C-20, No. 10, pp. 771–780, Oct. 1981.
15. Pease, M. C. "The Indirect Binary *n*-cube Microprocessor Array," *IEEE Trans. on Computers,* Vol. C-26, No. 5, pp. 458–473, May 1977.
16. Rathi, B. D., A. R. Tripathi and G. J. Lipovski. "Hardwired Resource Allocators for Reconfigurable Architectures," *Proc. of 1980 International Conference on Parallel Processing,* pp. 109–117, Aug. 1980.
17. Sejnowski, M. C., et al. "Overview of the Texas Reconfigurable Array Computer." *AFIPS Conference Proceedings,* Vol. 49, pp. 631–642, 1980.
18. Siegel, H. J., and R. J. McMillen. "The Cube Network as a Distributed Processing Test Bed Switch." *2nd Int'l. Conf. on Dist. Comp. Sys.,* pp. 377–387, April 1981.
19. Siegel, H. J., and R. J. McMillen. "Using the Augmented Data Manipulator Network in PASM." *IEEE Computer,* Vol. 14, No. 2, pp. 25–33, Feb. 1981.
20. Stone, H. "Parallel Processing with the Perfect Shuffle." *IEEE Trans. on Computers,* Vol. C-20, No. 2, pp. 153–161, Feb. 1971.
21. Wu, C., and T. Y. Feng. "On a Class of Multistage Interconnection Networks." *IEEE Trans. on Computers,* Vol. C-29, No. 8, pp. 694–702, Aug. 1980.
22. Wulf, W. A., and C. G. Bell. "C.mmp—A Multi-mini Processor." *Proc. AFIPS 1972 Fall Joint Comp. Conf.,* Vol. 41, AFIPS Press, Montvale, NJ, pp. 765–777, 1972.
23. Hicks, A., *Resource Scheduling on Interconnection Networks.* M.S. Thesis, Purdue University, 1982.

APPLICATIONS OF COMPUTING

A microcomputer system for color video picture processing

by YOSHIKUNI OKAWA
Gifu University
Gifu, Japan

ABSTRACT

A color picture processing system is proposed. It consists of a microcomputer and a color video recorder. A picture is taken by a portable videotape recorder and a camera on cassette tapes in the field and brought back to the laboratory where the processing computer is installed.

The scenes are replayed on the videotape player on the monitor TV screen, from which signals are stolen by three A-D converters (one each for R, G, and B) and stored in the memory of the microcomputer.

The software package provides several commands which make it possible to process images on the CRT screen by man-machine interaction. A functional description of the commands is stated in some detail. One example of the application of this system is briefly described.

1. INTRODUCTION

A computing system to process color video pictures is described. The system consists of (1) a 16-bit microprocessor, (2) a color videocassette tape player, (3) a color monitor TV, and (4) a color graphic display.

We can record any scene onto videocassette tape by using a portable videotape recorder and a color TV camera. The scenes are replayed on the screen of a monitor TV. The signal voltages of red, green, and blue color components which drive the monitor TV are stolen and converted into three 8-bit digital signals and stored in the memory of the microcomputer. The scene will be processed digitally afterward by the software system provided by man-machine interaction.

As an example of a possible application of the system, the efficiency of road signs is studied. Various colors of tapes are placed on roads. Scenes are recorded on cassette tape in the field, carried back to the laboratory, and processed by the computer. The numerical measure of the recognizability of a sign against its background is defined and computed for all of the recorded scenes. The best color and form of a guiding line is determined for each road condition.

The cost of the proposed system is very low because microprocessors and color video picture recorders and players are produced massively by modern industry. A portable videotape recorder and a TV camera give mobility to the picture processing computer system. This computer will become a powerful tool in the field of digital color picture processing.

2. THE CONFIGURATION OF THE SYSTEM

The system configuration is shown in the block diagram of Figure 1. The central processing unit is a microprocessor (Z-8000) having 216 Kbyte of memory. A character display, a keyboard, a printer, and other usual computer peripherals are attached to the processor.

The picture input device is either a videotape player or a color TV camera. If the TV camera is used as a picture-taking instrument, scenes within the laboratory room in which the computer system is installed can be processed by the system. We call this an online processing mode.

For processing scenes outside the computer room, a combination of a TV camera and a portable videotape recorder is used. The outside scenes are recorded onto videocassette tapes. They are carried back to the computer room and replayed in the videotape player, whose pictures are displayed on the color monitor television set. This is called a time-freezing mode. Analogue signals are stolen from the driving circuit of the color television CRT tube. Red, blue, and green voltages are converted into three 8-bit digital form, which is stored in the memory of the computer by means of a DMA controller.

In the time-freezing mode, for example, we can stop the scene or replay it in slow motion. Although these capabilities originate from the intrinsic function of the videotape player, our picture processing system can make efficient use of them.

3. THE COMMANDS OF THE CONTROL PROGRAM

3-1. Image Sampling

Picture processing, in general, proceeds in a conversational fashion. A regular command form is

a prompt character from the control program

> a command character [,possible parameters] (CR).

a carriage return code

Various commands provided in the control program are described in the following:

1. A command to set a sampling window in a picture plane:

> W,SX,SY,NX,NY (CR)

The four command parameters SX, SY, NX, and NY assign a rectangular region in a picture plane, as shown in Figure 2. The parameters are key-input in a hexadecimal form and stored in the RAM area of the memory. This region will be sampled later.

2. A command to sample an image:

>I (CR)

The DMA controller is initialized and the image sampling is started by this command. One vertical line in a window is sampled in one frame of television pictures. Since there are 60 frames in a second, the sampling time is calculated by

$$t = \frac{NX}{60} \text{ (second)}$$

In general, the sampling time is directly dependent on the conversion time of the A-D converters used. If the speed of A-D conversion is increased, the sampling operation can be completed within 1/60 second.

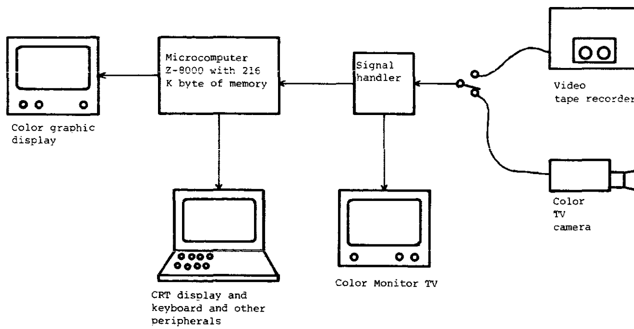


Figure 1—The system configuration

3-2. R, G, and B Display Commands

The stored image must be called out from the memory and displayed on the screen of the color graphic display. The following commands are provided in the control program:

(1) A command to erase the color graphic display screen.

>E(CR)

The screen of the graphic display is erased.

(2) A command to display a cross-sectional figure of an image: As stated before, R, G, and B signals of an image are sampled in 8-bit digital form. That is, the representation at one picture element is 3 bytes in the computer. The color graphic display has only 8 colors (3 bits) at each picture position. There is a significant gap between sampled image and displaying capability. We must design commands to overcome this difficulty.

First, we cut a three-dimensional distribution of an image in two pieces and make a cross-sectional distribution of the image. Three two-dimensional display lines are enough to display the image, which is easily shown on the CRT screen of the color graphic display. The command has the following form:

>H,h,v,F(CR)

where H is the command character, (h,v) indicates the starting point in a picture plane, and F is a Freeman code to specify cutting direction. One example of the displayed results is shown in Figure 3. The three height lines consist of red, green, and blue color dots. But at a picture element where at least two out of three colors have the same intensity level, colors other than red, green, and blue are displayed, since the dots are overlapped.

(3) A command to display a thresholded picture: If the sampled red, green, and blue brightness levels are thresholded at each picture element, the resulting image can be displayed on the screen of the color graphic display. The following command is provided for this purpose:

>F,RT,GT,BT(CR)

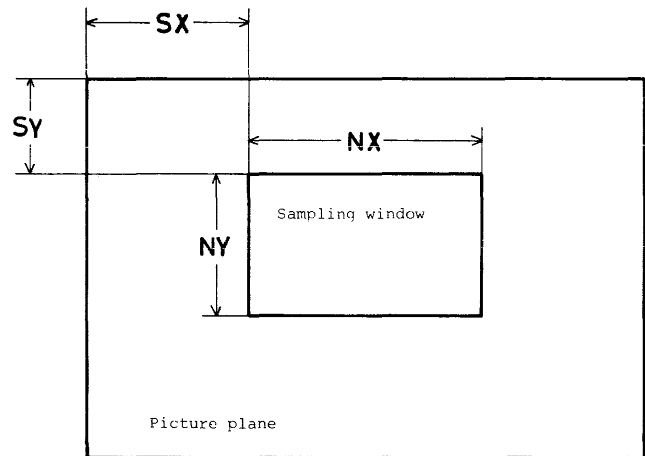


Figure 2—Sampling window
(the picture in the window is taken into the computer memory)

RT, GT, and *BT* are the threshold values in a hexadecimal form. Let us write sampled red, green, and blue brightness levels at a picture point *(i,j)* as R_{ij}, G_{ij}, and B_{ij}, respectively. Concerning the graphic display, if $r_{ij} = 1$, then a red spot is displayed on the (i,j) grid of CRT screen; and if $r_{ij} = 0$, a red spot does not appear at that point. The terms g_{ij} and b_{ij} can be defined in the

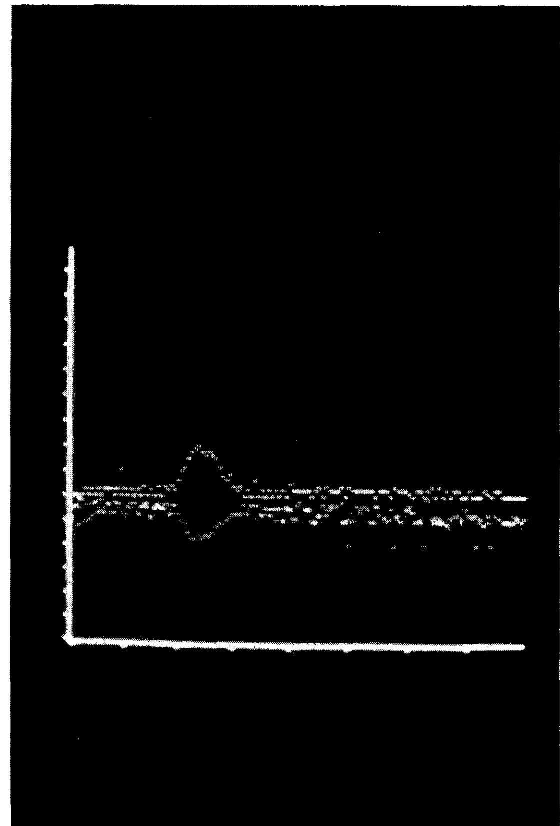


Figure 3—An example of a cross-sectional display of an image
(originally, red, green, and blue dots were displayed)

TABLE I—Standard color card used for the calibration

No.	Munsell color	x	y	R	G	B
1	7R 4.5/16.4	0.600	0.320	128	80	72
2	4G 5.2/13.5	0.210	0.485	52	96	96
3	5.5B 4.7/11.7	0.145	0.205	36	80	128
4	1.5RP3.5/19.8	0.370	0.150	100	78	96
5	4BG3.5/12.8	0.120	0.335	40	78	84
6	7.5Y 8.5/13.9	0.450	0.500	160	120	64
7	N 9	0.310	0.316	112	112	112

same manner. Then the action of the command can be stated as follows:

$$\text{If } R_{ij} > RT, \text{ then } r_{ij} = 1, \quad \text{otherwise } r_{ij} = 0.$$
$$\text{If } G_{ij} > GT, \text{ then } g_{ij} = 1, \quad \text{otherwise } g_{ij} = 0.$$
$$\text{If } B_{ij} > BT, \text{ then } b_{ij} = 1, \quad \text{otherwise } b_{ij} = 0.$$

It must be pointed out that the selection of the threshold values *RT, GT,* and *BT* changes the displayed figures. For example, if we set the threshold values very high, the image on the CRT screen becomes a black rectangle, whereas if we set the threshold low, a white rectangle will appear, regardless of the true shape and color of the object in the scene.

3–3. Display on the CIE plane

At this point we must consider the color transformation that will convert the measured color vector (R_{ij}, G_{ij}, B_{ij}) into CIEs (X_{ij}, Y_{ij}, Z_{ij}) at each picture element. The transformation equation can, in general, be written as

$$\begin{bmatrix} X_{ij} \\ Y_{ij} \\ Z_{ij} \end{bmatrix} = \begin{bmatrix} a_{11}\, a_{12}\, a_{13} \\ a_{21}\, a_{22}\, a_{23} \\ a_{31}\, a_{32}\, a_{33} \end{bmatrix} \begin{bmatrix} R_{ij} \\ G_{ij} \\ B_{ij} \end{bmatrix}$$

where a_{k1} $(k, 1 = 1,2,3)$ is an element of the conversion matrix and must be determined experimentally.

Munsell's standard color cards, listed in Table I, were placed before the TV camera one by one; and red, green, and blue values were sampled into the processor (R, G, and B column of Table I). The x_{ij} and y_{ij} values of the color cards are read from the Japanese Industrial Standard (JIS Z8721–1958), which are listed in the *x* and *y* columns of Table I. If we write

$$x_{ij} = \frac{X_{ij}}{X_{ij} + Y_{ij} + Z_{ij}}$$
$$y_{ij} = \frac{Y_{ij}}{X_{ij} + Y_{ij} + Z_{ij}}$$

and

$$Y_{ij} = \frac{R_{ij} + G_{ij} + B_{ij}}{3}$$

(brightness assumption)

then the conversion matrix is determined by the least-squares method. The result is written as

$$\begin{bmatrix} \tilde{X}_{ij} \\ \tilde{Y}_{ij} \\ \tilde{Z}_{ij} \end{bmatrix} = \begin{bmatrix} 0.46 & -0.21 & 0.15 \\ -0.08 & 0.78 & -0.36 \\ -0.04 & -0.23 & 0.60 \end{bmatrix} \begin{bmatrix} R_{ij} \\ G_{ij} \\ B_{ij} \end{bmatrix}$$

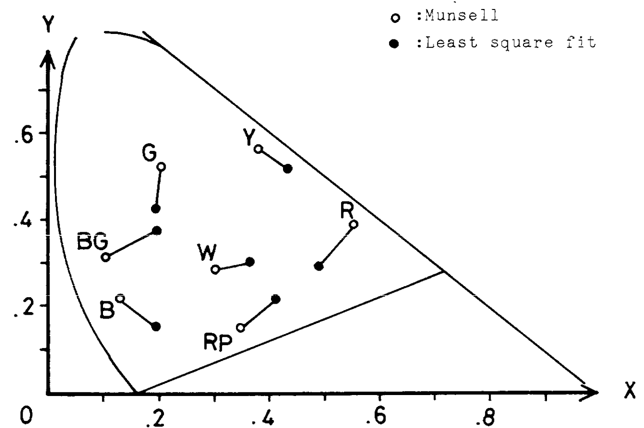
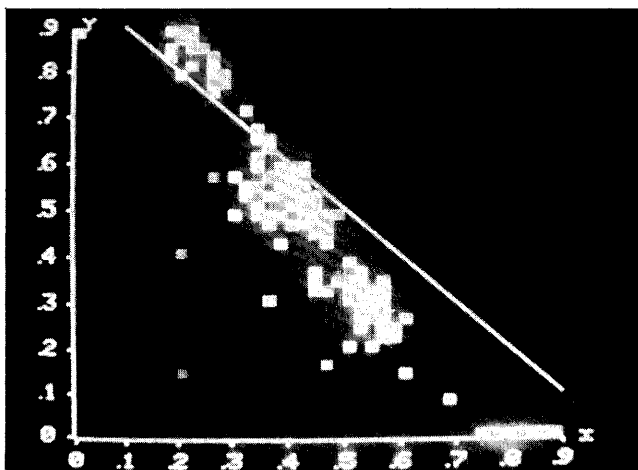
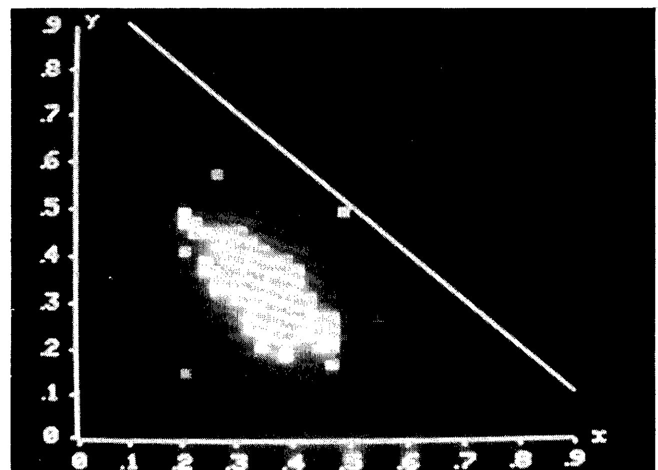


Figure 4—Deviation of the least-squares fitted points from their true points



(a)



(b)

Figure 5—An example of color distributions on the CIE's plane.
(a) color distribution of a signal region;
(b) color distribution of a background region.

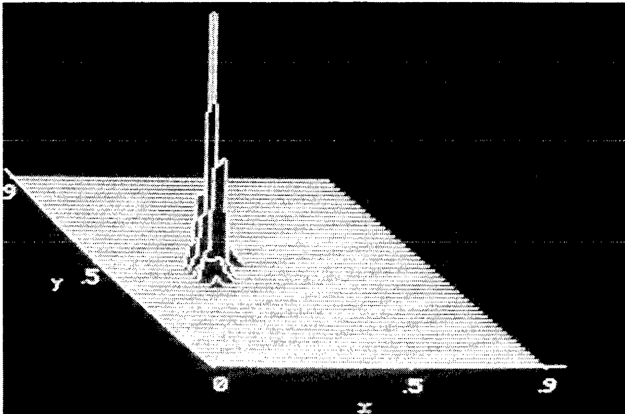


Figure 6—Display of a color distribution of an image in the CIE's coordinate system

whose transformational matrix is seriously affected by lighting condition. Because of the least-squares fitting, $(\hat{X}_{ij}, \hat{Y}_{ij}, \hat{Z}_{ij})$ do not lie exactly on (X_{ij}, Y_{ij}, Z_{ij}). Their corresponding position pairs are shown in Figure 4. Our control program can handle a picture in the CIE color coordinate system. A typical two of the provided commands are briefly described in the following:

1. A command to display a distribution on the color plane: Figure 5 shows the resulting display of this command. The sampled x_{ij} and y_{ij} in a picture are plotted in the CIE's color plane. The command form is

   $$>K(CR)$$

   The six standard color points (R, Y, G, B, P, W) are displayed at their location by their color.
2. A command to display the color frequencies: Let us define a color frequency as

   $$f_{k1} = \sum_{i=1}^{M}\sum_{j=1}^{N} \delta(x_{ij} - k)\,\delta(y_{ij} - 1)$$

   where $\delta(\alpha) = 1$, if $\alpha = 0$, and $\delta(\alpha) = 0$, if $\alpha \neq 0$. Its command form is

   $$>J(CR)$$

   The J command displays f_{k1} on the CRT screen, as shown in Figure 6.

4. ONE EXAMPLE OF APPLICATIONS

We want, in general, to design a software package that can cover a wider area, but no software can be designed without a concrete objective, especially in its infancy. Our motive for designing this software package will be briefly explained in the following paragraphs.

We are studying automatic guidance of an electrically driven vehicle. A TV camera sees guidelines on the floor. The microprocessor estimates the driving path and controls its trajectory. Experience has shown us that a monochromatic TV camera is not enough for recognizing objects in a real world: color seems to provide us with vital information.

If we are to place guiding signs on the floor, there emerge several fundamental questions, such as what color is best for a sign, what form is best, and where to place the signs.

Color distribution of a sign and its background are displayed on the color graphic display (see Figure 5), using a command of the package. The background distribution is rather concentrated in the center region of the CIE plane. The signal has a long, thin distribution. We can define a measure S that indicates separability of a signal from its background, as

$$S = (\bar{x}_s-\bar{x}_b)\left(\frac{1}{\sigma_{sy}}+\frac{1}{\sigma_{bx}}\right) + (\bar{y}_s-\bar{y}_b)\left(\frac{1}{\sigma_{sy}}+\frac{1}{\sigma_{by}}\right) + (\bar{Y}_s-\bar{Y}_b)\left(\frac{1}{\sigma_{sy}}+\frac{1}{\sigma_{by}}\right)$$

where subscripts s and b indicate signal and background, respectively, $\bar{}$ is an average operation, and σ is a standard deviation. If two distributions are well separated, then S takes a large value. On the contrary, if the two are mixed, S becomes small.

Now a thin tape is placed on the ground, and S is calculated. Then another tape is measured in the same manner. The tape with a larger S may be said to be more suitable for that background. Continuing like this, we can determine the best guiding signal for the specified background.

5. CONCLUSION

A new color picture processing system is introduced. It makes full use of recently advanced videotape recording technology. It gives mobility to a computer vision system. A software package for color picture processing has been coded and tested. It aims at an interactive processing of color image recorded on cassette tapes.

Although the commands now available in the control program cover only basic areas, they can be easily extended in any desired direction. If we consider the rapidly decreasing cost of microprocessors and videotape recorders, the proposed system may be said capable of being constructed at a very low price.

It is already concluded by the researchers that picture processing by monochromatic images has met a severe limitation in its real applications, especially in object recognition. Human processes color images. If we drop the color factor in picture processing, then the computer can never equal human capability. It is unreasonable to want the same results from image processing by computers as by human vision without the essential information provided by color.

But there are some problems in color picture processing. At least three times as much information must be stored in the memory as for black and white. The complexity of the resulting processing program will increase rapidly. The control program package described herein will serve as a core for color image processing and thus contribute to expanding computer power to a wider range of applications.

The importance and futility of device independence in computer graphics

by ANDERS VINBERG
ISSCO (Integrated Software Systems Corporation)
San Diego, California

ABSTRACT

Device independence in computer graphics refers to software that supports all graphics hardware devices. The economic reasons why device independence must be considered mandatory are reviewed in this paper. The ultimate futility of straightforward device independence, because of the widely differing characteristics of different devices, leads to a need for software that adapts intelligently to these characteristics—that has *device intelligence.* For good results we need not merely technical device intelligence, but also sensitivity to different applications. I coin the phrase *layout intelligence* for software that thus adapts the graph to the situation and show several examples.

WHAT IS DEVICE INDEPENDENCE?

A device-independent graphics system is one that works with all graphics output devices. Note the stringency here: not "all models of a major vendor," or "all devices compatible with some popular device," not "many" or even "most" graphics devices, but *all* graphics devices.

There are now commercially available graphics systems that offer device independence. It should be borne in mind that software products differ in degrees of support. Some are delivered and guaranteed with device interfaces; others require the user to develop and modify device interfaces. In software, as everywhere else, there is no free lunch; you get what you pay for.

What do we mean when we say that a graphics system "works with" all devices? Realistically, some changes will be needed when a new, previously unheard-of device is brought in. However, we can demand that these changes be kept from the end users. The graphics support staff should be able to modify the system, with minor effort, to support the device. But the instructions from the end users—the graph description, the source code, the English commands, the prompt responses, the touch or voice input—should produce a nearly identical graph on the new device with no change.

WHY IS DEVICE INDEPENDENCE IMPORTANT?

The hardware obsolescence argument

As with the stock market and the weather in England, the only thing that can be said with certainty about graphics hardware technology is that it will change in the future. Scores of hardware vendors bring better mousetraps to the market in a never-ending flow.

It should be remembered that although providing device independence is a technical problem, the motivation for requesting it is an economic one. The economics of the situation indicate that the important effects of a technology change are those experienced by a large number of people.

The effort of the support person is a cost, to be sure, and should be minimized; but it is the effort to adapt by all the end users (hundreds of end users, if graphics is a success) that may become prohibitively expensive and may prevent the organization from taking advantage of the new technology. Thus, by allowing end users to make a large investment in device-specific graph descriptions, an organization may paint itself into a corner and may end by being stuck with obsolete technology for a long time or facing an extremely costly conversion.

This means that when we specify that "the graph description should produce a nearly identical graph with no changes by the user," the phrase "no changes" is more important than "identical." If the system modifies the graph slightly, this may be acceptable as long as the meaning is retained. But forcing all users to modify all their old programs or graph descriptions is unacceptable.

This discussion has focused on one important reason for demanding device independence, which we can describe as "hardware obsolescence insurance." There is another reason, which has nothing to do with the future, but is still economic: previewing.

The previewing argument

Graphics CRTs are used for two kinds of applications: decision support graphics and preview of presentation graphics. Unlike decision support graphics, presentation graphics and report graphics rarely use CRTs as the final output medium; slides, overheads, and above all paper hard copy dominate. But, because of the low speed and high cost of most hard-copy devices, CRTs are preferred during the design phase, when several graph forms are being tried out to select the most effective.

For this preview work, the most important requirements are that there be absolutely no changes to the graph descriptions and minimal changes to the graph. The preview must be a faithful reproduction of the final result, even if this means that it does not use well the characteristics of the previewing device.

In summary, device independence is of critical importance, because most users will want to use several devices today and all users will want to be able to use new devices tomorrow. Since the software knowledge is now available, and since most graphics software today offers device independence at some level of support, any investment of money, effort, or training in device-dependent software today is indefensible and is surely the most fundamental mistake that can be made when moving into graphics.

WHY IS DEVICE INDEPENDENCE ULTIMATELY FUTILE?

Now for the bad news. Graphics output devices are sufficiently different that a graph that looks good on one device may not look good on another.

Note that we introduced a new concept here: what looks good. Previously we have talked about what can be done—"Can the software system produce the same graph on all devices?" Now we question what should be done.

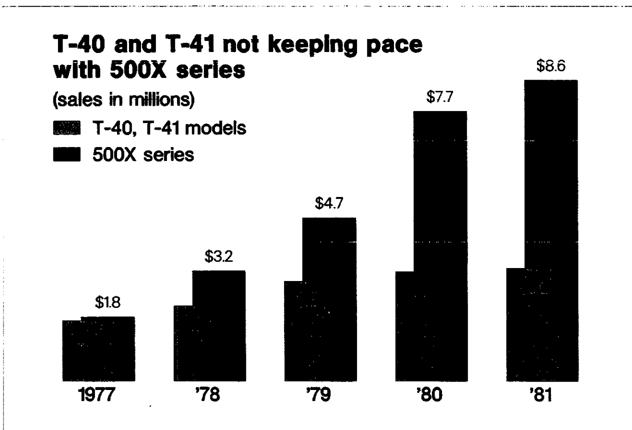


Figure 1a—Graph well adapted to slide presentation (color simulated with grayscale)

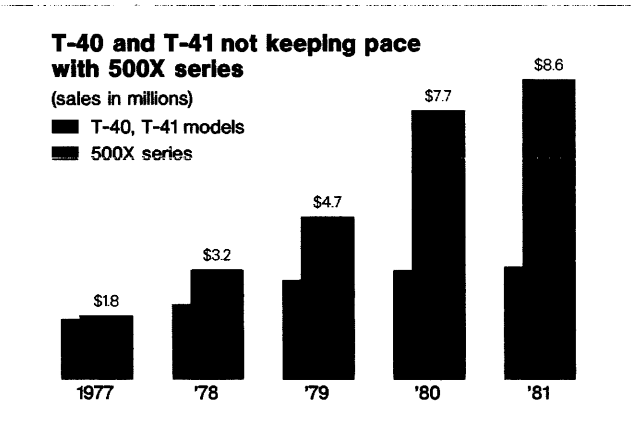


Figure 1b—Slide-adapted graph produced with black-and-white device: illegible results because of hardware mismatch

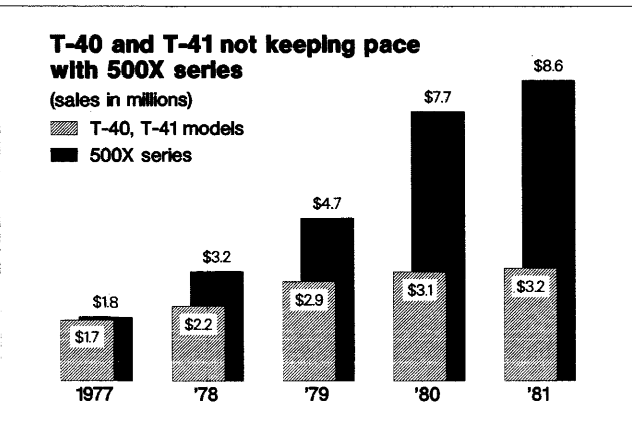


Figure 1c—Replacement of solid-color fields with crosshatching: chart legible, but still not suitable for use in a report

A basic example is illustrated in Figure 1. Figure 1a shows a very well-designed graph for use as a color slide or a color viewgraph. (Color is represented here as grayscale.) If this chart is copied onto a black-and-white device for reproduction as a report or handout, or if the color viewgraph is simply placed in a monochrome copier, Figure 1b results: a worthless chart, where the two data sets are indistinguishable, but very common. The minimum requirements for a black-and-white paper copy is that shade patterns are used to distinguish the data sets, as in Figure 1c. But the chart, which was designed for projection and hence viewing at a distance, looks amateurish and childlike when copied on an 8½-inch-by-11-inch sheet of paper, viewed at a distance of about 10 inches. Figure 1d shows a better version of the chart—vertical page orientation, different typefaces, smaller annotation, more annotation. It is the same chart, but different. The chart has been tailored for two different applications, embodied by two different output devices.

Note that the term *output device* may mean more than simply the graph production device. A color viewgraph and a black-and-white report illustration may both be produced on the same device, say one of the many desktop color pen plotters available in the $5,000- to $10,000-range today. The device that differs in this case is not the production device, but the presentation or reproduction device: an overhead projector versus a copying machine.

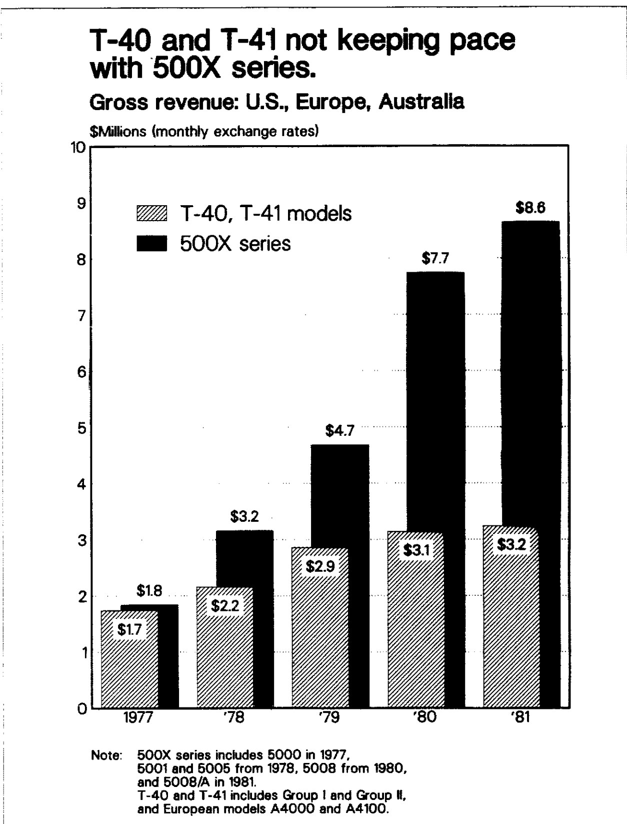


Figure 1d—Extensive redesign of chart to make it suitable for report use

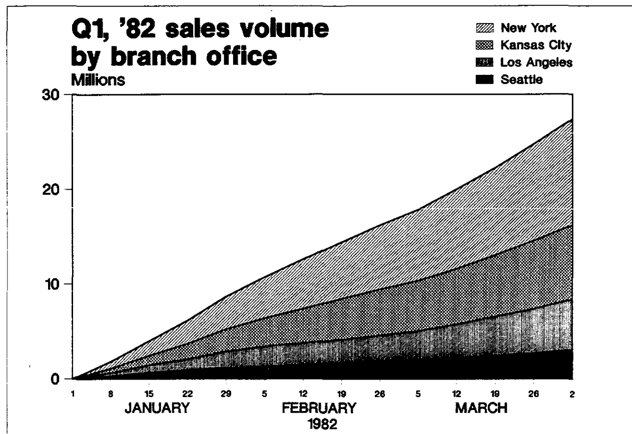


Figure 2a—Slide layout (color simulated with grayscale)

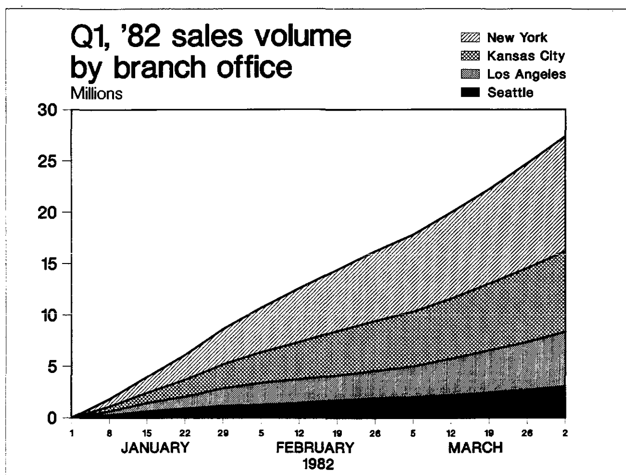


Figure 2b—Vugraph layout (color simulated with grayscale)

The differing requirements of color and monochrome are obvious; the differing requirements of hard copy and projection are equally important but less generally recognized. Many of today's output devices have idiosyncrasies that pose other difficulties, some obvious, others deeply technical.

The low-resolution CRT devices so common today require simple, large annotation without frills to be legible. Some are also cell-oriented, able to place text only in some locations on the screen; for these devices, graphics elements must be adjusted to fit the text. Text may be available only in some sizes or orientations. All these need adaptation by the software.

Other differences to be accommodated concern what happens when two graphics items occupy the same location: does one hide the other, do both shine through, do colors mix, and if so how—or do you get a smear?

All this, and much more, means that device independence is not the answer. What is needed is what I call device intelligence.

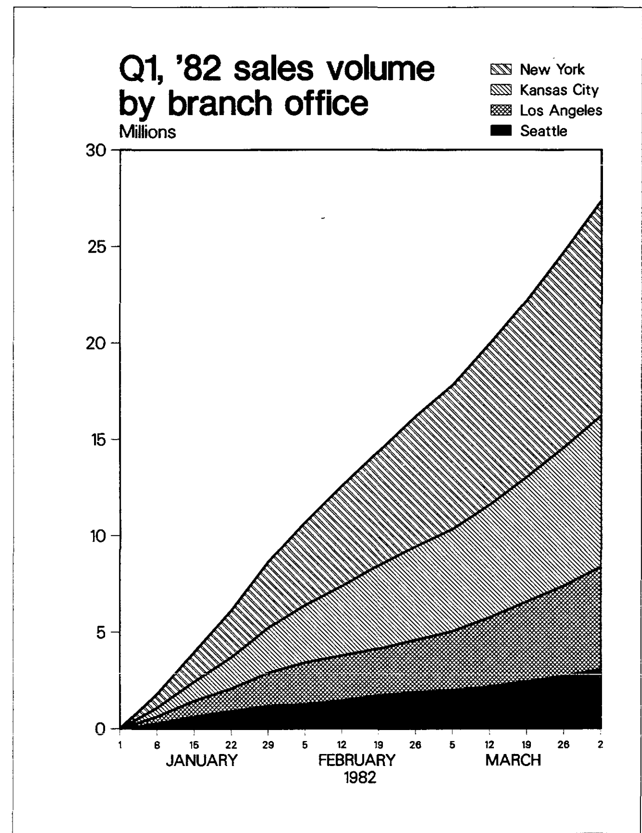


Figure 2c—Vertical report layout (note space for binding along left edge)

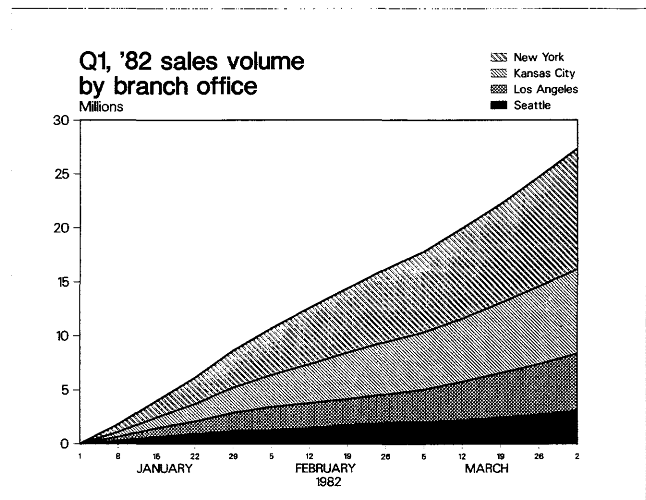


Figure 2d—Horizontal report layout (note space for binding along top edge)

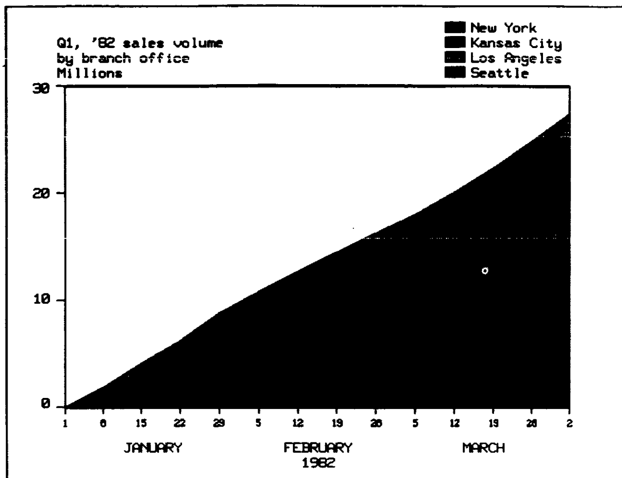


Figure 2e—Color CRT layout (color simulated with grayscale)

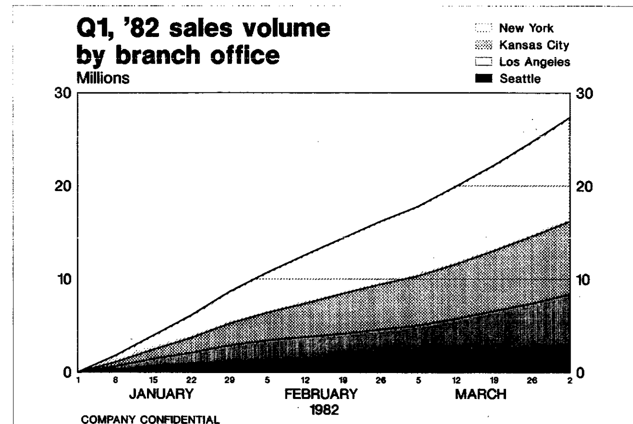


Figure 3a—Locally modified slide layout (color simulated with grayscale)

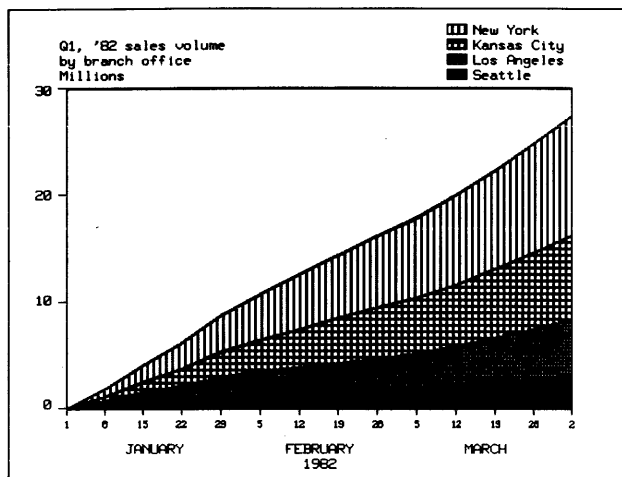


Figure 2f—Monochrome CRT layout

GRAPHICAL VERSUS TECHNICAL DEVICE INTELLIGENCE

Device intelligence means that the software adapts to the many peculiarities of the output device.

The phrase is occasionally used in a narrow technological sense, referring merely to using the varying capabilities of the device. If the device can draw higher-level constructs, such as a circle, rectangle, dashed line, character, conic segment, or axis, or if it can fill in an area of certain shape, these functions can be offloaded from the host computer, and, more important, from the communications line. This means that if the software knows and adapts to the capabilities of the device, the graph is drawn faster. However, device intelligence in this technical sense does not mean that the graph is good or even meaningful; it just means that the same ugly graph is drawn quicker.

Speed is certainly important, and technical device intelligence is a valuable first step (and one not trivially achieved: the complexities of the software needed to use fully all device functions, and emulate them fully when not present, are significant). But to achieve good results we need graphical device intelligence. This means that the graph layout is adapted to the device characteristics: page orientation, annotation style, amount and size, and data set identification, are all affected. Therefore I will refer to it as *layout intelligence.*

We remember from the discussion of the many possible uses for the common pen plotter that the choice of layout must not be determined only by the choice of production device. The intended application or intended reproduction device also affect the ideal layout choice. When using a CRT device to design and preview a graph for eventual production and use as

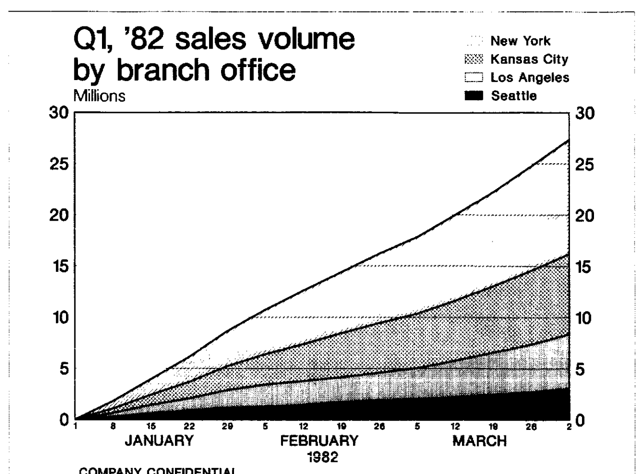


Figure 3b—Locally modified vugraph layout (color simulated with grayscale)

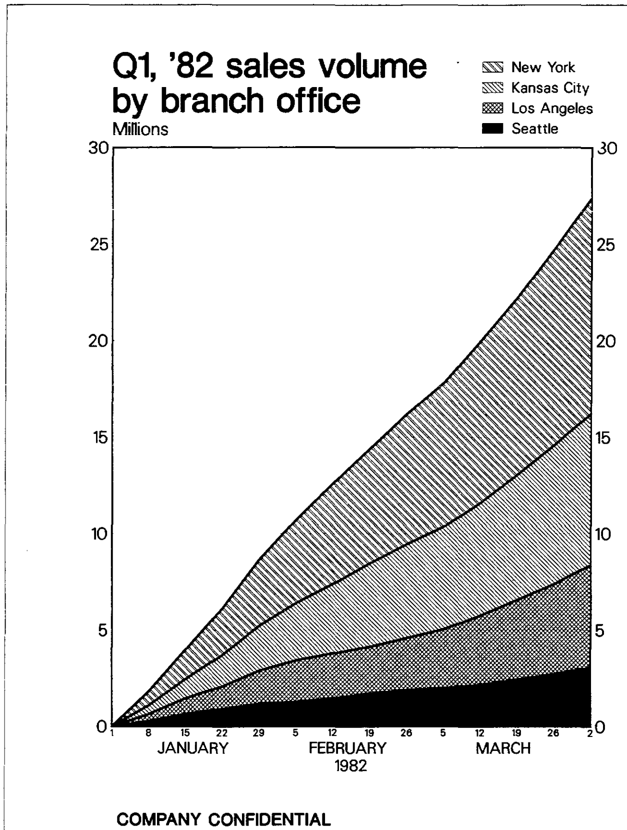


Figure 3c—Locally modified vertical report layout (note space for binding along left edge)

a slide, we certainly want to see the slide layout faithfully rendered, even if the limitations of the CRT device may make some details of the chart ugly or even illegible. Thus, in this situation, "improvements" of the graph to make it fit the CRT would be detrimental. The layout intelligence must be driven both by device choice and explicit specification of intended use, desired use of colors, and other relevant factors.

In an upcoming version of ISSCO's TELL-A-GRAF system, for example, these choices may all be made automatically with minimal specification. Let us look at an example.

We have prepared a format description to allow TELL-A-GRAF to read data from the COBOL files of the accounting programs and to select interesting data from it. We specify what information we want, and how we want to see it:

DATA FILE IS "ACC1Q82".
DATA FORMAT IS "SALES BY OFFICE".
GENERATE A DATE AREA CHART.

If we now specify

LAYOUT IS SLIDE.
DEVICE IS DICOMED MODEL D148C.

we get the slide shown in Figure 2a.

If we specify

LAYOUT IS VUGRAPH.
DEVICE IS HP MODEL 7221.

we get the viewgraph shown in Figure 2b.

If we specify

LAYOUT IS REPORT.
DEVICE IS COMP80.

we get the chart shown in Figure 2c. If we prefer a horizontal ("landscape") page orientation for our report illustration, we can specify

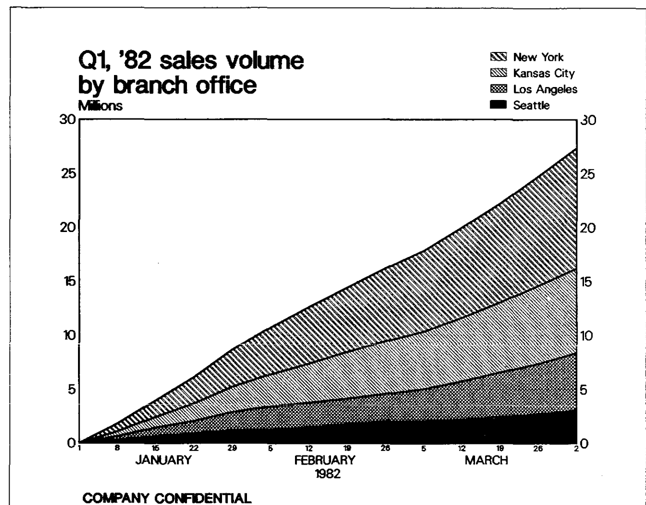


Figure 3d—Locally modified horizontal report layout (note space for binding along top edge)

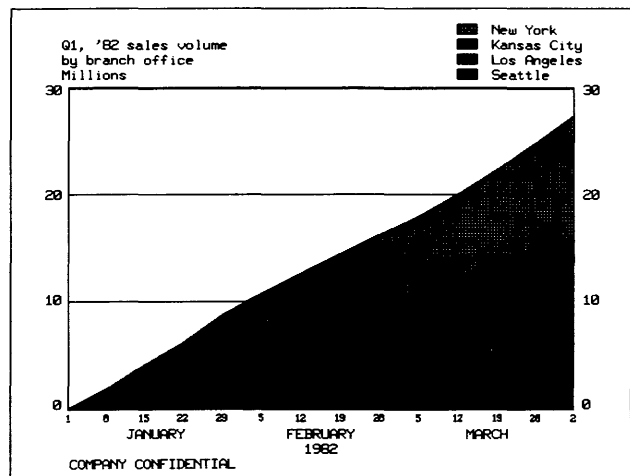


Figure 3e—Locally modified color CRT layout (color simulated)

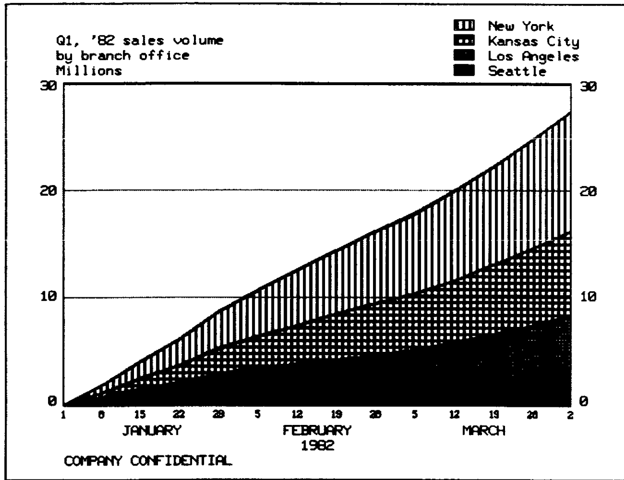


Figure 3f—Locally modified monochrome CRT layout

LAYOUT IS HORIZONTAL REPORT.

to get the chart in Figure 2d.

If we do not want a chart optimized for showing on a CRT screen, we specify

LAYOUT IS CRT.
DEVICE IS TEKTRONIX MODEL 4027.

and get the graph shown in Figure 2e. Note that if the device is monochrome, the layout automatically adapts to being sensible. With this specification:

LAYOUT IS CRT.
DEVICE IS TEKTRONIC MODEL 4025.

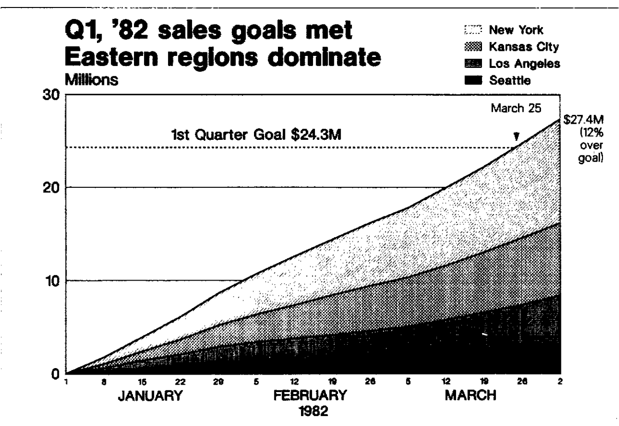


Figure 4—Slide customized for a particular message (color simulated with grayscale)

we would get the graph illustrated in Figure 2f, since the system knows that the 4025 model has no color.

All the variants shown represent the same data and are basically the same chart. Each variant is optimized for its intended use.

This application intelligence allows even the casual user to get a really good graph by simply making five choices:

1. Which data file to use ("DATA FILE" statement)
2. Which information to retrieve from this data file ("DATA FORMAT" statement)
3. Which chart type to use ("GENERATE" statement)
4. Which layout to use ("LAYOUT" statement)
5. Which device to use ("DEVICE" statement)

FLEXIBILITY RETAINED

An important point must be added: all this automation is achieved without sacrificing flexibility and control over details. For example, assume that the organization has its own graphics standards that state that all area charts must have the following:

- Tick marks pointing inward on the horizontal axis
- Horizontal grid lines, no tick marks on the vertical axis, and a double vertical axis
- The words COMPANY CONFIDENTIAL in the lower left corner

These local standards can be entered into the system once and for all and will then apply to all subsequent area charts generated (unless explicitly overridden, of course). To specify these local standards, one enters:

GENERATE AN AREA CHART
X TICKS REVERSED.
Y GRID, NO TICKS, DOUBLE AXIS.
COMMENT "COMPANY CONFIDENTIAL".
STORE DEFAULTS.

If these standards had been stored, the six charts shown would have looked like the ones shown in Figure 3a through 3f. Even so, the individual user can still customize the chart to make a special point, as shown in Figure 4.

Thus, while preserving the basic flexibility of the software system, the layout intelligence gives the casual user the benefit of the accumulated experience of the graphics experts involved in the design of these application-specific layouts. If you are an expert in something else, with no desire to become a graphics expert, this layout intelligence should prove a great boon through making the use of graphics more effective without requiring you to reinvent the wheel of good graphics.

Optimal three-dimensional flight control of a supersonic fighter

by CHING-FANG LIN and KHAI LI HSU
The University of Michigan
Ann Arbor, Michigan

ABSTRACT

This paper discusses optimal three-dimensional flight control of a supersonic fighter through an onboard automatic guidance and control system. This system is simulated to check the speed requirements of the algorithms to be solved before being implemented in real hardware. A very-high-speed digital computer is used for this time-critical simulation. In the simulation optimal trajectories are generated in real time, on line. Results obtained from the particular problem of a real-time, online minimum-time supersonic chandelle with prescribed final point are displayed.

INTRODUCTION

This paper discusses optimal three-dimensional flight control of a supersonic fighter through an onboard automatic guidance and control system. The essential portion of this system consists of a fast computer system called the mission computer and another fast computer system called the flight control computer. The function of the mission computer is to generate real-time, online optimal trajectories; the function of the flight control computer is to track the trajectories generated by the mission computer. The automatic guidance and control system is simulated to check the speed requirements of the algorithms to be solved before implementing it in real hardware. Naturally the simulation is very time-critical and therefore requires a very-high-speed computer to perform the simulation. In this paper, the computer used for this simulation is a very-high-speed special-purpose digital computer designed specifically for time-critical, continuous system simulation tasks.

Simulation of real-time, online optimal trajectories of supersonic flight has been discussed by Lin.[1,2] In these references, for each maneuver a family of trajectories is precalculated off line and stored in the mission computer. These trajectories are then generated in real time, on line, by the mission computer by table lookup. They are then checked by computer simulation to see if they can be adopted in real-time, online computation. If so, the flight control computer will track these optimal trajectories generated by the mission computer. However, a substantial computer memory space is required to store all these precalculated trajectories; therefore this approach creates a problem, since the mission computer has a limited memory space. To avoid this problem, optimal trajectories in this paper are generated entirely in real time, on line, without table lookup. The particular problem of a real-time, online minimum-time supersonic chandelle with prescribed final point is used as an example in this paper. Several techniques are used to obtain real-time, online optimal trajectories of this maneuver. They include (1) modeling of the aerodynamic and engine characteristics of a typical lightweight, high-thrust-to-weight ratio supersonic fighter; (2) introduction of a set of dimensionless variables, which leads to general results for a whole class of vehicles having similar physical characteristics; (3) general properties of optimal trajectories; and (4) use of the switching theory.[3]

THEORETICAL ANALYSIS

If the thrust is considered as nearly aligned with the velocity vector $\vec{V}$, then the motion of a point mass lifting vehicle over a flat nonrotating earth, with the assumption of symmetrical flight, is governed by the following set of nonlinear ordinary differential equations.

$$\dot{X} = V\cos\gamma\cos\psi \qquad \dot{V} = g\left(\frac{T-D}{W} - \sin\gamma\right)$$
$$\dot{Y} = V\cos\gamma\sin\psi \qquad \dot{\gamma} = \frac{g}{V}\left(\frac{L\cos\phi}{W} - \cos\gamma\right)$$
$$\dot{Z} = V\sin\gamma \qquad \dot{\psi} = \frac{gL\sin\phi}{VW\cos\gamma} \tag{1}$$

In these equations, the position vector is composed of the longitudinal range X, the lateral range Y, and the altitude Z; and the velocity vector is composed of the speed V, the flight path angle γ, and the heading ψ. For turning flight, during a relatively short interval, we can neglect the mass flow equation and consider the weight as practically constant. The acceleration of the gravity g is assumed to be constant. The aerodynamic and propulsive forces are determined by the following relations

$$L = \tfrac{1}{2}\rho(Z)V^2SC_L \tag{2}$$
$$D = \tfrac{1}{2}\rho(Z)V^2SC_D \tag{3}$$
$$T = \zeta T_{\max}(Z, M) \tag{4}$$

where $\rho(Z)$ is the atmospheric density given as a tabular function, and

$$0 \le \zeta \le 1 \tag{5}$$

represents the thrust control parameter. For a parabolic drag polar, as function of the Mach number, we have

$$C_D = C_{D_0}(M) + K(M)C_L{}^2 \tag{6}$$

where the zero-lift drag coefficient, C_{D_0}, and the induced drag coefficient, K, are functions of the Mach number. Because of the lift-drag relation, the flight is controlled by the lift coefficient C_L which is equivalent to the angle of attack α, the bank angle ϕ, and the thrust magnitude T.

The Hamiltonian to the variation problem is

$$H = P_XV\cos\gamma\cos\psi + P_YV\cos\gamma\sin\psi + P_ZV\sin\gamma + P_Vg\left(\frac{T-D}{W} - \sin\gamma\right) + P_\gamma\frac{g}{V}\left(\frac{L\cos\phi}{W} - \cos\gamma\right) + P_\psi\frac{gL\sin\psi}{VW\cos\gamma} \tag{7}$$

It is known that the problem has the integrals[3]

$$H = C_0,\ P_X = C_1,\ P_Y = C_2,\ P_\psi = C_1Y - C_2X + C_3 \tag{8}$$

The problems considered here are minimum time problems. Hence, for maximization of the Hamiltonian, $C_0 > 0$. By using the Hamiltonian integral only two of the three remaining adjoint variables P_Z, P_V and P_γ need to be found. In vertical flight all three variables are involved, hence their presence imposes a difficulty in solving the optimization problem, the

same difficulty encountered in three-dimensional flight problems. As compared to the vertical flight, three-dimensional flight has two more variables, i.e., the lateral range Y and the heading ψ, which may pose a problem in obtaining the optimal solution; however, the adjoint variables P_Y and P_ψ associated with these variables have been found in Eqs. (8). In the case of horizontal flight, the adjoint variables P_Z and P_γ are not present, and the remaining adjoint variable P_V is given by the Hamiltonian integral. The problem is completely solved in the book by Lin.[3]

Since the aerodynamic and the engine characteristics are functions of the Mach number, it is convenient to use the following dimensionless variables:

$$M = V/a(Z), \quad \omega = 2W/kp(Z)S \tag{9}$$

where a(Z) is the speed of sound and p(Z) is the ambient pressure. Both a(Z) and p(Z) are given as tabular functions; and ω is the dimensionless wing loading, which is a function of the ambient pressure p, and is also a variable representing the altitude. For convenience of notation, we define

$$\begin{aligned} C_L^* &= \sqrt{C_{D_0}/K}, \qquad E^* = \tfrac{1}{2}\sqrt{KC_{D_0}} \\ \lambda &= C_L/C_L^*, \qquad \tau_{max}(M, \omega) = T_{max}/W \end{aligned} \tag{10}$$

where E^* is the maximum lift-to-drag ratio, which is a performance characteristic; and λ is the normalized lift coefficient, which can be considered a control variable for the lift control. The normalized lift coefficient is rescaled so that when $\lambda = 1$, the lift coefficient is equal to the lift coefficient C_L^* for maximum lift-to-drag ratio. Note both $E^*(M)$ and $C_L^*(M)$ are functions of the Mach number and that the maximum thrust-to-weight ratio $\tau_{max}(M, \omega)$ is a function of the Mach number as well as ω. The lift control is bounded by an upper limit $\lambda_{max}(M)$ that corresponds to $C_{L_{max}}(M)$. It is assumed that $C_{D_0}(M)$, $K(M)$, $\tau_{max}(M, \omega)$ and $C_{L_{max}}(M)$ are known functions of the Mach number. For numerical computation, we use data of a supersonic fighter assembled in Lin,[3] but the same procedure applies to any other set of data. The three-dimensional turning flight is a difficult maneuver; hence we use the load factor n as a lift control variable to represent the angle of attack. The load factor n is equal to the dimensionless life force ℓ defined as

$$n = \ell = L/W = M^2C_L/\omega = \Delta\lambda \tag{11}$$

where

$$\Delta = M^2C_L^*/\omega \tag{12}$$

The load factor n can also be used as a control variable to replace the normalized lift coefficient λ, and it is an important parameter that can limit the flight domain. From Eq. (11), since $\lambda \leq \lambda_{max}(M)$, the flight domain is bounded by the curve

$$n = \Delta\lambda_{max}(M) \tag{13}$$

Hence, the load factor n is subject to the constraints

$$|n| \leq n_{max} = \inf.\left[n_s, \frac{M^2C_{L_{max}}(M)}{\omega}\right] \tag{14}$$

where the constant value n_s is a physiological/structural constraint and n_{max} is the maximum permissible load factor, either for $C_L = C_{L_{max}}$ or $n = n_s$. This domain of flight may be further restricted by the line of maximum dynamic pressure

$$M^2 \leq \frac{2q_{max}}{kp} = \frac{S}{W}\omega\, q_{max} \tag{15}$$

and the line of maximum Mach number obtained by solving the equation $dV/dt = 0$ with $\tau = \tau_{max}$ such that

$$\tau_{max} = \frac{1}{2E^*}\left(\Delta + \frac{n^2}{\Delta}\right) + \sin\gamma \tag{16}$$

By using the notations introduced thus far, the Hamiltonian (7) becomes

$$\begin{aligned} H = {} & C_1V\cos\gamma\cos\psi + C_2V\cos\gamma\sin\psi + P_ZV\sin\gamma \\ & + P_Vg\left[\zeta\tau_{max}(M, \omega) - \frac{1}{2E^*}\left(\Delta + \frac{n^2}{\Delta}\right) - \sin\gamma\right] \\ & + P_\gamma\frac{g}{V}(n\cos\phi - \cos\gamma) + P_\psi\frac{g\,n\sin\phi}{V\cos\gamma} \end{aligned} \tag{17}$$

In this formulation, the control variables are the thrust parameter ζ, the bank angle ϕ, and the load factor n. The thrust parameter ζ and the load factor n are subject to constraints (5) and (14) respectively, and the bank angle ϕ is subject to the constraint $|\phi| \leq \phi_{max}$. The ϕ_{max} can be either a constant or a function of the state variables, depending on the problem considered.

Regarding the thrust control, we consider the adjoint P_V, called the switching function. Then to maximize the Hamiltonian, if

$$\begin{aligned} &P_V > 0, \quad \zeta = 1 && \bullet\text{Boost arc } (B \text{ arc}) \\ &P_V < 0, \quad \zeta = 0 && \bullet\text{Coast arc } (C \text{ arc}) \\ &P_V \equiv 0 \text{ for } t\,\varepsilon\,[t_1, t_2] && \bullet\text{Sustained arc } (S \text{ arc}) \\ &\qquad \zeta = \text{variable} \end{aligned} \tag{18}$$

The optimum trajectory is a combination of boost arc (B arc), coast arc (C arc), and sustained arc (S arc). At the junction of the different thrust control arcs, $P_V = 0$. For a junction between arcs, a C-B sequence is optimum if at the junction $dP_V/dt > 0$. For a reverse condition, a B-C sequence is optimum.

The aerodynamic control consists of the bank angle ϕ and the load factor n. The optimal aerodynamic control can be obtained by using the technique of the domain of maneuverability presented by Lin.[3] First, whenever interior bank angle and interior load factor are used, we have

$$\tan\phi = \frac{P_\psi}{P_\gamma\cos\gamma} \tag{19}$$

$$n^2 = \frac{\Delta^2E^{*2}}{V^2P_V^2}\left(P_\gamma^2 + \frac{P_\psi^2}{\cos^2\gamma}\right) \tag{20}$$

By applying Eq. (19) to Eq. (20), we have

$$n = \Delta E^*P_\gamma/V P_V\cos\phi \tag{21}$$

If all P_V, P_γ, and P_ψ approach zero simultaneously, the in-

determination of ϕ and n can be resolved by applying L'Hopital's rule, which leads to

$$\tan\phi = \frac{A}{B} \tag{22}$$

$$n = \frac{\Delta E^* B}{C_0 \cos\phi \sin\gamma} \tag{23}$$

where

$$A = V\sin\gamma\,(C_2\cos\psi - C_1\sin\psi)$$
$$B = C_0\cos\gamma - C_1 V\cos\psi - C_2 V\sin\psi \tag{24}$$

and ϕ in Eq. (23) is obtained from Eq. (22). Second, whenever interior bank angle and boundary load factor are used, i.e., when the load factor is on its upper boundary as given by condition (14), the bank angle remains an interior bank angle as given by Eq. (19). Third, whenever boundary bank angle and interior load factor are used, i.e., when the bank angle is on the boundary $\phi = \phi_{max}$, the corresponding interior load factor is given as

$$n = \frac{\Delta E^*}{VP_V}\left(P_\gamma\cos\phi_{max} + \frac{P_\psi\sin\phi_{max}}{\cos\gamma}\right) \tag{25}$$

If all P_V, P_γ, and P_ψ approach zero simultaneously, the indetermination of n can be resolved by applying L'Hopital's rule, which leads to

$$n = \frac{\Delta E^*}{C_0\sin\gamma}(A\sin\phi_{max} + B\cos\phi_{max}) \tag{26}$$

where A and B are given in Eqs. (24). Fourth, whenever boundary bank angle and boundary load factor are used, the bank angle is on the boundary $\phi = \phi_{max}$, and the load factor is on its upper boundary, as given by condition (14).

An important condition as seen in the technique of the domain of maneuverability is that the interior load factor be used with B arc. In particular, while B arc can be flown with interior or boundary load factor, C arc and possibly S arc can be flown with only boundary load factor. From the above four arcs of the aerodynamic control, it is seen that the optimal aerodynamic controls are functions of the adjoint variables P_V, P_γ, and P_ψ associated with the velocity vector. Since P_ψ is known, the adjoint variables remaining to be found are P_V and P_γ. Their differential equations are coupled with the equation of P_Z. With the existing integrals (8), one of the three adjoint equations of P_V, P_γ, and P_Z can be deleted. It is found that in order to save substantial computation time and to enhance real-time, online optimization, it is simpler to integrate the adjoint equations of P_V and P_γ and obtain the adjoint variable P_Z from the existing integrals (8). From the Hamiltonian (17) we deduce the adjoint equations of P_V and P_γ. For interior bank angle and interior load factor

$$\begin{aligned}\frac{dVP_V}{dt} = & -C_0 + P_V g\left\{\zeta\tau_{max}(2-\tau_{max_M}) - 2\sin\gamma + \frac{1}{2E^*}\left[\left(\Delta - \frac{n^2}{\Delta}\right)C_L{}^*{}_M - \left(\Delta + \frac{n^2}{\Delta}\right)E^*{}_M - \frac{4n^2}{\Delta}\right]\right\}\\ & + P_\gamma\frac{2g}{V}(n\cos\phi - \cos\gamma) + P_\psi\frac{2g\,n\sin\phi}{V\cos\gamma}\\ \frac{dP_\gamma}{dt} = & \; C_1 V\sin\gamma\cos\psi + C_2 V\sin\gamma\sin\psi - P_Z V\cos\gamma + P_V g\cos\gamma\\ & - P_\gamma\frac{g\sin\gamma}{V} - P_\psi\frac{g\,n\sin\phi\sin\gamma}{V\cos^2\gamma}\end{aligned} \tag{27}$$

In the above equations the Hamiltonian integral has been used for simplification, and the subscript M is defined as

$$y_M = \frac{d\log y}{d\log M} = \frac{M}{y}\frac{dy}{dM} \tag{28}$$

In Eqs. (27), $\zeta = 1$, ϕ is given in Eq. (19), and n is given in Eq. (21). For interior bank angle and boundary load factor, if $n = n_s$, the adjoint equations are given in Eqs. (27) with $n = n_s$ and ϕ is given in Eq. (19). If

$$n = M^2 C_{L_{max}}/\omega$$

$$\begin{aligned}\frac{dVP_V}{dt} = & -C_0 + P_V g\left\{\zeta\tau_{max}(2-\tau_{max_M}) - 2\sin\gamma + \frac{1}{2E^*}\left[\left(\Delta - \frac{n^2}{\Delta}\right)C_L{}^*{}_M - \left(\Delta + \frac{n^2}{\Delta}\right)E^*{}_M + \frac{2n^2}{\Delta}C_{L_{maxM}}\right]\right\}\\ & - P_\gamma\frac{g}{V}\left(2\cos\gamma + n\cos\phi\, C_{L_{maxM}}\right) - P_\psi\frac{g\,n\sin\phi}{V\cos\gamma}C_{L_{maxM}}\\ \frac{dP_\gamma}{dt} = & \; C_1 V\sin\gamma\cos\psi + C_2 V\sin\gamma\sin\psi - P_Z V\cos\gamma\\ & + P_V g\cos\gamma - P_\gamma\frac{g\sin\gamma}{V} - P_\psi\frac{g\,n\sin\phi\sin\gamma}{V\cos^2\gamma}\end{aligned} \tag{29}$$

In the above equations,

$$n = M^2 C_{L_{max}}/\omega$$

and ϕ is given in Eq. (19). For boundary bank angle and interior load factor, if ϕ_{max} is a constant, the adjoint equations are given in Eqs. (27) with $\zeta = 1$ and $\phi = \phi_{max}$, and n is given in Eq. (25). For boundary bank angle and boundary load factor, if $n = n_s$, then for a constant ϕ_{max} the adjoint equations are given in Eqs. (27), with $\phi = \phi_{max}$ and $n = n_s$. If

$$n = M^2 C_{L_{max}}/\omega$$

then for a constant ϕ_{max} the adjoint equations are given in Eqs. (29), with $\phi = \phi_{max}$ and $n = M^2 C_{L_{max}}/\omega$.

In the most general case, the solution requires the estimate of five parameters C_1, C_2, C_3 and the initial adjoint variables P_{V_0} and $P_{\gamma 0}$. This, coupled with the optimal switching from one control regime to another, constitutes the main difficulty of the problem. Success in obtaining the solution depends on the knowledge of the particular flight program considered. A very-high-speed digital computer is used for the computation. The total computing time for a single pass through the entire equations for optimal flight in three dimensions is approximately 457.5 microseconds. This shows that integration frame rate of up to 2186 per second can be accomplished.

The computation of the general minimum time problem is greatly simplified by ruling out the sustained arc, which is not likely to be involved. A partial proof of the nonoptimality of this singular arc is as follows: The singular condition is characterized by the contition $P_V = 0$, $P_V' = 0$ for a finite time interval. As seen in Eqs. (20) and (25), this will require that n be on the boundary, unless $P_\gamma = 0$ and $P_\psi = 0$ too. But this will be ruled out as follows: Along a sustained arc, the derivative of the equation $P_V = 0$ is taken, using the Hamiltonian integral under the singular condition

$$\frac{dP_V}{dt}=\frac{1}{V}\left\{-C_0+P_\gamma\left[\frac{2g}{V}(n\cos\phi-\cos\gamma)-g\cos\phi\frac{\partial n}{\partial V}\right]\right.$$
$$\left.+P_\psi\left(\frac{2gn\sin\phi}{V\cos\gamma}-\frac{g\sin\phi}{\cos\gamma}\frac{\partial n}{\partial V}\right)\right\}=0 \quad (30)$$

Thus from this equation it is clear that $P_\gamma \neq 0$ and $P_\psi \neq 0$.

If

$$n=n_s,$$

then

$$\partial n/\partial V=0$$

and Eq. (30) becomes

$$\frac{dP_V}{dt}=\frac{1}{V}\left[-C_0+P_\gamma\frac{2g}{V}\left(n_{max}\cos\phi-\cos\gamma\right)\right.$$
$$\left.+P_\psi\frac{2gn_{max}\sin\phi}{V\cos\gamma}\right]=0 \quad (31)$$

The variable thrust along a sustained arc is obtained by taking the derivative of this equation using the available singular conditions

$$\frac{d^2P_V}{dt^2}=-C_0\frac{g\tau_{max}}{V^2}\zeta+(\ldots)=0 \quad (32)$$

where the term in parentheses is a function of the state variables and constants of integration. Since the order of the singular arc is $q=1$, then according to the generalized Lengendre-Clebsch condition, a necessary condition for the optimality condition of the singular arc is that

$$C_0\frac{g\tau_{max}}{V^2}\leq 0 \quad (33)$$

This condition is not satisfied with $C_0>0$. If $n=M^2C_{L_{max}}/\omega$, Eq. (30) becomes

$$\frac{dP_V}{dt}=-\frac{1}{V}\left[C_0+P_\gamma\frac{g}{V}(2\cos\gamma+n\cos\phi\, C_{L_{max_M}})\right.$$
$$\left.+P_\psi\frac{gn\sin\phi}{V\cos\gamma}C_{L_{max_M}}\right]=0 \quad (34)$$

By taking the derivative of Eq. (34) using the available singular conditions, the equation for the intermediate thrust control is obtained in the form

$$\frac{d^2P_V}{dt^2}=A\zeta\tau_{max}+(\ldots)=0 \quad (35)$$

where

$$A=-\frac{g}{V^2}\left\{C_0+C_{L_{max_M}}\left(P_\gamma\frac{gn\cos\phi}{V}+P_\psi\frac{gn\sin\phi}{V\cos\gamma}\right)\right.$$
$$\left.\left[2+C_{L_{max_M}}+\left(C_{L_{max_M}}\right)_M\right]\right\} \quad (36)$$

and the term in parentheses in Eq. (35) is a function of the state variables and constants of integration. According to the generalized Lengendre-Clebsch condition for the optimality of the singular arc, $A\geq 0$. If $C_{L_{max}}$ is independent of the Mach number, the condition $A\geq 0$ is not satisfied. This is particularly true for the case of maneuver at low Mach number. For any prescribed function $C_{L_{max}}(M)$, the condition $A\geq 0$ defines a small region in the state variable and adjoint variable space where singular arc can be optimal.

With only *B* arc and *C* arc involved, the optimal thrust control is a combination of *B* arc and *C* arc. At the junction of a *B* arc and a *C* arc, $P_V=0$. For continuity of the load factor this occurs either when $n=n_s$ or $n=M^2C_{L_{max}}/\omega$, or $P_\gamma=0$ and $P_\psi=0$. If all P_V, P_γ, and P_ψ approach zero simultaneously, and the interior bank angle and interior load factor are used, then the indeterminations of ϕ and *n* are given by Eq. (22) and Eq. (23), respectively. At this point, from the equation for P_V in Eq. (30) with $P_\gamma=0$ and $P_\psi=0$,

$$\frac{dP_V}{dt}=-\frac{C_0}{V}<0 \quad (37)$$

Hence the connection is from a *B* arc to a *C* arc. If all P_V, P_γ, and P_ψ approach zero simultaneously, and the boundary bank angle and interior load factor are used, the indetermination of n is given by Eq. (26). At this point, from the equation for P_V in Eq. (30) with $P_\gamma=0$ and $P_\psi=0$, Eq. (37) is true. Hence the connection is from a *B* arc to a *C* arc. If the discontinuity of the angle of attack is neglected, the switching is always at $n=n_s$ or $n=M^2C_{L_{max}}/\omega$. If it is at $n=n_s$, then a switching from a *C* arc to a *B* arc is optimal if at the switching point $dP_V/dt>0$, i.e., from Eq. (31), we have

$$P_\gamma\frac{2g}{V}\left(n_{max}\cos\phi-\cos\gamma\right)+P_\psi\frac{2gn_{max}\sin\phi}{V\cos\gamma}>C_0 \quad (38)$$

For a switching from a *B* arc to a *C* arc the above inequality is reversed. If the switching is at $n=M^2C_{L_{max}}/\omega$, then a switching from a *C* arc to a *B* arc is optimal if at the switching point $dP_V/dt>0$, i.e., from Eq. (34) we have

$$C_0+P_\gamma\frac{g}{V}\left(2\cos\gamma+n\cos\phi\, C_{L_{max_M}}\right)$$
$$+P_\psi\frac{gn\sin\phi}{V\cos\gamma}C_{L_{max_M}}<0 \quad (39)$$

This inequality is reversed for a switching from a *B* arc to a *C* arc.

COMPUTATIONAL RESULTS

The problem of minimum-time supersonic chandelle with free final longitudinal range X_f and free final lateral range Y_f is completely solved in real time, on line, by Lin.[4] Instead of free final longitudinal and lateral ranges, as in Lin,[4] this paper focuses on prescribed final longitudinal range X_f and prescribed final lateral range Y_f. Furthermore, the final altitude is prescribed. Thus this problem is minimum-time supersonic chandelle with prescribed final point. This particular three-dimensional turn can be analyzed in comparison with the trajectories on a horizontal and a vertical plane, as shown in Figure 1.

Figure 1 gives a comparison of three 180° turning maneuvers, i.e., the horizontal or level turn, the vertical turn or Immelman, and the chandelle.[5] These three turnings are useful maneuvers in combat. In the figure, I, II, and III represent three different positions of the intruder's inbound. Consider 0 the offset point of our aircraft. The goal of all three turning maneuvers is to reach the attack cone, defined as any position to the rear of the target from which we can maneuver and overtake the target in the position of the weapon-firing

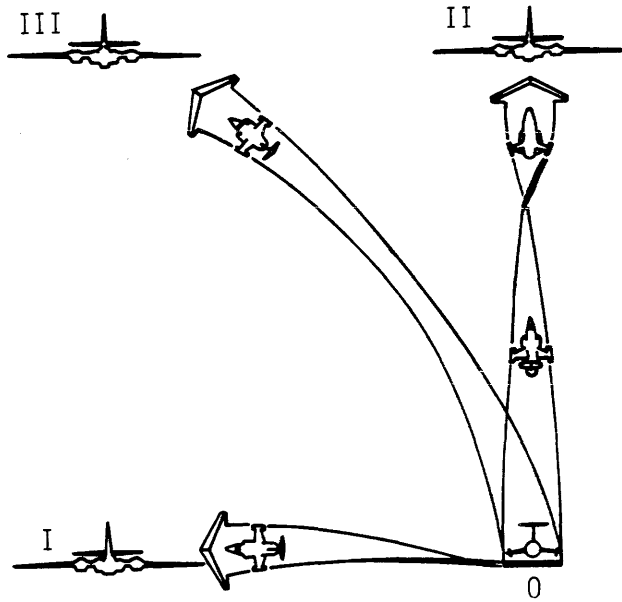


Figure 1—180° turns

ranges. In general, the attack cone lies in the 5 to 7 o'clock position behind the target and at coaltitude or slightly below the target altitude. Depending on the position of the intruder's inbound, an appropriate turning maneuver is chosen for interception. For example, if the position of the intruder's inbound is Location I, then the horizontal turning maneuver is used. This maneuver as shown in the figure is a turn to a heading. If the position of the intruder's inbound is Location II, the vertical turning maneuver needs to be performed to get to the attack cone. The vertical turning maneuver as shown in the figure is a particular vertical turn called the Immelman. If the position of the intruder's inbound lies between Locations I and II, e.g., in Location III in the figure, the appropriate tactical maneuver to choose is the chandelle, which is a three-dimensional 180° climbing turn. For simplification, in this paper we define the offset point as the position when the fighter arrives at Mach two and is ready to immediately initiate the minimum-time supersonic chandelle to reach the attack cone. The altitude of this position is referred to as the initial altitude of the offset point.

In this problem we have the terminal conditions

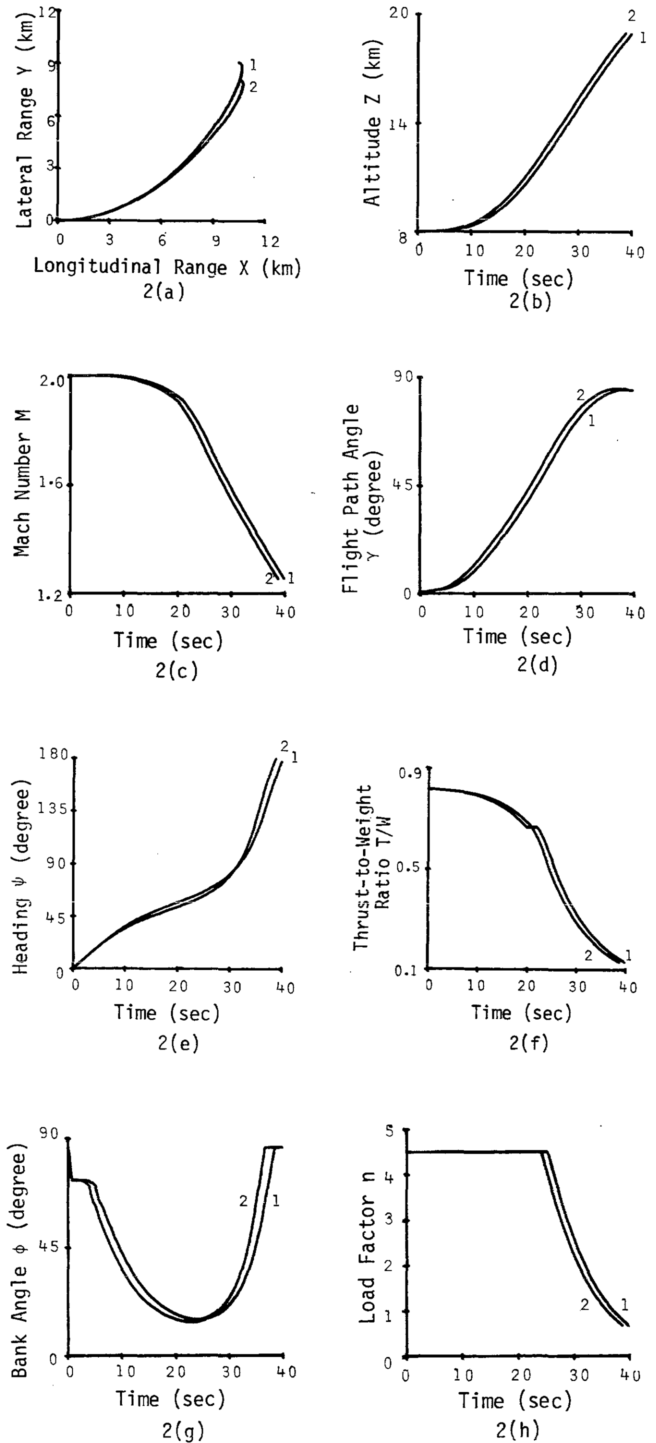
$$t = 0,\ X = 0,\ Y = 0,\ Z = Z_0,\ V = V_0,\ \gamma = \gamma_0,\ \psi = 0° \quad (40)$$

$$t_f = \text{min.},\quad X = X_f,\quad Y = Y_f,\quad Z = Z_f,\quad V_f = \text{free},\quad \gamma_f = \text{free},\quad \psi_f = 180° \quad (41)$$

Hence, we have the transversality conditions

$$C_0 = 1,\ P_{V_f} = 0,\ P_{\gamma_f} = 0 \quad (42)$$

Since $P_{V_f} = P_{\gamma_f} = 0$, $P_{\psi f} \neq 0$, by continuity of the bank angle and load factor the last portion of the trajectory must be flown with boundary bank angle and boundary load factor. For $n_f = n_s$, if the terminal point is considered a switching point, then condition (38), with $P_{\gamma_f} = 0$ at the final time, dictates a final *C* arc. If this resulting inequality reverses, the final arc is a *B* arc. For $n_f = M^2 C_{L_{max}}/\omega_f$, if the terminal point is considered a switching point, condition (39) with $P_{\gamma_f} = 0$ at the final time dictates a final *C* arc. If this resulting inequality reverses, the final arc is a B arc. The problem in terms of C_1, C_2, C_3, P_{V_0}, and P_{γ_0} is a five-parameter problem. The five parameters C_1, C_2, C_3, P_{V_0} and P_{γ_0} are to be selected to satisfy the final



Figures 2a-h—Minimum time supersonic chandelle with prescribed final point

conditions (41) and the transversality conditions (42). For the solution we guess C_1, C_2, C_3, P_{V_0}, and P_{γ_0} and start the integration of the state equations (1) and adjoint Eqs. (27) and (29), along with the use of the optimal thrust and aerodynamic control law. At the final heading $\psi_f = 180°$, the conditions on $X = X_f$, $Y = Y_f$, $Z = Z_f$, $P_{V_f} = 0$, and $P_{\gamma_f} = 0$ are used to adjust the five unknown parameters C_1, C_2, C_3, P_{V_0}, and P_{γ_0}.

Figures 2a–2h show the results of the computation obtained by running a very-high-speed digital computer in real time, on line, using the supersonic fighter as the model. These results are obtained by using the example problem of minimum-time supersonic chandelle, turning from an initial point of $X_0 = 0$ km, $Y_0 = 0$ km, $Z_0 = 8$ km, with an initial Mach two, to a prescribed final point of $X_f = 10.5$ km, $Y_f = 9$ km, $Z_f = 19$ km for Trajectory 1, and $X_f = 10.5$ km, $Y_f = 8$ km, $Z_f = 19$ km for Trajectory 2. The constraints $\phi_{max} = 1.5$ radians and $n_s = 4.5$ are imposed. The indetermination in evaluating the initial value of P_Z when $\gamma_0 = 0°$ is avoided by using initially a slightly positive value of γ_0, since the trajectory has the tendency to start with a climb for a high initial Mach two. For the same prescribed final altitude $Z_f = 19$ km and the same prescribed final longitudinal range $X_f = 10.5$ km, Trajectory 1 has a longer prescribed final lateral range $Y_f = 9$ km and hence requires longer time to complete ($t_f = 40.1$ seconds), whereas Trajectory 2 has a shorter prescribed final lateral range $Y_f = 8$ km and hence takes a shorter time to complete ($tf = 38.8$ seconds).

REFERENCES

1. Lin, Ching-Fang. "Real-Time, On-Line Digital Simulation of Optimum Maneuvers of Supersonic Aircraft." *Proceedings of the AIAA Computers in Aerospace III Conference,* San Diego, October 1981. New York: American Institute of Aeronautics and Astronautics, 1981.
2. Lin, Ching-Fang. "Real-Time Simulation of Onboard Flight Control Microcomputer System." *Proceedings of the SCS Conference on Modeling and Simulation on Microcomputers,* San Diego, January 1982. La Jolla, California: Society for Computer Simulation, 1982.
3. Lin, Ching-Fang. *Optimum Maneuvers of Supersonic Aircraft.* Ann Arbor: The University of Michigan Publications, 1980.
4. Lin, Ching-Fang. "Microcomputer Control Applications in Onboard Flight Control System." *Proceedings of the IEEE Asilomar Conference on Circuits, Systems and Computers,* Pacific Grove, November 1981. Los Alamitos, California: IEEE Computer Society, 1981.
5. McCorkle, Burt. "Flight Path Optimization Method." Lockheed Report CA/GME 3044.

Structured D-chart: A diagrammatic methodology in structured programming

by C. JINSHONG HWANG
Purdue University
W. Lafayette, IN

ABSTRACT

The rules and principles of structured programming resemble the rules and principles of good musicianship. Good programmer performance depends on both a competent programmer and the proper logic design methodologies. This paper presents a new diagrammatic methodology for such programming that accurately depicts the restricted control structures and their close correlation with natural thought process. A good programming style and coding indentation are the direct results of the use of structured D-charts.

1. INTRODUCTION

During the 1970s the structured programming revolution produced several significant results:

a. The principle of good programming was universally accepted.[5]
b. The use of restricted control structures and top-down programming were widely accepted methods.[6]
c. The flow chart was developed as a schematic depiction of restricted control structure specification of program logic.[7] The flow chart depictions are shown in Figure 1.

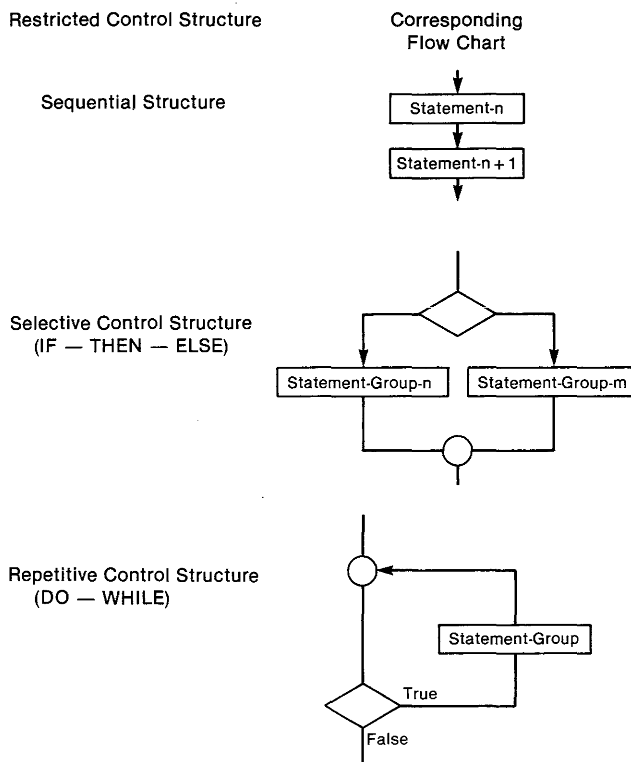


Figure 1—Flow chart as a schematic depiction of restricted control structure specification of program logic

The IF-THEN selective structure is a particular case of the IF-THEN-ELSE selective structure. The CASE structure is a generalized case of the selective structure. DO-UNTIL is an alternative repetitive structure.

The rules and principles of structured programming resemble the rules and principles of good musicianship. A good musical performance depends on both competent musicians and proper instruments. The currently most popular instrument for depicting program structure and logic is the conventional flow chart described above. The controversies surrounding structured programming and the GOTO statement[3,4] pertain mostly to the use of such an instrument.

The use of flow charts to depict program structure and logic can make it easier for a programmer to violate the single-entry, single-exit rule for programs and program modules. The composition rules for flow charts make it possible to draw a chart, using lines that cross one another and move off in all directions. Flow charts are not very suitable for showing how structured algorithms closely reflect natural thinking and problem-solving processes.

Other instruments introduced to represent structured programming include pseudocode and the Nassi-Shneiderman chart,[8] or something similar like the Chapin Chart.[9] Pseudocode, however, is not a diagrammatic visual aid for designing program logic. The Nassi-Schneiderman chart does not indicate the logic flow or progression in a clear, concise flow manner. All these instruments, of course, have their advantages and disadvantages, but this article presents a new diagrammatic methodology for structured programming that accurately depicts restricted control structures and their close correlation with natural thought processes. Parallel with the structured D-charts presented herein, pseudocode will be used to express the meaning of structured D-chart in a disciplined, restricted narrative. The original idea for the D-chart appeared in Bruno and Steiglitz.[1] (The "D" in D-chart is in honor of Edsgar W. Dijkstra, who was one of the earliest proponents of structured programming.)[3] Certain revisions for the idea of D-chart appeared in Denning and Denning.[2] This article presents a new revision of the D-chart, called the structured D-chart, which can be used by all levels of programming students and professionals. The structured D-chart was developed by the author in fall 1978, and teaching experiments using structured D-charts have continued for four years. At the end of this article, some results of these experiments will be presented. The next section will present the composition of the structured D-chart with respect to control structures and the meaning of the symbols used in the structured D-chart. Section 3 will discuss the implementation of the structured D-chart in non-structured FORTRAN, COBOL, BASIC-PLUS, and PASCAL. In that section we shall see that the use of the structured D-chart is universally applicable to every kind of programming language. Section 4 will describe the relationship between the structured D-chart and programming style.[10] Section 5 will present some simple rules for using the structured D-chart. Section 6 will summarize the advantages of the structured D-chart and provide a direct comparison between the structured D-chart and the flow chart. Finally, the results of the teaching experiments will be described.

2. STRUCTURED D-CHARTS AND RESTRICTED CONTROL STRUCTURES

2.1 Structured D-chart Symbols

This section gives a detailed description of structured D-charts by illustrating and explaining the symbols used to construct them. Structured D-chart representation of control structures will be emphasized.

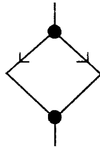
Structured D-charts are made up of a limited set of special geometric symbols, corresponding to specific parts of a program unit. These symbols and their meanings are as follows:

An oval (⬭) indicates the starting and ending point for the program unit and a return to a main program from a subroutine.

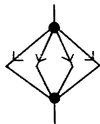
A parallelogram (▱) indicates general input-output operations, the input, reading, and printing of data.

A rectangle (▭) indicates assignment and arithmetic operations, where the assignment of values and computation of arithmetic operations occur.

A large dot shows the upper boundary or lower boundary of a selective control structure and designates the point at which this control structure begins or ends. All statements within this control structure will be executed, depending upon the status of a certain condition. Dots must appear in pairs to indicate one entrance into and one exit from the selective control structure.



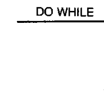
Two or more arrows emanating from a large dot and diverging downward indicate the multiple alternate paths of a selective control structure. Each diverging arrow eventually converges into another dot which marks the end of the selective control structure.



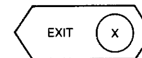
A small circle indicates the top boundary or delimiter of repetitive control structures and indicates at what point the repetitive control structure begins. All statements contained within a repetitive control structure are executed according to the status of specific condition. The circle ⓧ contains an alphabetic character to uniquely identify each repetitive control structure.

A small triangle indicates the lower boundary or delimiter of repetitive control structures, showing at what point the repetitive control structure ends. The triangle contains an alphabetic character matching the character in the top delimiter ⓧ of the same repetitive control structure.

An arrow is used to indicate the flow of the D-chart. In a repetitive structure, the flow should always be to the right and down. In a DO-WHILE repetitive control structure, the conditions that determine the control structure flow will be written directly above the horizontal arrow.

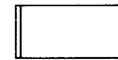


This figure indicates an interrupt exit from a repetitive control structure to the first executable statement immediately following the repetitive control structure ⓧ. The alphabetic character in the circle ⓧ identifies the repetitive control structure from which the exit is to be made.

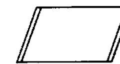


A rectangle of broken lines indicates that the control structure causes an automatic increment. The auto-increment is part of the DO-FROM-TO control structure. (DO-FROM-TO is an alternate form of repetitive control structure; see later in this section.)

This block figure indicates that control is passed to a subroutine, procedure, or a block of program statements, located in a separate structured D-chart. It indicates a CALL to a subroutine. Subroutines end with an oval, indicating a RETURN to the main program.



This figure indicates an implied repetitive control structure for input or output from a collection of related data items in an array.



A connector symbol indicates the continuation of the structured D-chart on another page. It should not be used for the branching of execution control. It is only used for the connection of pages. One symbol at the end of the first page and another symbol at the beginning of the second page. Numerals inside the connection indicate the location of connection.

2.2 Restricted Control Structures

Sequential control is the simplest type of control structure: control goes from statement to statement in a straight uninterrupted line. The structured D-chart and pseudocode in Figure 2 show the flow of control in a sequential control structure.

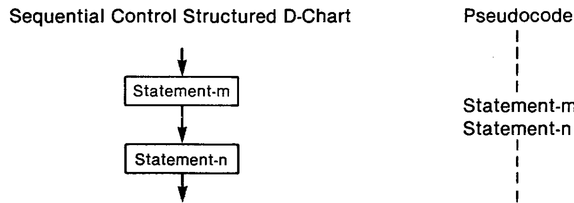


Figure 2—Flow of control in a sequential control structure

After the execution of statement-*m*, the next sequential statement, statement-*n*, is executed, and so on for all statements controlled by a sequential structure.

2.2.2 Selective Control Structured D-Chart

Selective control is more sophisticated than sequential control and affords the programmer more power and flexibility. Selective control is applicable in situations where a sequence of program statements is executed, depending upon the status of a specific condition. Selective control is identified by the use of IF-THEN, IF-THEN-ELSE, and CASE constructs within a program. The flow of logic indicated by the use of a selective control structure is illustrated by the structured D-charts and pseudocode in Figures 3a, 3b, and 3c.

In Figure 3a, after the execution of statement-*m*, the status of the condition is evaluated. If the status is TRUE, statement-group is executed. If the status of condition is FALSE, statement-*m* is followed directly by statement-*n*. Statement-*m* and statement-*n* are executed regardless of the status of the condition.

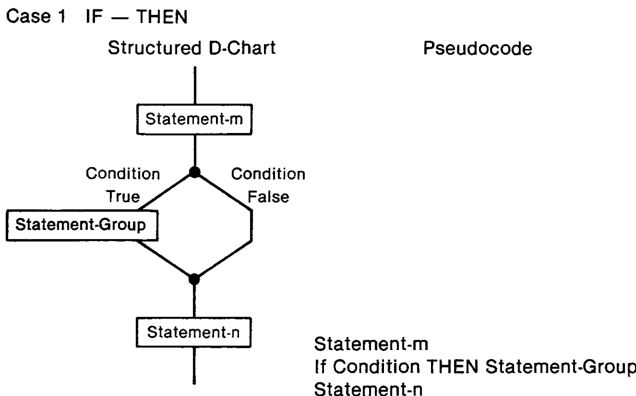


Figure 3a—Flow of logic in a selective control structure, Case 1

In Figure 3b, after the execution of statement-*m*, the status of condition is evaluated. A status of TRUE causes the execution of statement-group-1, followed by the execution of statement-*n*. If the status is FALSE, statement-group-2 is executed, followed by statement-*n*. In either situation, statement-*m* and statement-*n* are executed.

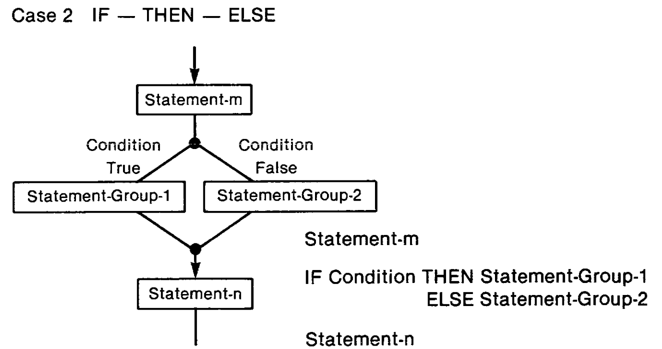


Figure 3b—Flow of logic in a selective control structure, Case 2

In Figure 3c, after statement-*m* is executed, the expression is evaluated. Assume the values of expression are positive integers between 1 and *i*. The line numbers are the statements or statement groups to which control is to be passed according to the integer value. In this example, if the value is 1, stagement-group-1 is executed, followed by statement-*n*. A value of 2 transfers control to statement-group-2. The value determines the flow of the program. Statement-*m* and statement-*n* are executed regardless of the value.

Case 3 CASE Construct

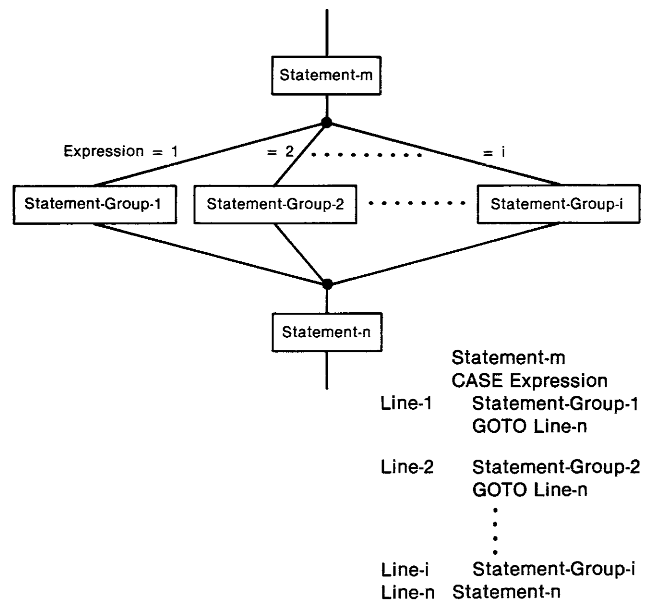
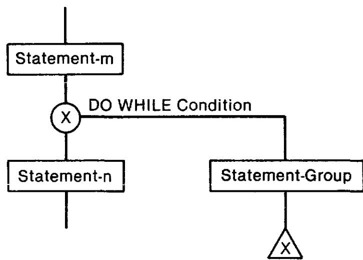


Figure 3c—Flow of logic in a selective control structure, Case 3

Case 1 DO WHILE

Structured D-Chart



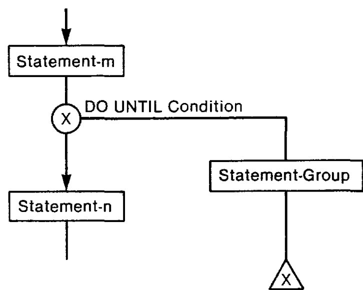
Pseudocode

Statement-m

DO WHILE Condition
  Statement-Group
END-DO

Statement-n

Case 2 DO UNTIL

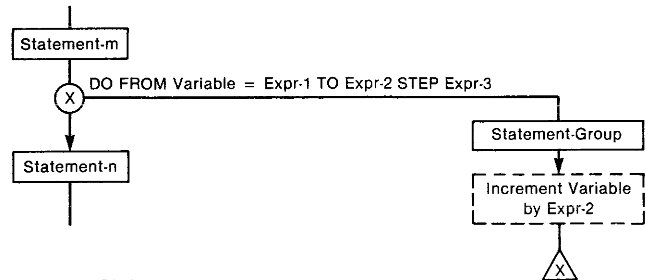


Statement-m

DO UNTIL Condition
  Statement-Group
END-DO

Statement-n

Case 4 DO-FROM-TO

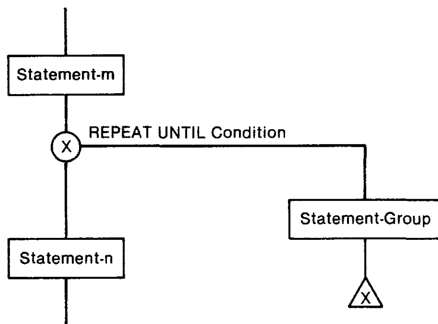


Statement-m

DO FROM Variable = Expr-1 TO Expr-2 STEP Expr-3
  Statement-Group
END-DO

Statement-n

Case 3 REPEAT UNTIL

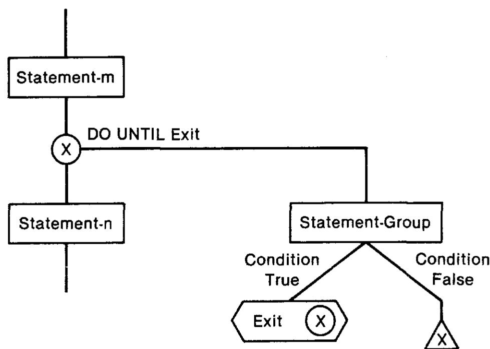


Pseudocode for Language With REPEAT

Statement-m

REPEAT UNTIL Condition
  Statement-Group
END-REPEAT

Statement-n



Pseudocode for Language Without REPEAT

```
       Statement-m

Line-i Statement-Group
       IF Condition
          THEN Line-i
          ELSE Line-j

Line-j Statement-n
```

Figure 4—Structured D-charts and pseudocode for repetitive structures

2.2.3 Repetitive Control Structured D-Chart

The repetitive structure is the third type of control structure. The distinctive characteristic of a repetitive structure is that it causes a statement or group of statements to be executed repeatedly, according to the value of a specified condition. There are four kinds of repetitive structures: the DO WHILE structure, where statements are executed repeatedly, while the logical value of the specified condition is TRUE; the DO UNTIL structure, where a group of statements is executed repeatedly, until the logical value of the specified condition becomes TRUE; the REPEAT UNTIL structure, where the condition is checked at the end of the repetitive structure while the similar DO UNTIL structure checks the condition at the beginning of the structure; and the DO FROM-TO structure, where a group of program statements is executed for a specified number of times. The structured D-charts and pseudocode for these repetitive structures are as shown in Figure 4, Cases 1–4.

After statement-*m* is executed, the value of condition is evaluated. While the logical value of condition is TRUE, the statement-group is executed. As in the selective control structure, the statement-group in a repetitive control structure can be one or more statements. When the logical value of condition becomes FALSE, control is transferred to statement-*n*.

The execution of statement-*m* is followed by the evaluation of the specified condition. Until the logical value of the condition becomes TRUE (i.e., while it is FALSE), the statement-group is executed. When the logical value of the condition becomes TRUE, control is transferred to statement-*n*.

The REPEAT UNTIL structure is a special case of the repetitive structure which is used to ensure that a given statement or statement block within a repetitive structure will be executed at least once. Some programming languages do not provide a formal statement or set of statements to accomplish this task efficiently. REPEAT-UNTIL structure provides this special case of the repetitive structure. Despite its name, the REPEAT-UNTIL structure does not contain the UNTIL statement in most languages; it is made up of an IF-THEN-ELSE selective structure. The structured D-chart and pseudocode for the REPEAT-UNTIL structure are as follows.

After entering the loop and the execution of statement-group, the conditional expression is evaluated. If the logical value is TRUE, the loop exits and statement-*n* is executed. A logical value of FALSE causes statement-group to be executed again, thus creating a repetitive structure. This iterative process is repeated, until the logical value becomes TRUE, at which point the loop exits. The conditional expression is not evaluated, until statement-group has been executed for one time. This guarantees that statement-group will be executed at least once. In the DO-UNTIL control structure, on the other hand, the logical value is determined prior to the execution of the statement group contained in the repetitive structure. The REPEAT-UNTIL structure eliminates repetitious code to guarantee one execution of a statement block. This special case of repetitive control structure can resolve the controversies over the proper use of the GOTO statement. The GOTO statement may be used if for the purpose of implementing REPEAT-UNTIL structure in a language without REPEAT or for the error handling.

After the execution of statement-*m*, the value of expr-1 is assigned to variable and statement-group is executed. The value of variable is then incremented by the value of expr-3 and variable is evaluated to determine if it exceeds the value of expr-2. Statement-group is repetitively executed, until the value of variable becomes greater than the value of expr-2. At that point, control is transfered to statement-*n*.

3. IMPLEMENTATION OF STRUCTURED D-CHARTS

This section demonstrates the implementation of structured D-charts in programming languages such as nonstructured FORTRAN, COBOL, BASIC-PLUS, and PASCAL. The example algorithm is the bubble sorting algorithm, which reads a set of numbers until an end of file marker is encountered, sorts the numbers in ascending order and prints the sorted result. The logic involves a combination of all three restricted control structures. The same structured D-chart (Figure 5) for the bubble sorting algorithm will be used to implement the algorithm in four different programming languages. Furthermore, the same structured D-chart can be implemented in assembly language.[11] The structured D-chart is applicable as a flow diagram for both structured and nonstructured programming languages.

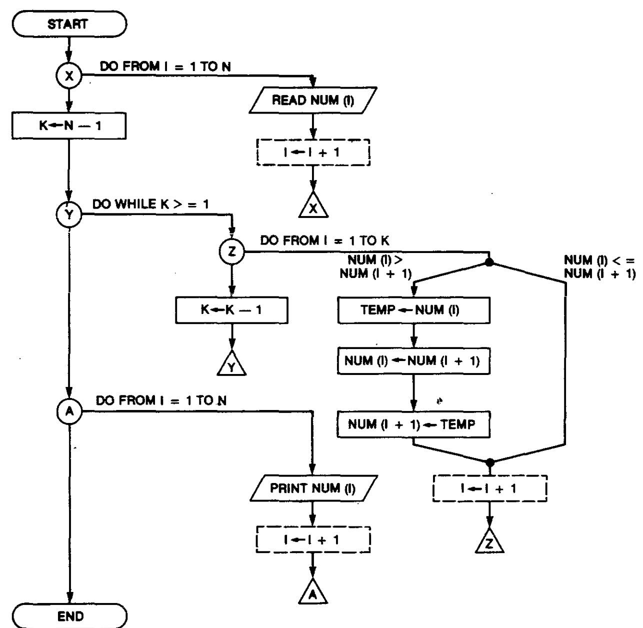


Figure 5a—Bubble sorting algorithm

4. STRUCTURED D-CHART AND PROGRAMMING STYLE

Good programming style complements structured programming by clearly representing control structures and their pur-

```
PROGRAM SORT (INPUT, OUTPUT) ;

VAR K, I, J, N, TEMP : INTEGER ;
    NUM : ARRAY[1..10] OF INTEGER ;

BEGIN
N  := 10 ;

FOR J := 1 TO N DO READ (NUM[J]) ;

K := N - 1 ;

WHILE K >= 1 DO
  BEGIN
  FOR I:= 1 TO K DO
      BEGIN
      IF NUM[I] > NUM[I+1] THEN
         BEGIN
         TEMP := NUM[I] ;
         NUM[I] := NUM[I+1] ;
         NUM[I+1] := TEMP ;
         END
      END ;

      K := K - 1
  END ;

FOR I := 1 TO N DO
    WRITELN (NUM[I]) ;
END.
```

Figure 5b—Implementation in PASCAL

```
100      DIM NUMS%(10%)
110 !
120      N% = 10%
130 !
140      FOR J% = 1% TO N%
150          READ NUMS%(J%)
160      NEXT J%
170 !
180      K% = N% - 1%
190 !
200      WHILE K% >= 1%
210 !
220             FOR I% = 1% TO K%
230                 IF NUMS%(I%) <= NUMS%(I%+1%) THEN 270
240                    TEMP% = NUMS%(I%)
250                    NUMS%(I%) = NUMS%(I%+1%)
260                    NUMS%(I%+1%) = TEMP%
270             NEXT I%
280 !
290             K% = K% - 1%
300      NEXT
310 !
320      FOR I% = 1% TO N%
330          PRINT NUMS%(I%)
340      NEXT I%
350 !
360      DATA 823, 791, 768, 587, 456, 345, 268, 212, 123, 100
370      END
```

Figure 5c—Implementation in BASIC-PLUS

```
         INTEGER I, J, N, K, NUM(10), TEMP

         OPEN (UNIT=1, NAME='TEST.DAT', TYPE='OLD', READONLY)

         N = 10

10       READ(1,*) (NUM(J), J = 1, N)

20       K = N - 1

25       IF (K .LT. 1) GOTO 35

            DO 30 I = 1, K

               IF (NUM(I) .LE. NUM(I+1)) GOTO 30
                  TEMP = NUM(I)
                  NUM(I) = NUM(I+1)
                  NUM(I+1) = TEMP

30          CONTINUE

            K = K - 1
         GOTO 25

35       DO 40 I = 1, N
            WRITE(7,*) NUM(I)
40       CONTINUE

         CLOSE (UNIT=1, DISPOSE='SAVE')

         STOP
         END
```

Figure 5d—Implementation in nonstructured FORTRAN

poses. The structured D-chart not only clearly reflects natural thinking via restricted control structures, but also accommodates the indentation requirements of program style in a very direct way. The use of indentation to indicate control structures in a program is one of the elements of good programming style. The degree of indentation depends upon the programmer, but the level of indentation should correspond to the level of the control structures within the program. The structured D-chart (Figure 6) illustrates what is meant by the level of a control structure.

In structured programming, a level refers to a series of sequential statements. Whenever a series is broken by a control structure (DO WHILE, DO UNTIL, DO FROM-TO, IF-THEN or IF-THEN-ELSE) all subsequent statements belonging to that control structure are considered to be on another level, and should be indented accordingly. All statements of the same level should be indented the same number of spaces. For example, the pseudocode in Figure 7 corresponds to the structured D-chart (Figure 6), and illustrates what the indentation should look like.

Note that all statements of a given level are indented the same number of spaces. All Level 1 statements are not indented, all Level 2 statements are indented 5 spaces, and all Level 3 statements are indented 10 spaces. If there were a Level 4, it would be indented 15 spaces, and so on for any other levels. The number of spaces of indentation for each level is up to the programmer, as long as it makes the structure clear. Compare the pseudocode with indentation (Figure 7) to pseudocode in Figure 8. Note how much easier the indented pseudocode is to read, how each control structure is more clearly defined, and how the structured D-chart is used to reflect the indentation levels of the programming style. The logic of this pseudocode is much harder to follow, because of the lack of indentation. The use of indentation is a very powerful tool in the writing of clear and easy-to-read programs. The example in Figure 8 on control structure levels clearly points out another advantage of using the structured D-chart. Indentation for good programming style comes very naturally from the logic design of the structured D-chart. The control levels of structured D-charts directly tells the programmer

```
IDENTIFICATION DIVISION.

PROGRAM-ID.        PROJ1B.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.          PDP-11.
OBJECT-COMPUTER.          PDP-11.

INPUT-OUTPUT SECTION.

FILE-CONTROL.
    SELECT INPUT-DATA
       ASSIGN TO READER.
    SELECT OUTPUT-RESULT
       ASSIGN TO PRINTER.

DATA DIVISION.

FILE SECTION.

FD  INPUT-DATA
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS NUMBER-INPUT-RECORD.
01  NUMBER-INPUT-RECORD.
    05 NUM-INPUT                      PIC 999 OCCURS 10 TIMES.
    05 FILLER                         PIC X(50).

FD  OUTPUT-RESULT
    RECORD CONTAINS 133 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS OUTPUT-REPORT-RECORD.
01  OUTPUT-REPORT-RECORD.
    05 CARRIAGE-CONTROL               PIC X.
    05 NUM-OUTPUT                     PIC X(3).
    05 FILLER                         PIC X(129).

WORKING-STORAGE SECTION.
77 TEMP                               PIC 999.
77 N                                  PIC 99.
77 I                                  PIC 99.
77 J                                  PIC 99.
77 K                                  PIC 99.

PROCEDURE DIVISION.

    OPEN INPUT INPUT-DATA
         OUTPUT OUTPUT-RESULT.
    MOVE 10 TO N.
    READ INPUT-DATA.
    COMPUTE K = N - 1.
    PERFORM OUTER-LOOP
        UNTIL K IS LESS THAN 1.
    MOVE 1 TO I.
    PERFORM WRITE-OUTPUT
        UNTIL I IS GREATER THAN N.
    CLOSE INPUT-DATA
          OUTPUT-RESULT.
    STOP RUN.

OUTER-LOOP.
    MOVE 1 TO I.
    PERFORM INNER-LOOP
        UNTIL I IS GREATER THAN K.
    COMPUTE K = K - 1.

INNER-LOOP.
    COMPUTE J = I + 1.
    IF NUM--INPUT (I) IS GREATER THAN NUM-INPUT (J)
           MOVE NUM-INPUT (I) TO TEMP
           MOVE NUM-INPUT (J) TO NUM-INPUT (I)
           MOVE TEMP TO NUM-INPUT (J).
    COMPUTE I = I + 1.

WRITE-OUTPUT.
    MOVE SPACES TO OUTPUT-REPORT-RECORD.
    MOVE NUM-INPUT(I) TO NUM-OUTPUT.
    WRITE OUTPUT-REPORT-RECORD.
    COMPUTE I = I + 1.
```

Figure 5e—Implementation in COBOL

where to use indentation in programming languages (and in pseudocode).

5. RULES OF STRUCTURED D-CHART COMPOSITION

There is great potential for program flexibility when restricted control structures are used to create a structured program. Structured D-charts are essential to the writing of effective structured programs, and should be understood and used as a basis for good programming practices. The following is a list of rules helpful in creating structured D-charts.

1. A structured D-chart must consist of one or a combination of any of the three types of control structures.
2. There must be only one entrance to and one exit from each selective control structure.
3. Each selective structure must be identified by a pair of large dots to symbolize the entrance into and the exit from selective control structures.
4. There must be only one entrance and one exit for each repetitive structure with the exception of ⟨Exit ⓧ⟩, in which case the exit must be the first executable statement following the repetitive structure ⓧ, which may be a nested outer repetitive structure.
5. Each repetitive structure must be uniquely identified by a different alphabetic character within the boundary symbols (○ and △) for the repetitive structure.
6. The logic of the structured D-chart must proceed from top to bottom with any repetitive structures depicted to the right.
7. A control structure can completely contain another control structure. This is called nested structure. A control structure may not contain only a portion of another control structure. There may be no flow lines drawn to connect one control structure to another external structure or outer (nested) structure, the only exception being the beginning or ending boundary.
8. The GOTO statement may be used to implement a control structure. This is the only time the GOTO statement may be used for the sole purpose of implementing control structures in a particular language, no matter if it is structured or non-structured.
9. The structured D-charts in Figure 9 describe the illegal (left column) and the corrected D-charts (right column.)

6. STRUCTURED D-CHART VS. FLOW CHART

The structured D-chart is superior to the flow chart because it agrees completely with the restricted control structures in structured programming techniques. The logic of a structured program and a structured D-chart involves only three restricted control structures: sequential, selective, and repetitive. Implementing the logic of a structured D-chart as a structured program is a direct one-to-one translation. Structured D-charts are easier to read than flow charts because execution flow always proceeds through a structured program in a downward direction; there are no crossing or upward-pointing

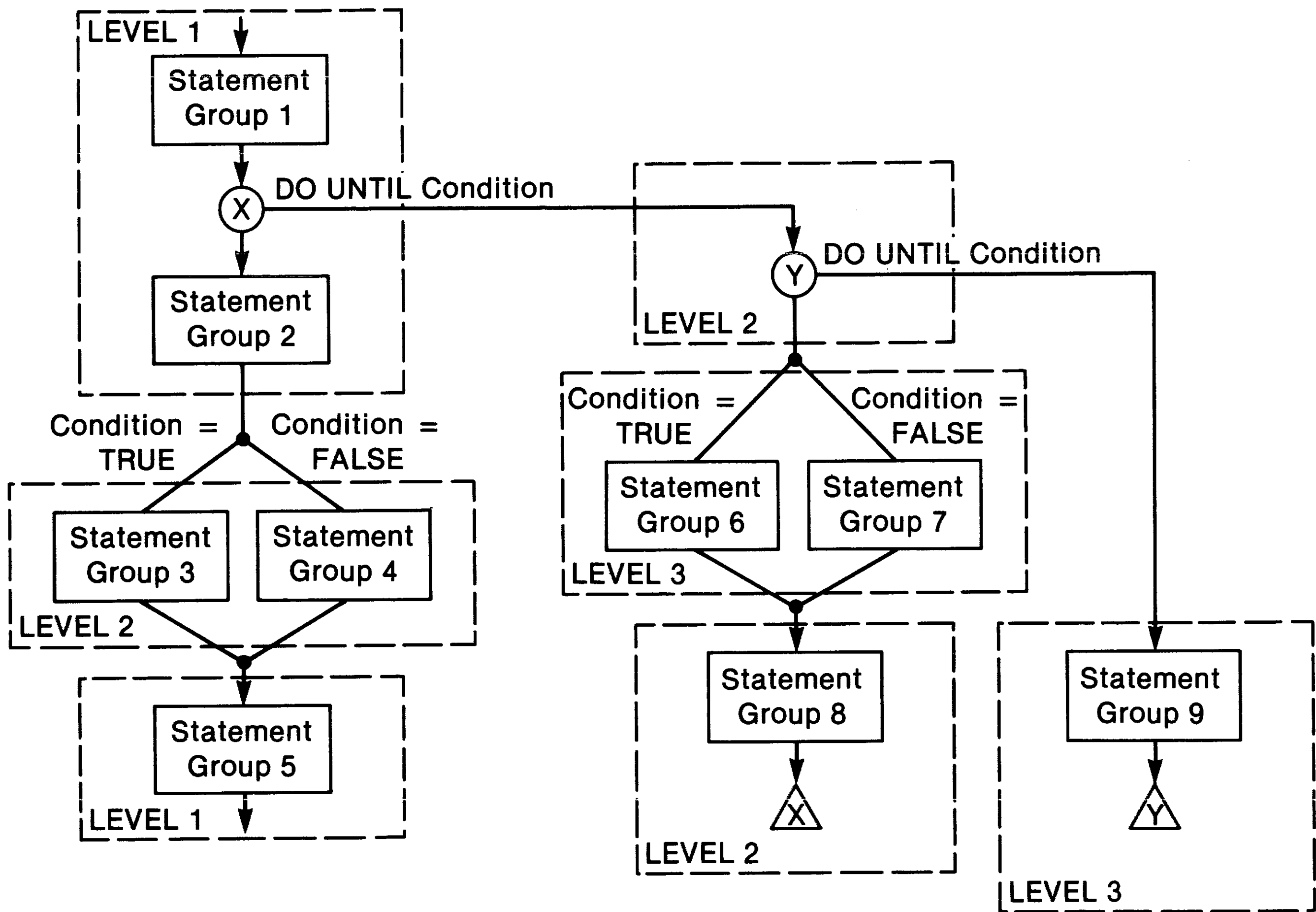


Figure 6—Level of a control structure

lines. All structured D-charts are drawn with single-entry, single-exit control structures, and all conditions are explicitly stated in words similar to those found in the actual program code.

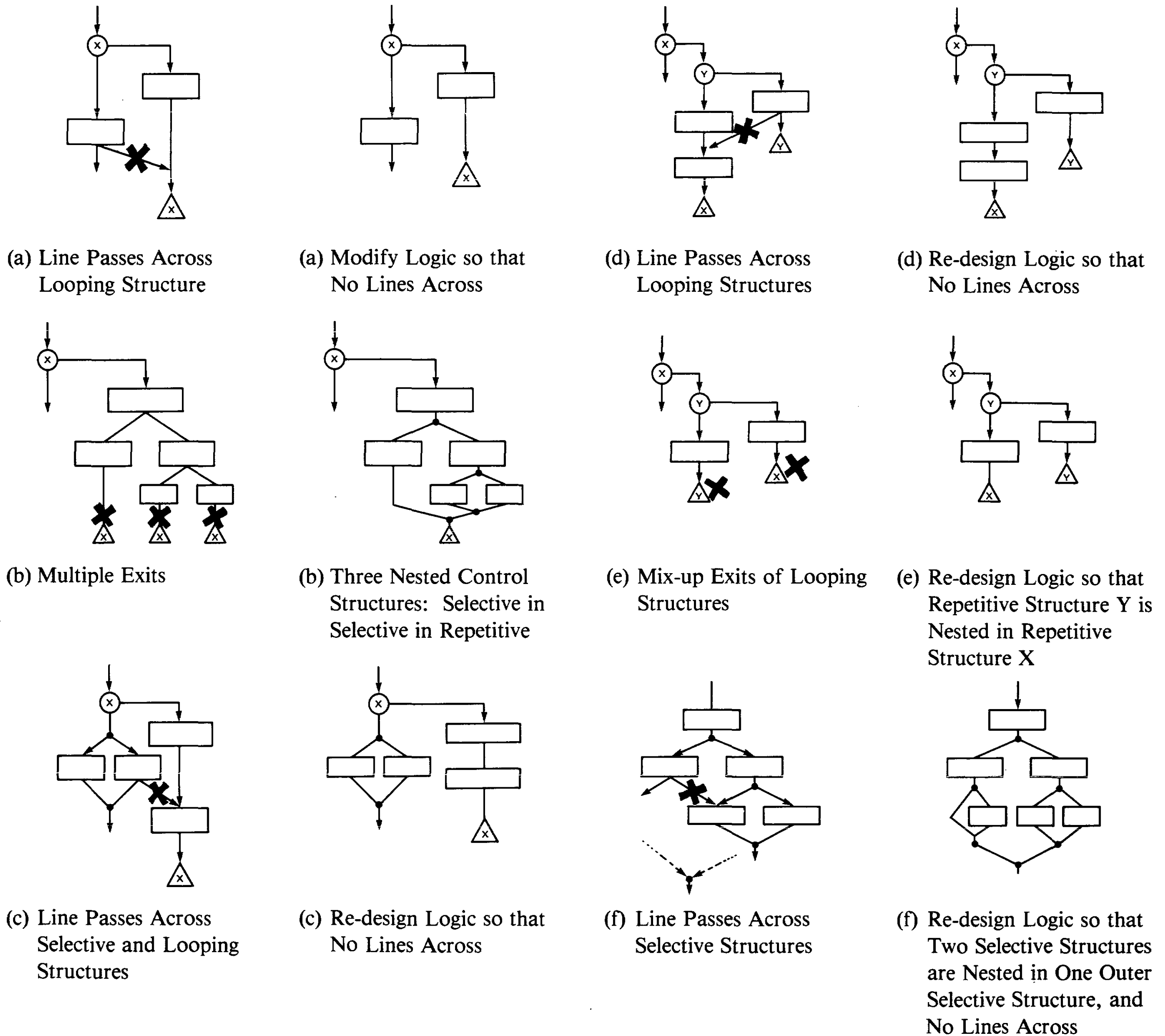
Figure 10 illustrates a conventional flow chart, showing the logic necessary to read a list of number, sort the numbers into ascending numerical order, and print the results. Figure 10 is

```
LEVEL NO.        INDENTATION

  1              Statement Group 1

  1              DO WHILE condition TRUE

  2                  DO UNTIL condition TRUE
  3                      Statement-Group-9
  2                  END-DO

  2                  IF condition TRUE
  3                      THEN Statement-Group-6
  3                      ELSE Statement-Group-7
  2                  Statement-Group-8

  1              END-DO

  1              Statement-Group-2

  1              IF condition TRUE
  2                  THEN Statement-Group-3
  2                  ELSE Statement-Group-4
  1              Statement-Group-5
```

Figure 7—Illustration of indentation in pseudocode

```
Statement Group 1
DO WHILE condition TRUE
DO UNTIL condition TRUE
Statement Group 9
END-DO
IF condition TRUE
THEN Statement Group 6
ELSE Statement Group 7
Statement Group 8
END-DO
Statement Group 2
IF condition TRUE
THEN Statement Group 3
ELSE Statement Group 4
Statement Group 5
```

Figure 8—Pseudocode without indentation



(a) Line Passes Across Looping Structure

(a) Modify Logic so that No Lines Across

(d) Line Passes Across Looping Structures

(d) Re-design Logic so that No Lines Across

(b) Multiple Exits

(b) Three Nested Control Structures: Selective in Selective in Repetitive

(e) Mix-up Exits of Looping Structures

(e) Re-design Logic so that Repetitive Structure Y is Nested in Repetitive Structure X

(c) Line Passes Across Selective and Looping Structures

(c) Re-design Logic so that No Lines Across

(f) Line Passes Across Selective Structures

(f) Re-design Logic so that Two Selective Structures are Nested in One Outer Selective Structure, and No Lines Across

Figure 9—Illegal and corrected D-charts

a well-written flow chart, doing the best possible job of depicting program logic, given the inherent weaknesses of flow charts. Note that it includes crossed lines, lines moving right, and an upward-pointing flow of execution control. The logic is hard to follow and difficult to translate into a structured program.

Figure 5 is a structured D-chart representing the same logic as Figure 10. No lines are crossed; all control structures are single-entry, single-exit; control is never transferred upward; and repetitive structures and their conditions are clearly shown. The control structures in Figure 5 are shown in such a way that code blocks and level breaks (used for the indentation of program lines) are explicitly indicated. The structured D-chart makes it easier to implement structured technique and good programming style.

Real application programs are larger and far more complex than any found here. As a result, the logic required to produce the algorithms for such programs is larger and more complex, sometimes consisting of many pages. Because structured programming is a superior method for creating large programs that are effective and efficient, structured D-charts should be used to represent program logic. A programmer can move from a structured D-chart to a structured program quite easily, for the structured D-chart is based on the same concepts and uses the same control structures in structured programming.

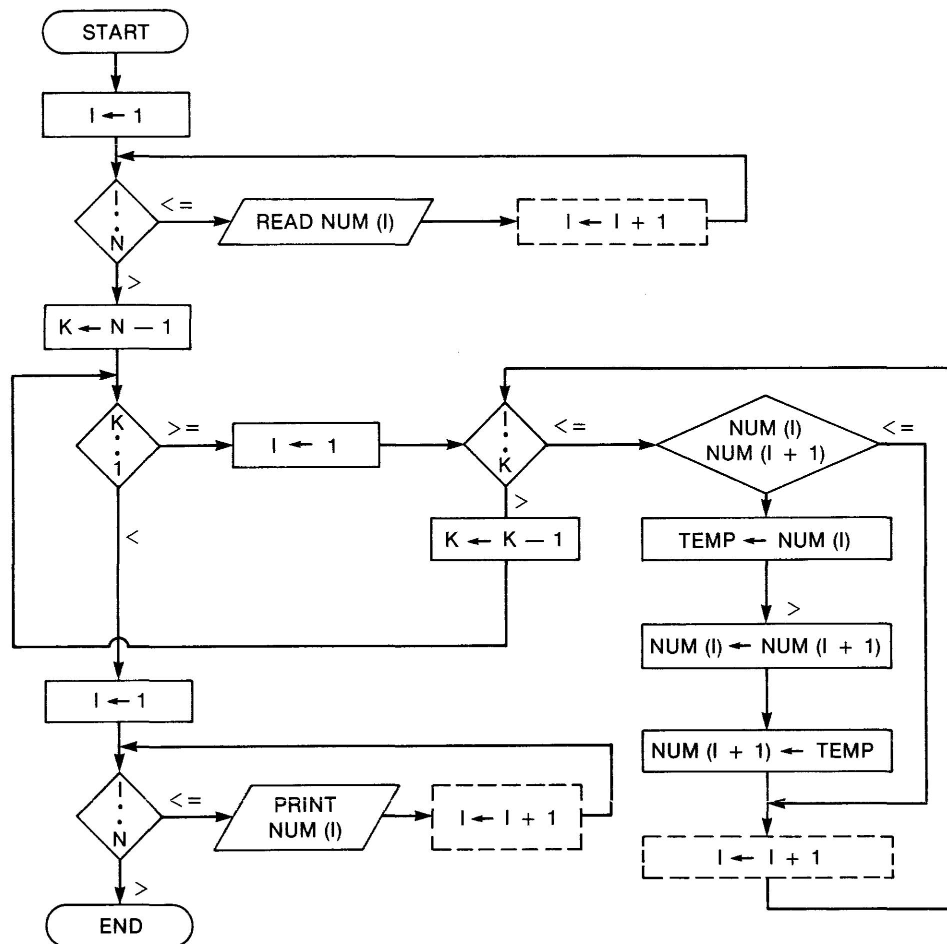


Figure 10—Bubble sorting algorithm in flow chart whose logic is identical to that of Figure 5

7. EXPERIMENTS IN USING STRUCTURED D-CHARTS

In fall 1978 the programming curriculum for the Department of Computer Technology at Purdue University was established. The department undertook to offer an application programming education based upon an understanding of conceptual foundations. The ability to use structured D-charts to express program logic with restricted control structures in the program designing phase is an essential for the conceptual foundations. During the past four years of teaching structured D-charts to freshmen and all new-entry students, we experienced great success with the structured programming method. All students used the structured D-chart method in all programming courses and were informed concerning flow charts during the second year of the curriculum in order that they would be able to communicate with other computer progressionals. But most students have learned about the use of the flow chart before entering our program. They would have to unlearn the nonstructured programming technique of using flow charts.

The following statistics are based upon a survey made in December 1980 of 148 randomly selected students who had had one or more semesters' experience using structured D-charts in writing structured programs.

1. 16 freshman-level students did not know what a flow chart was. They only understood the usage of structured D-charts.
2. 132 freshman-, sophomore-, and junior-level students had a knowledge of flow charts in addition to structured D-charts. Among them,
   a. 86 students learned the use of flow charts before entering our program.
   b. 46 students were exposed to flow charts after learning about structured D-charts.
3. Students were asked to compare structured D-charts with flow charts, if they knew both methodologies.

Question: If you know about structured D-chart as well as flow charts, give a letter grade to the usage of structured D-charts and flow charts in terms of overall performance in logic design, debugging, program understanding, program-

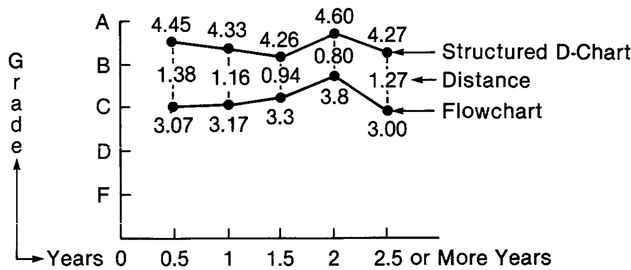


Figure 11—Results of survey of 86 students who learned the flow chart before the structured D-chart

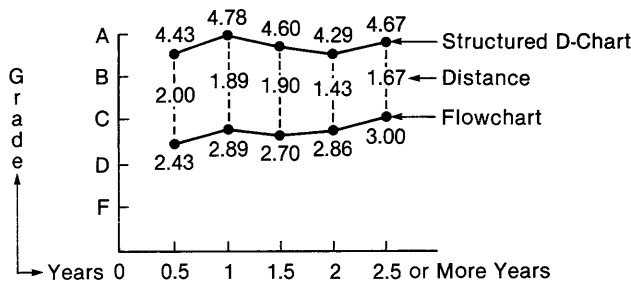


Figure 12—Results of survey of 46 students who learned the structured D-chart before the flow chart

ming style, programming style indentation, and restricted control structure implementation.

Answer: Structured D-chart	A B C D F
Flow chart	A B C D F

a. Giving 5 points for an A and 1 point for an F, the average result of the survey of 86 students who had learned the usage of flow chart before the entrance of our program can be graphed as in Figure 11.

b. On the same 5-point scale, the average result of the survey of 46 students who had learned the structured D-charts from the very beginning and then were exposed to the usage of flowcharts can be graphed as in Figure 12.

c. Interpretation of the figures in 11 and 12:

(1) Students who learned the structured D-chart from the very beginning favored the use of the structured D-chart much more than those who had learned the flow chart before the structured D-chart.

(2) From Figure 11, in spite of the influence of their first experience with flow charts, students still preferred structured D-charts after they learned them. The gap between the rating of structured D-charts and flow charts is narrower in Figure 11 than in Figure 12, possibly because of the difficulty of unlearning flow charts while learning structured D-charts.

(3) In both figures, the distance between structured D-charts and flow charts remains nearly constant. However, the gap closes a bit as the student gets older. This is possibly because they are more mature in their understanding of the fact that the structured programming approach is due to a concept, not to a methodology.

A second survey was made in November 1981 of 138 randomly chosen students who had had one or two semesters' experience using structured D-charts. Its results were very close to the results of the survey made in 1980. Its corresponding figures are as follows:

1. 16 freshmen students did not know what a flow chart was. They only understood the use of structured D-charts.
2. 122 freshman-level students knew the methodologies of flow charts as well as structured D-charts.
   a. 84 students learned about flow charts before entering the program and then learned the use of structured D-charts.
   b. 38 students learned structured D-charts from the very beginning and learned about flow charts later in the year.
3. The distribution curve of 11 and 12 above can be figured as shown in Figures 13 and 14. The figures are very similar to those in the first survey except that the gap between the ratings for the structured D-chart and the flow chart for the 0.5 years in Figure 14 is much closer (0.51). The possible interpretation is that there were only 8 students in that category and that the accuracy of that category might have been affected by one or two students or by simple, inadvertent error entered into the survey.

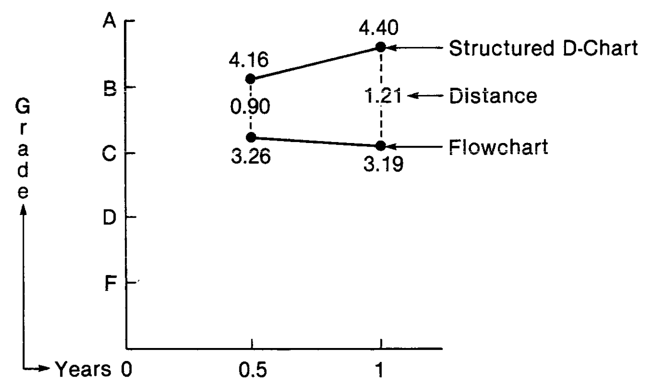


Figure 13—Results of survey of 84 students who learned the flow chart before the structured D-chart

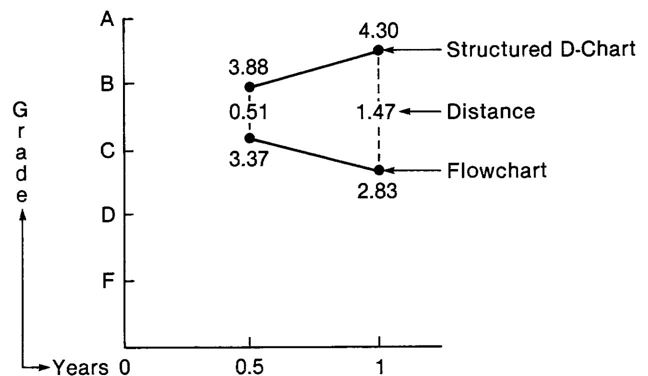


Figure 14—Results of survey of 38 students who learned the structured D-chart before the flow chart

8. ACKNOWLEDGMENTS

The author is indebted to the following individuals for their contributions and suggestions to this article: Phil A. Beetley, Darryl E. Gibson, Jeff Whitten, and all other professors in the Computer Technology Department for their assistance in the survey.

REFERENCES

1. Bruno, J., and K. Steiglitz. "The Expression of Algorithms by Charts." *Journal of ACM,* 19 (1972), pp. 517–525.
2. Denning, P. J., and D. E. Denning. "D-charts." Purdue University, Department of Computer Science, 1976.
3. Dijkstra, E. W. "GO TO Statement Considered Harmful." *Communications of the ACM,* 11 (1968), pp. 147–148.
4. Knuth, D. E. "Structured Programming with GO TO Statements." *ACM Computing Surveys,* 6 (1974), pp. 261–301.
5. Denning, P. J. "Guest Editor's Overview." *ACM Computing Surveys,* 6 (1974), pp. 209–212.
6. Wirth, N. "On the Composition of Well-Structured Programs." *ACM Computing Surveys,* 6 (1974), pp. 247–260.
7. Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming."* New York: Academic Press, 1972.
8. Nassi, I., and B. Shneiderman. "Flowchart Techniques for Structured Programming." *SIGPLAN Notices,* 8 (1973), pp. 12–26.
9. Chapin, Ned. "New Format for Flowcharts." *Software—Practice and Experience,* 4 (1974), pp. 341–357.
10. Hwang, C. J., and Thomas Ho. "Structured Programming in BASIC-PLUS." New York: John Wiley and Sons, 1982.
11. Hwang, C. J. *Structured Programming in PDP 11 Assembly Language.* Englewood Cliffs, New Jersey: Prentice-Hall, 1983.

Planning for software tool implementation: experience with Schemacode

by PIERRE N. ROBILLARD and RÉJEAN PLAMONDON
Ecole Polytechnique
Montréal, Canada

ABSTRACT

The interactive tool called Schemacode assists users in the development, documentation, and structured coding of programs. Its unique property of word-graphic type of communication can be of great help during the main phase of program development: defining the control structure of the program at different levels of refinement.

A Schematic Pseudocode which represents the control skeleton of the program constitutes the very-high-level language input. Construction of the schematic structure and its translation into an appropriate language are the two main functions of Schemacode. These transformations involve editing, formatting, cross referencing, and structure checking. The use of a formal language appears only at the end of the development, after the logic of the problem has been solved.

A real advantage provided by Schemacode is that every program developed has a unique up-to-date documentation and listing. However, modest changes are made in the way programming is done. An integration plan is specifically designed to minimize disturbances in the work milieu where Schemacode is to be implemented and is thus effected in three phases: creation first of a *virtual environment,* then of *linked environment,* and finally of an *integrated environment.* The virtual environment promotes the introduction of the methodology, the linked environment favors an interactive process for learning about the tool, and the integrated environment leads the new users to autonomous control of Schemacode.

INTRODUCTION

This paper describes an experimental tool and its automation. The tool links together the manager and the programmer on the development of a software project. It simply provides a way for initiating feedback at every level. This tool is fully interactive, since the software system design is arrived at only after several trial and error attempts at every level of design.

The essence of an automatic programming system is that it assumes responsibilities otherwise borne by a human being and thereby reduces the human task and makes it more manageable.

Software system developers need a change in their programming environment that will relieve them of at least some of the tasks that they currently must perform. It is the goal of automatic programming research to effect this change by transferring responsibility for some segments of the programming process from the human programmer to an automated computer-based system.

Faced with a new tool, and usually a corresponding new methodology, the personnel, hardware, and software must also adapt or change accordingly; personnel must adjust to the new products, existing structures and division of labor must be modified, new hardware must be purchased, existing software must be modified, and scheduling must be changed.[1]

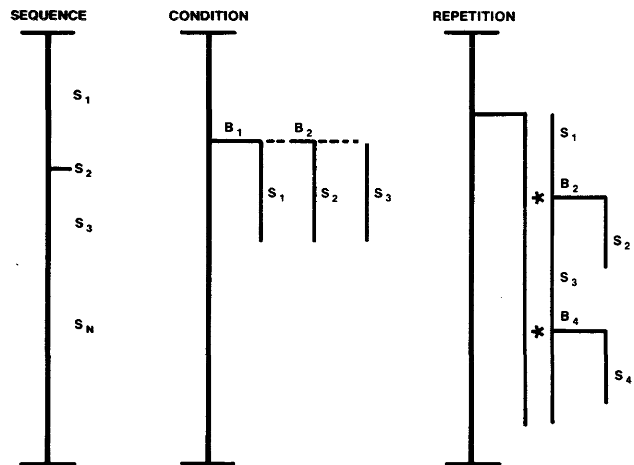
A new interactive programming tool must include features for facilitating its integration into new environments. The introduction of Schemacode, based on Schematic Pseudocode, a methodology easily understood by managers as well as programmers and analysts, is effected in three distinct phases, each of which is a necessary stage in the transition process: (1) creation of a *virtual environment,* (2) creation of a *linked environment,* and (3) creation of an *integrated environment.* This gradual approach allows for a thorough training in the methodology, for the carrying out of various tests, and for a complete clarification of the tool specifications which the user requires in a finished product that meets all his specific needs.

SCHEMATIC PSEUDOCODE

One of the key elements of the methodology is the scheme used for communication between the manager and the programmer. Graphics, words, and codes are widely accepted as the basic elements of communication. Graphics alone do not carry enough readily accessible information. Words and codes portray the structure of the project poorly. Programmers do not like the documentation task, and computing managers who are most responsible for the quality of software dislike programming details. On the other hand, the unique property of word-graphics type of communication can satisfy the needs of both the manager and the programmer.

With Schematic Pseudocode, the graphic symbolism is exclusively designed to represent the structure of the process while the words are used to express the "activity" of the structure. All the graphic symbols are shown in Figure 1.

Schematic Pseudocode is above all else a pseudocode that expresses itself through graphic language.[2] When solving problems, it is often useful to be able to visualize the structure of an algorithm. There are several existing techniques for providing graphic translations of algorithms, but most of these techniques are applicable only after the creation of the algorithm.[3–5] Such graphic representations reflect, as a result, the programmer-analyst's competence in solving his problem rather than the intrinsic structure of the problem. Schematic Pseudocode, on the other hand, provides graphic representation throughout the development of a program. Schemacode is the tool that allows for the direct input of Schematic Pseudocode at a terminal and for the obtaining of the formal code in FORTRAN, PASCAL, COBOL or other languages.[6]



Schematic Pseudocode (SPC) is based upon a conventional nomenclature where each program can be expressed in terms of actions which can be sequential, conditional or repetitive. We use S for statement and B for boolean expression. a) Statement which begins with a dash (sequence S2) serves for descriptive documentation statements. Statements which are not preceeded by a dash are assignments, declarations procedures or general comments S1, S3, ... Sn). The general comment identifies in natural language the actions to be refined. b) SPC uses the same basic representation for the SELECT, the CASE and the IF statement. In general, a conditional statement may contain a sequence of statements in each branch (S1) and may contain an arbitrary number of ELSIF (linked by dotted lines) on the statements. c) SPC uses an unique diagram form which is a general loop with multiple exits. Double vertical lines show the scope of the loop and stars indicate the exits.

Figure 1—Schematic pseudocode

Before going further into the tool description, let us look at the methodology behind Schemacode. The introductory example is an algorithm called "update sequential file processing."[7]

SEQUENTIAL FILE PROCESSING

Using Schematic Pseudocode, a general algorithm for file updates is derived. The purpose of this example is to illustrate the ease of use of Schemacode in the design of a complex algorithm and its usefulness as a documentation tool.

In this example, we present the basic operations that are performed when processing a file sequentially. The algorithm uses three sequential files:

1. The MASTER file (M) consisting of records m_r, which are sorted in an ascending sequence according to the value of the key m_k.
2. The TRANSACTION file (T) consisting of records t_r. The records are also sorted in an ascending sequence according to the value of the key t_k. Each record t_r contains two parts: an operation code t_{op} and the data t_d (incl. t_k). The code has one of the three possible values: (1) 'D' deletes from the file M the record m_r for which $m_k = t_k$; (2) 'I' inserts t_d in the file M; and (3) 'U' updates in the file M the record m_r for which $m_k = t_k$.
3. The NEW MASTER file (N) consisting of records n_r similar to those of the file M and sorted in an ascending sequence according to the value of the key n_k.

No duplicate keys are allowed in the file, and the algorithm gives an error message when a transaction tries to insert an existing record or to delete or update a nonexisting record.

A step-wise refinement approach to this problem requires 8 steps, which follow.

A top-down approach is used here; each of the actions present can be further defined by step-wise refinement. These actions are

EOF (T)	END OF FILE ON (T)
PROCESS 1 KEY	
EOF (M)	END OF FILE ON (M)

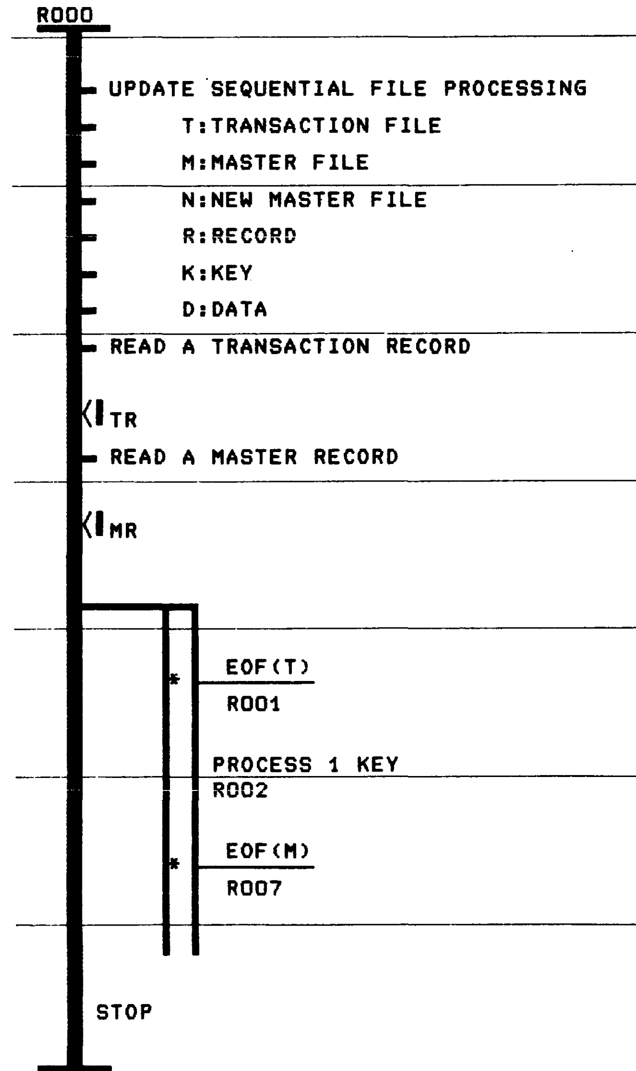
(See Figure 2.)

This algorithm also includes lines that begin with a dash. These lines are called comment statements. They provide an English description of the algorithm to help the person understand the significance of the action to be executed or to be subsequently refined.

The Schematic-Pseudocode feature designed for specifying data values is a triangle pointing toward the algorithm. In the example, TR is a variable for which value is required. The triangle pointing away from the algorithm indicates values that are to be recorded.

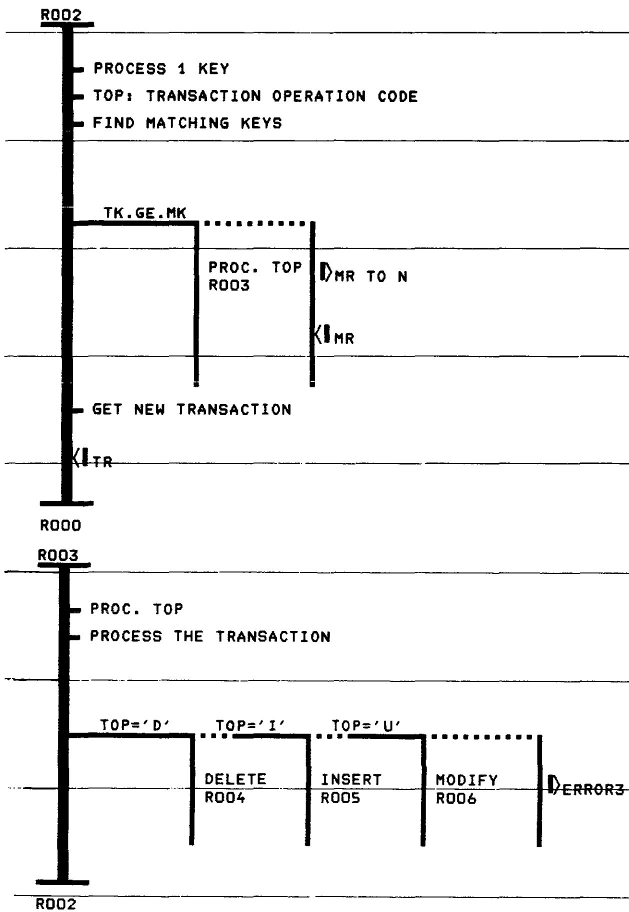
Repetition in some form is necessary. The Schematic Pseudocode statement for this action is the double vertical lines that show the scope of the loop. The star indicates the exit. In this case, the loop is designed to repeat the processing until the end of one of the files T or M. Within the loop, every key is processed. Each of the actions is subsequently refined as shown in Figures 3–6. Conditional statements allow choice between alternative courses of action. For example, in Refinement r-003, the value of TOP (transaction operation code) is compared with the character 'D,' 'I,' 'U,' or ELSE. If 'TOP' is equal to 'D,' the action DELETE is executed. That action is defined in Refinement R-004.

All the refined steps are integrated to form the final version of the update file processing. The Schematic Pseudocode represents the control skeleton of the program, i.e., its structure as well as its flow. As can be seen from this example, the Schematic Pseudocode is more than an easy-to-learn language; it is a design methodology that can be used by managers and programmers in the development of software. The Schematic Pseudocode is a pictorial representation that serves



Three refinements are required to define this step. They are described in steps R001, R002, and R007.

Figure 2—Schemacode output of Refinement



(a) Step R002 describes how PROCESS 1 KEY is performed. A match is tested between the transaction key (TK) and the master key (MK), if it is succeeded the transaction operation code will be processed (PROC.TOP), such a processing is done in Refinement (R003). If the match is not performed the master record is written onto a new file and a new master record (MR) is read.
The R000 printed at the bottom of the main vertical line indicates that the process resumed in refinement R000.

(b) Step R003 describes in more details how the processing of the operation code is performed.
To do so 3 more refinements are required.
R004 to process the DELETE of a transaction
R005 to process the INSERT of a transaction and
R006 to process the MODIFY of a transaction.
Otherwise an error 3 message is printed.
Note that after execution of this refinement the process continues into refinement #2 as described by the R002 at the bottom of the printed output.
All this labelling is done automatically by Schemacode.

Figure 3—Schemacode output of Refinement 2(a) and Refinement 3(b)

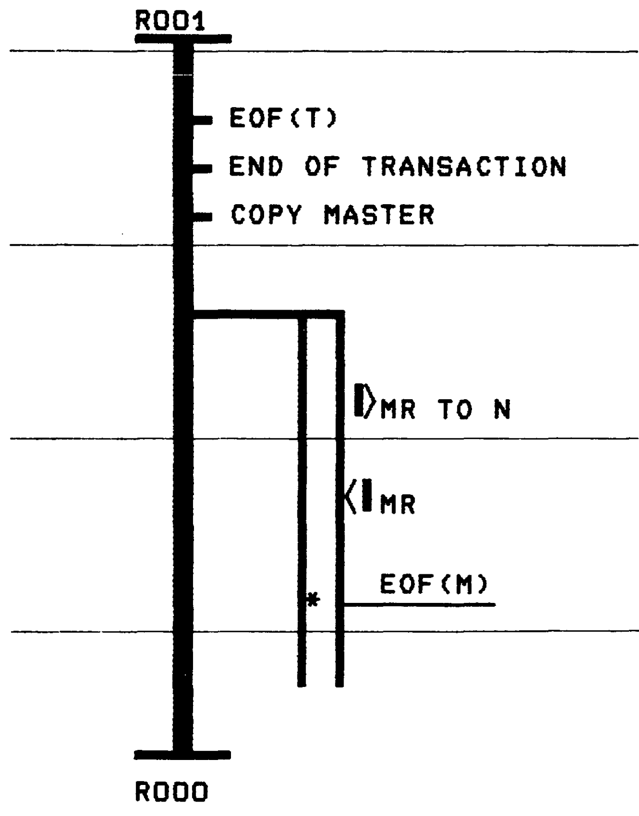
SCHEMACODE

Schemacode is a software package running on an IBM 4341. The terminal (VT-100 compatible) makes it interactive. The primary task of Schemacode is to assist users in the development, documentation, and structured coding of programs.[8] It usually transmits the source program to the main computer for execution. In the development phase of a project, the Schematic Pseudocode is output at the graphic printer, whereas in the coding phase, a structured listing can be output.[6,9]

The graphic structure of the Schematic Pseudocode constitutes a very-high-level language. It is entered to Schemacode via special keys or commands on the keyboard. Several control inputs are also used to specify the type of operation and the detail associated with it.

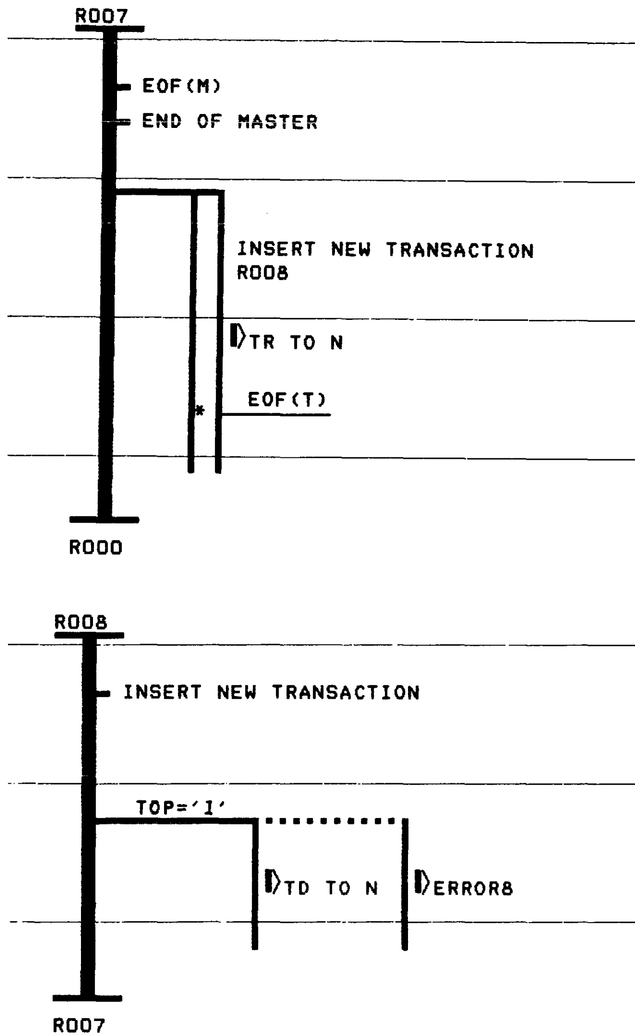
When the desired level of development is reached, Schemacode will automatically integrate all the steps and provide a complete chart of the step-wise refinement process. The user can recall any refinement, redraw it, or modify it. All modifications to structures or comments will be automatically integrated.

Two types of output can be provided by Schemacode. At each step of refinement, a graphical output can be printed. The graphic symbolism of the Schematic Pseudocode is used as a means of recording, analyzing, developing, and communicating program information. Throughout these steps, emphasis can thus be put on human communication. The Schematic Pseudocode is a universal means for communicating software development even if you have been educated in FORTRAN, PASCAL, COBOL, or other languages.



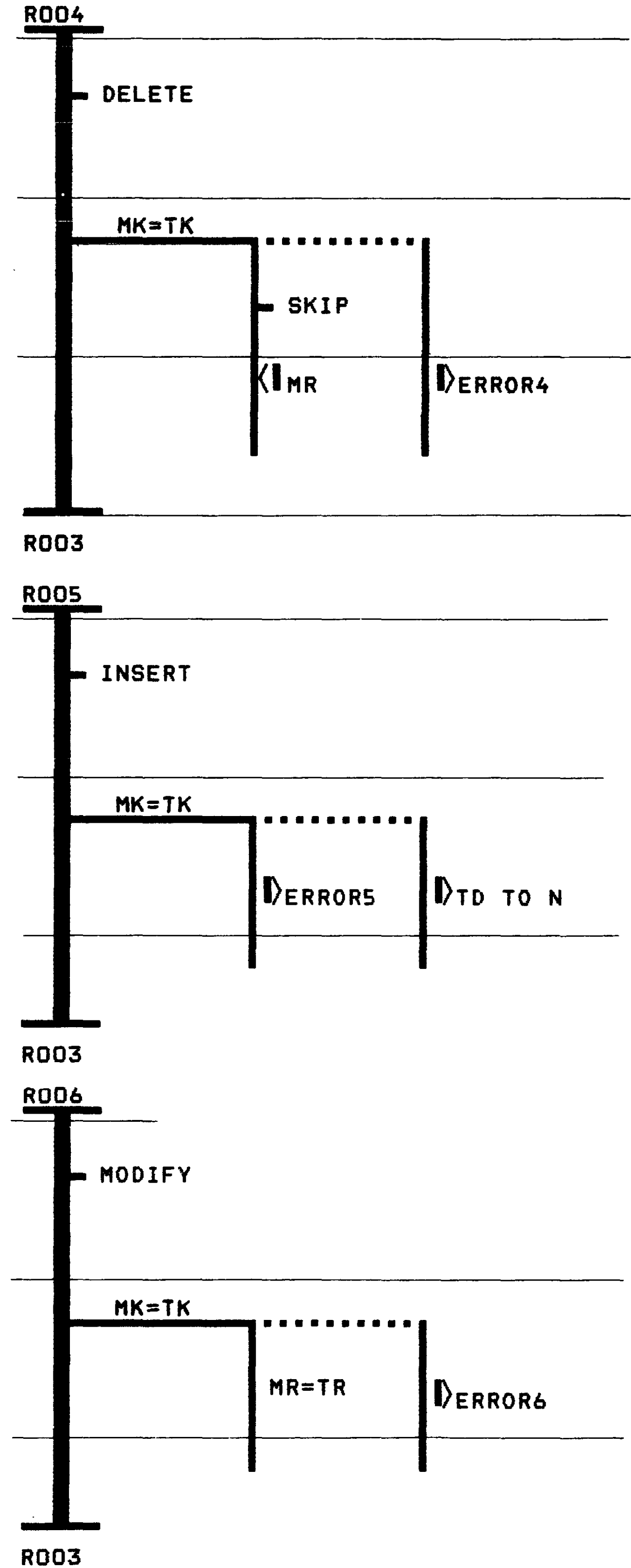
This refinement describes the copying of the remaining master records into the new file.

Figure 4—Schemacode output of refinement 1



a) This refinement describes the insertion of the remaining transaction files into the New File once the end of the master file is reached.
b) The insertion of the new transaction. An error 8 message is printed when the transaction operation code (TOP) is not of the INSERT (I) type.

Figure 5—Schemacode output of Refinement 7(a) and Refinement 8(b)



These 3 refinements describe the action to be taken when a transaction is to be DELETED, INSERTED or MODIFIED. They are set in refinement #3 (see Fig. 3).

Figure 6—Schemacode output of Refinements 4, 5, and 6.

exclusively to represent the control structure of the process, whereas words in natural language are used to describe statements and Boolean expressions.

When the process is pursued to the code level, a structured listing can be output in a selected language with labeling and paragraphing done automatically. All the comments specified during the development phase are automatically integrated into the listing. Boolean expressions, variables, formats, etc. must be specified in the formal language before editing. As one can see, a real advantage provided by Schemacode is that every program developed has a unique up-to-date documentation and listing. This source program can then be transmitted to the main computer for execution.

The use of a specialized language appears only at the end of the development, when the problem is almost solved. At this time, a programmer specialized in a formal language has to

translate the Boolean expressions, and define the format, the variables, etc. to allow Schemacode to edit the listing in the proper language.

SCHEMACODE IMPLEMENTATION

To demonstrate exactly how the technological transfers take place, let us examine each of the three stages of tool implementation. The discussion will show the gradual development of a small group of programmers, analysts, and managers towards autonomy in their use of Schemacode.

Virtual Environment

Schemacode creates a preliminary environment based on manual use of Schematic Pseudocode (SPC); we have named this first phase the virtual environment. This stage begins as soon as the user manifests an interest in the tool and ends when all personnel involved in using the tool have completely mastered the concepts of Schematic Pseudocode.

The virtual environment is created with the aim of allowing a gradual introduction of the methodology supported by Schemacode. The idea is to disturb work already in progress as little as possible,[10,11,12] while at the same time including in the virtual environment all personnel affected by the use of the new methodology.

Those responsible for the quality of the software—managers, analysts, and programmers—must ascertain that they have a thorough understanding of all aspects of Schematic Pseudocode. To this end, the team to be initiated into the new methodology undertakes a simple project under the direction of competent personnel. An ideal project for this purpose consists of rewriting in SPC a subprogram that the team has previously worked out using conventional methodology.

Before launching the chosen project, however, the team receives a formal introduction to the new tool and obtains all the documentation necessary for the use of Schemacode.[2,6,9,13] This preliminary stage of the virtual environment takes about two hours and should be carried out in a group of about six people that includes a representative from each level of activity—programmer, analyst and manager. The aim of this session is to introduce Schematic Pseudocode, Schemacode's methodology based on structured programming with a top-down approach and step-wise refinement.

The Linked Environment

Using Schemacode in a linked environment constitutes the second stage in the transplantation of the tool. This stage consists of using the Schemacode software, in its IBM version, at a terminal connected to our central computer by a packet-switching network of the Datapack or Telenet type. During this period, the users become familiar with the commands and the real possibilities of the tool. They become able to specify exactly which input and output options they would like to see added to Schemacode to create a modified tool that will adapt most readily to the third phase of the transplantation plan, the integrated environment. This second stage ends when all new specifications have been defined, developed, and tested to the satisfaction of the users on the IBM version of the tool.

The aim of the linked environment is to allow the user to understand the full potential of Schemacode. Schemacode's primary task is to assist in the development, documentation, and structured coding of source programs, which are then transmitted to the main computer for execution.

Schemacode constructs, according to the rules of Schematic Pseudocode, an aggregate file developed from the different refinements specified by the user. This software also checks for basic errors in the structures entered. As a result of discussions which crop up among the team members during the virtual stage, this module can be modified slightly to adjust to the technical peculiarities of the third phase of implementation. The two output modules provide for the reproduction of printouts of the complete documentation of the program developed: an alphagraphic representation of the logical structure of the different refinements of a program as developed by the user at the terminal and a coded version produced by Schemacode, in FORTRAN, from the control structures and the comments specified during program development.

Exercises such as those described allow the team members to evaluate the input and output modules in terms of their own particular work environment, as well as allowing each group member to develop his competence with Schemacode without disturbing the existing hierarchical structure of the work milieu. The linked environment stage thus allows the tool builders to clearly define the software and hardware specifications of the input and output features of Schemacode as it will appear in the third stage, the integrated environment. The tool builders can thus adapt Schemacode's input and output modules to the particular type of hardware that they already possess or that they intend to buy. As each specific option is added to the tool, the team can test and evaluate it. The give and take among the team members and the tool means that there will be a clear evaluation of the needs of the group and minimizes the costs of transition to the integrated environment.

The Integrated Environment

The integrated environment is the stage at which Schemacode is fully functional. At this stage the software and all its component parts have been completely adapted to the needs of the particular user and the hardware that the user possesses, including terminals, printer, and files. This environment stabilizes when all technical and administrative difficulties have been overcome. It is noteworthy that the preparatory phase of the integrated environment is complete when the linked environment has become fully functional. Thus, the transition from the linked to the integrated environment, however major a change for the firm as a whole, does not normally entail modifications in the work habits and relationships of the team adopting Schemacode.

The integrated environment can be effected in one of two ways, through the use of internal or of external resources. In the latter case, the team makes use of an outside computer, much as in the linked environment, with the difference that the Schemacode software is now perfectly adjusted to the new

user's particular needs. However, even with the use of external resources, a program generated by Schemacode can be executed on the firm's internal computer.

Internal resources are divided into the following categories: (1) main systems and (2) work stations. The new user can decide to adapt Schemacode to his own particular computer and to connect this computer to different terminals, or he can give each programmer-analyst a work station. In the latter case, Schemacode is implanted in a microcomputer, and from his work station, the programmer can conceive and analyze his programming problem, obtain the code, and then transmit the program to the main computer for execution.

The advantages of using internal resources are significant if a computer is reserved exclusively for the development of software. In other cases, the user's needs may require buying an additional computer capable of handling the concept of interactive software development. No matter what type of resources are used for the creation of the integrated environment, as far as the development of the software, the result is the same: Schemacode ensures uniformity. All documentation is automatically incorporated into the code and demanded for each refinement step.

The adaptation and adoption of Schemacode are effected only with continual feedback between these two groups. Figure 7 provides a résumé of the kind of feedback for each stage —virtual environment, linked environment and integrated environment involved in the transition to Schemacode.

CONCLUDING REMARKS

The Schematic Pseudocode is easy to draw by hand. However, one major interest of this methodology is its automation by an interactive tool named Schemacode. Using the Schematic Pseudocode, Schemacode assists designers in successively refining the solution into further details. Schemacode performs its task by keeping track of all the processes involved and making sure that they are properly done, integrated, and documented. The process can go as far down as required by the user, possibly to the code level.

This article describes how Schemacode progressively modifies the environment in which software is developed to gradually produce a completely automated environment for the development of software. Schemacode is specifically designed for adaptation to, rather than imposition on, a new milieu; and the adaptation is accomplished by the work of the new team adopting Schemacode.

The success of a tool depends on several factors: the tool's flexibility and adaptability, the environment, the tool's acceptability to the team members, etc. It is not enough for a manager to want a tool; the tool must be gradually accepted and accepted with enthusiasm by the team affected. The implantation of a tool must modify the existing design environment as little as possible and must adapt this environment to the requirements of the tool.

The development of software usually proceeds under conditions that are severely restricted in terms of money and time. A design tool can hardly be introduced effectively when it serves only to contribute to the already existing confusion and tension brought on by the constraints of time and money and, furthermore, ends in confusing the documentation. This is why particular attention must be paid to the implantation of a new tool and to the elaboration of its functional environment. Careful research is required to better evaluate the effects of tool implantation into any given environment. The project and its preliminary results, presented in this article, do not pretend to solve the problem at hand, but they can perhaps provide some guidelines when considering the modification of environment.

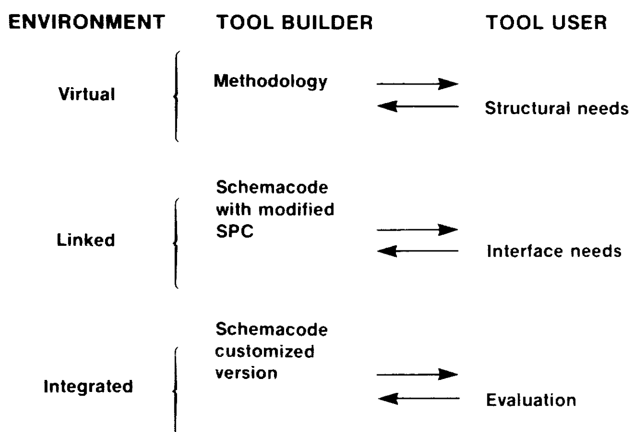


Figure 7—Resume of the feedback between the tool builder and the tool user for every step of the tool implementation

ACKNOWLEDGMENTS

The plan for integration proposed in this article is in fact an experiment now in progress in the industrial milieu of Canadian Marconi Corporation. The first and the second stages in the integration-transplantation plan—the virtual and the linked environments—have been successfully completed, and the creation of the integrated environment is in progress.

We would like to thank Mr. Tony Murphin, the manager in charge of software quality control, and the analyst, Mr. Roger Gauthier, for their unconditional support.

The projects described above were realized with the collaboration of Ecole Polytechnique de Montréal and were partially financed with the help of subsidies from the governments of Quebec (FCAC grant No. EQ-1727) and Canada (CRSNG No. RD-0119).

REFERENCES

1. Panel Session, G-2, "How Practical are Today's Software Engineering Methodologies." *5th International Conference on Software Engineering,* San Diego, CA, March 1981.
2. Robillard, P.-N., and R. Plamondon. "An Interactive Tool for Narrative, Operational and Structural Documentation." *Proceedings of 23rd IEEE Computer Society International Conference* (COMPCON 81), Washington D.C., Sept. 1981, pp. 291–295.
3. Miller, E., "Tutorial: Automated Tools for Software Engineering." *IEEE Computer Society,* IEEE Catalog No. EHO-150-3.
4. Freeman, P., Wasserman, A. I. "Tutorial: Software Design Techniques." *IEEE Computer Society,* 2nd Ed., Catalog No. 76CH 1145-2C.
5. Belady, I. A., C. J. Evangeliste, and L. R. Power. "Greenprint: Graphic Representation of Structured Programs." *IBM Syst. Journal,* Vol. 19, no. 4, 1980, pp. 542–553.

6. Plamondon, R., and P.-N. Robillard. "Harness a Computer to Write Better Software, Faster." *Electronic Design,* 29 (1981), pp. 125–129.
7. Robillard, P.-N., and D. Thalmann. "Complex Problem Solving Using Schematic Pseudocode." IRO Université de Montréal, Publication No. 373, Nov. 1980, 17 pp.
8. Robillard, P.-N., and R. Plamondon. "Schemacode: An Interactive Schematic Pseudocode for Program Development, Documentation and Structured Coding." *Proceedings of Tool Fair, 5th International Conference on Software Engineering,* San Diego, CA, March 1981, pp. 161–169.
9. Robillard, P.-N., and R. Plamondon. "Introduction to Schemacode." EP-81R-20, École Polytechnique de Montréal.
10. Riddle, W. E. Software Development Environments, Tutorial on Software System Design Description and Analysis, Computer Society Press, IEEE cat. EHO-166-9, 1980, pp. 210–219.
11. Riddle, W. E. et al. "Behavior Modelling During Software Design." *IEEE Trans. on Software Engineering,* Vol. SE-4, (1978), pp. 283–292.
12. Basili, V., and R. W. Reither, Jr. "A Controlled Experiment Quantitatively Comparing Software Development Approaches." *IEEE Trans. on Software Engineering,* Vol. S4-7-N.3, May 1981, pp. 299–320.
13. Robillard, P.-N. "Programming from Semantic Knowledge." *3rd World Conference on Computers in Education,* Lausanne, Switzerland, July 27–31, 1981.

Distributed processing of problem-solving applications for farmers

by ROBERT GAMMILL and LYNN THORP
North Dakota State University
Fargo, North Dakota

ABSTRACT

Intercomputer data communication via modem and telephone promises to have an important impact on rural areas. Unattended and inexpensive late-night communication has the potential for providing routine, noninterruptive, and rapid information interchange. An application of intercomputer communication to reduce the cost of using remote timesharing services has been experimentally developed at North Dakota State University (NDSU). An agricultural problem-solving program from the AGNET system in Nebraska has been split into three parts: interactive input of user data, computation of results, and production of an output report. The interactive input is then carried out on a computer near the user, eliminating interactive long-distance communication costs. The data collected are transmitted to the AGNET timesharing machine for calculation. The transmission, including login, is rapidly carried out by the local machine, using a protocol interpreter developed at NDSU. The results of the calculations are then transmitted back in compact form to the user's machine, where they are formatted and displayed. Only a minute of connect time to AGNET is required instead of half an hour. The protocol interpreter used to do this is described in some detail. Such splitting of functions between computers promises to allow high-quality problem-solving services to be provided on central machines, with personal or local machines providing interactive input and printed output services.

INTRODUCTION

Rural areas have a high variance in quality of life. Despite positive attributes such as quiet, clean air, woods, lakes, slower pace, and other amenities, there are numerous negative attributes stemming from isolation, such as a limited number of stimulating jobs; reduced access to educational, cultural, and entertainment opportunities; reduced peer interaction; and the lack of other positive attributes of urban life. Modern communications media, such as telephone, mail, newspapers, magazines, radio, and television, have helped; but there is still a lack of rapid, inexpensive, convenient, noninterruptive two-way communication facilities for rural use. Computer-controlled telephones promise to remedy this situation in the near future. Microcomputer-controlled data communication over telephone lines should produce a major improvement in quality of life in rural areas.

This paper concerns one use of computer-controlled data communication and describes research being carried out at North Dakota State University to provide inexpensive access to farm-related problem-solving and management computer services. The work is part of a project called RAIN (Rural Agricultural Information Network), sponsored jointly by the North Dakota State University Agricultural Experiment Station and the Cooperative Extension Service. The research uses adaptive software[10] that allows a computer running UNIX[6,13] to login on another computer and run programs on that machine as if the UNIX system were an ordinary timesharing user, but at much greater speed than is possible for a human user. This capability allows the UNIX system to communicate with computers that are unprepared to communicate with it! In a rural environment, where most computers will be owned by individuals, such capabilities are likely to be critical.

The goal of the present work is to demonstrate the feasibility of distributing the processing of farm-oriented computer services through such means. North Dakota farmers have access to AGNET, a timeshared computer service for generating solutions to farm management problems. A limit to the ultimate utility of AGNET to North Dakota farmers is its location in Lincoln, Nebraska, since long-distance telephone costs limit how much use a farmer can make of the system. The research demonstrates that a data sheet can be collected near the farmer, either on a personal computer or on a nearby timesharing system, and then transmitted through the telephone network from computer to computer. This eliminates the costly time delays involved in human typing over long-distance telephone lines and also allows the transmission to be scheduled late at night, when long-distance charges are minimal. Results are subsequently transmitted back to the machine near the farmer for examination or printing. This method is related to the concept of a network operating system, where processes may be scheduled on geographically distributed processors; but here the distribution of function is being carried out at the application level by a network terminal "agent" working like a postman.

RURAL COMMUNICATION

Considerable attention has been focused recently on the possibility of saving energy by using communication in place of transportation. For example, instead of traveling a salesperson might make sales by telephone. In rural areas where distances are great, not only energy but human resources are at stake, since travel time reduces time available for education, recreation, or business. At present, rural areas depend primarily on mail service, telephone, and the media (e.g., television and publication) for communication. Although the media provide a great deal of general information, specialized information desired by an individual can be difficult to obtain quickly and cheaply. Most limiting for the media is the one-way direction of the communication. For two-way communication the rural resident must depend upon telephone or mail. Mail has the advantage of being noninterruptive, but it is very slow. The telephone, though rapid, is interruptive and requires that both parties to the conversation be at their telephones. In the rural environment where much work is done outdoors and long travel times can keep people from phones, the telephone's usefulness is reduced. A result has been that some farm wives, who remain closer to the house, have taken on the role of communications expediter. The computer-controlled modem and telephone promise to combine many good features of telephone and mail and to provide new opportunities and capabilities in rural areas. The promise stems from their noninterruptive nature (like mail) and their ability to minimize costs by automatically dialing up another machine at low-cost times (when humans are asleep), or by maximizing speed of outgoing information (at higher cost) by making an immediate connection and data transfer. No matter when data is sent, it can go at a rate of 300 to 1,200 words per minute—much faster than even the fastest person can type. With improving technology, even greater speeds are possible.

The particular work to be described in this paper is the extension of a farm-management-oriented computer timesharing system called AGNET. It is extended by allowing some of its communications-intensive functions to be carried out on computers near the user, in the hope of substantially reducing communication costs. This research concentrates on ways in which this redistribution of the functions of AGNET

programs can be done with minimal changes in existing AGNET software on the central system to minimize the cost of changing over to this new technique. We begin by examining several current agricultural timesharing systems.

CURRENTLY AVAILABLE AGRICULTURAL TIMESHARING SERVICES

In the past few years several computer services have been developed especially for the agricultural community. Some are designed for a very limited audience. For example, HAMS (Hog Accelerated Marketing System) is designed to market hogs. Both buyers and sellers input information about animals available for sale and prices dealers are willing to pay. The seller may request a firm deal from a buyer or put up his lots for auction. The auctioning or acceptance and rejection of firm deals are all completed through the computer. A similar service is available to cattlemen in Texas through CATTLEX, a program funded by the United States Department of Agriculture. Producers will list cattle for sale and buyers will then have access to this list.[15,16,17]

Other computer systems have been designed to be more multipurpose. These services usually provide a variety of programs relating to agriculture, such as budgeting, crop planning, marketing ideas, and management. The most widely known of these services are FACTS (Fast Agricultural Communications Terminal System) at Purdue University, TELPLAN at Michigan State University, and AGNET, a regional facility in Lincoln, Nebraska.[5,11]

AGNET

AGNET is a regional AGricultural NETwork computer system headquartered in Lincoln, Nebraska, which serves mainly the states of Nebraska, North Dakota, South Dakota, Wyoming, Montana, and Washington, with users from several other states and countries. AGNET's primary purpose is as an information delivery system, information ranging from mail between users, current market information, and current news clippings to information derived from problem-solving programs. The major users of AGNET are farmers, homemakers, Cooperative Extension Service personnel and specialists, adult vocational education instructors, agricultural researchers, and students in home economics and agriculturally related areas. At the present time 35 to 40% of North Dakota AGNET usage is from students, mainly at North Dakota State University. Originally funded as an Old West Regional Commission project, the project now receives support from member states. In North Dakota part of this support comes from the state legislature in the form of budget money for the Cooperative Extension Service. The Extension Service now supports the use of AGNET by people involved in research, teaching, and adult education programs. External users (people not associated directly with the university or Extension Service) pay a fee that covers the direct cost of their computer use. In other states external users' fees are used to support the rest of the system.

AGNET has a wide variety of programs to serve many agricultural needs. Some of these are simulations of real systems, such as grain drying and feeder cattle performance. Other programs help the user to figure out the best way to solve problems, such as the right feed mix, irrigation scheduling, the best crop to plant under certain conditions, or the feasibility of buying more land or machinery. General farm planning, estate planning, and tax management programs are also available. These problem-solving programs are designed to guide the user to a possible solution by asking questions to gather relevant information. The user can see the result of one set of data, change several answers, and see the new outcome.

Besides problem-solving programs, AGNET also provides services such as news clippings of importance, lists of buyers and sellers for hay and other commodities, and other general information useful to the agricultural community. A mailing system is also available to send messages to specialists, state programmers, and other users.

Currently the most widely used programs fall in the areas of market information, financial packages, and communications, which includes mail, listings of magazine and newspaper articles, and conferencing. A problem-solving program that is used widely is FEEDMIX, which calculates the least-cost feed mix for certain nutritional requirements.[1]

Its interactive nature and quick response to the "what if" type of question help to make AGNET a well-used and rapidly growing service. There are now about 2,100 user numbers available for the whole system; about 215 of them are in North Dakota. The number of user IDs, however, is not a true indicator of the number of users, since several of these numbers are general-purpose numbers available to a large number of people, especially students. In North Dakota alone, the use both in total number of logins and average login session length increased by 35% in fiscal year 1980 and by 41% in fiscal year 1981. The average user in North Dakota is logged on for about 35 minutes and typically runs two or three packages, besides receiving mail. In other states, such as Wyoming, the major use is for communication through mail, which makes the average login time shorter, although the number of logins is about the same.[1]

PROBLEMS WITH CURRENT TIMESHARING SYSTEMS

With the increasing use of the services provided by today's timesharing systems, including AGNET, several potential problem areas come into focus. As actual computing costs decrease, a larger part of a timesharing user's cost will be the communication cost to link to the system. An interactive timesharing session usually involves a large amount of instructions and information to be transmitted to the user. The user then types in a reply, and the program continues. However, the user normally must think awhile before typing in his or her response. Even when typing, few users are able to achieve 30 words per minute, or 10% of the available data rate. This means that the communication line is sitting idle for long periods of time, reducing the efficiency of line use to perhaps as low as 1%. At the normal rate of 300 baud a session can be very lengthy and expensive, especially with daytime telephone

rates. Besides being more costly to the user, a lengthy session ties up the resources of the timesharing and telephone systems. Such resources include telephone switching equipment, lines, and the computer's terminal port. Although the resources such as the ports are in use a large percentage of the time, this does not mean that they are being used efficiently. North Dakota currently has seven ports to Nebraska. Four of these are dedicated to users on the campus of North Dakota State University and external users. The other three are dedicated to the field staff. The presently available ports are usually filled, and there are plans to increase the number of ports and lines to Nebraska. Handling the increase in demand solely by adding new equipment without increasing the effective use of existing resources can be an expensive and short-term solution.

At a time when demand for services has increased and adding physical resources costs a great deal, increasing the efficiency of the use of existing resources is another way to maintain and possibly expand the current level of service. One way to increase the efficiency of the communication lines is to use computer-to-computer communication, which can increase the line use to nearly 100%.

Splitting an interactive session into parts and running the communication-intensive portions of the session on a home microcomputer, sending only a small amount of information over long-distance lines, allows this higher use of communication lines. A microcomputer is a logical choice for the communication-intensive tasks, for several reasons. First, many farmers now have or will soon have a personal computer to aid them in the record-keeping aspects of farming. Currently available software can handle these tasks, but very few agricultural-problem-solving programs are available for microcomputers. Second, many people are taking advantage of the ability of their microcomputer to act as a terminal for accessing the larger timesharing services that have this software available. The highly interactive nature of microcomputers is ideally suited to the data gathering necessary for the input worksheet. Downloading a program from a remote machine to the personal computer to allow this data gathering would cut the amount of connect time to long-distance facilities. Long-distance lines cost about $9 per hour late at night and $21 during working hours. At those prices a local computer to provide interaction and collect data for subsequent transmission can quickly pay for itself.

One further point should be made about computer-to-computer communication, as opposed to timesharing. Not only does intercomputer communication reduce costs to the computer user; it should also be a boon to telephone companies and to other residential telephone users. Present use of the telephone system by computer terminals connected for hours to timeshared computers has clogged local exchanges and encouraged movement toward metered local service, since earlier tariffs were based on an assumption of 3- to 5-minute calls. The computer-to-computer communication we describe will operate in short bursts and be most economical and nearly as convenient for use at night, when voice traffic is minimal. It will generate additional telephone revenue at low load times, require no additional facilities, and increase the efficiency of the rural telephone system. This load leveling should have the effect of lowering residential phone rates, an important result in inflationary times.

THE COMPUTER-CONTROLLED MODEM

Most work on computer-controlled modems until now has focused on microcomputers using assembly language. Although UNIX running on a PDP11/45 cannot be viewed as a personal computer system, the wide use of UNIX on the next generation of 16-bit microcomputers makes it likely that it soon will be viewed in that way. The powerful programmer's environment of UNIX and its higher-level language C, in which UNIX is written, makes it an ideal system in which to do software research. As a result, we are using a PDP11/45 running UNIX, a Universal Data Systems (UDS) 103J-ACU modem with auto-dialer, and one port of an ABLE DMAX-16 (DEC DH11 equivalent) terminal multiplexer as the hardware support for our research. At a later date we plan to test the software on much less expensive equipment, namely a PDP11/23 running UNIX, a DZV11 terminal multiplexer, and an autodialer modem, such as the Hayes Smart Modem.

The goal of the research is not only to establish connections between computers using the dial-up telephone network, but to allow the computer to establish and use those connections without any human intervention so that the benefits of late-night operation and low rates can be realized. In addition, we are developing protocols that permit the "other" computer to be completely unprepared for computer-to-computer communication. The goal is to use the normal timesharing login mode of many machines as an access method and program our machine to be so "smart" that it can act like a human user of those machines. Clearly there are advantages and disadvantages to this goal. The data rates achievable are limited, but most critical is the requirement for having a different protocol for every computer with which interaction will take place. This places upon us the requirement to make the software interpretive and to provide compact definitions of the interaction protocols for the "other" machines. We are well aware that if N machines are to communicate through login on one another, this could require as many as $N*(N-1)$ protocols. However, we are optimistic that should wide use of this approach develop, the volume of protocols will provide a strong incentive for standards among the machines which could be implemented and tested using our interactive protocol interpreter. Despite some obvious problems in our approach, it has even more important advantages. Most existing network protocols demand that every participating computer "learn the language." On the ARPANET every host must have an IMP interfaced to it and the host-IMP protocol must be implemented for that host. Thus, any computer where the cost of an IMP and implementation of the host-IMP protocol is out of reach is precluded from participation. Rural communication using personal computers probably cannot accept such stringent requirements for entry into the "club." Therefore, we have decided to work on the interactive protocol interpreter to see just how far such an approach can be pushed. The payoff is that only "our" machine need be smart, and by distributing such smart machines around the countryside to

CALL	*MEANING*
$0'abody'	Defines $a to mean body (macro definition).
$0=r015	Defines $r to be the character 15 octal.
$1	End of line mark (allows comments etc.). Normal end of line causes transmission of all preceding text to the remote system.
$2241007	Set DH11 mode (241) and speed (007).
$3020A	Set timeout on no characters returned to 16 (20 octal) seconds and if so take action specified by A, where: A ::=n next command is to be issued. a abort entire session
$4'% 'A	Specify response string (e.g., UNIX prompt) and if found do action A (as above). More than one response-action pair may be defined.
$5001	Controls echoing into local transcript file. 000 echo everything (for half duplex). 001 no echo (for full duplex remotes and password input). 002 do not even provide local prompts.
$6	Flush out present buffer contents (no return character) but do not wait for a response.
$7001	Set pause parameter to one second. This controls the time between response and sending of the next command for half-duplex systems.
$8	Flush out present buffer contents (no return character) and then wait for a response.

Figure 1—Macro primitives for the protocol interpreter

act as postmen and public servants, we believe we can supply unparalleled communication and information services to rural people.

THE INTERACTIVE PROTOCOL INTERPRETER

The basis for the protocol interpreter was the RITA language developed at The Rand Corporation.[2,4,14] That language was used to control remote systems through the ARPANET (including the Illiac IV and various IBM 370's) from a small computer running UNIX.[3] RITA was designed for artificial intelligence research in rule-directed systems; and although it provided superb facilities for deducing what to do in complicated situations, it tended to have too much overhead and was moderately inflexible for protocol interpretation, because its primary goals lay elsewhere. At North Dakota State University, under the RAIN Project, we are building a protocol interpreter for interactive systems that has *only* that goal. To do this we are using a macrolanguage[9] that allows embedding of new language constructs within the interactive command text as primitives and sequences of primitives. The purpose of these embedded primitives is to provide control, expected responses, synchronization, and timing information to the interpreter that transmits the surrounding text to the remote computer as commands. Figure 1 shows examples of experimental primitives developed so far.

This new macro language is based on GPMX[7] which was derived from GPM, a well-known macroprocessor. The critical feature of this macrolanguage is that because it uses a special escape character ($ here), it allows new operations to be embedded in an existing language. The existing language in this case is the command language of some remote timesharing system. Perhaps an example will make this concept clearer. In Figure 2 we show an example of a UNIX login session, but without any control primitives embedded. This is simply the text that a user would type, but with no indication of what responses or timing should be expected.

It should be noted that the commands of Figure 2 must all end with a RETURN character and that the two (CTRL-D) characters at the end result from holding down the CTRL key and hitting a D at the same time. This generates the ASCII end-of-transmission (EOT) character. It should also be mentioned that the 12 at the beginning is preceded by an empty line (containing only a RETURN character) and these two command lines are for the purpose of selecting service 12 through a Gandalf Private Automatic Computer Exchange (PACX). Figure 3 shows the same session, but now with the macro calls embedded to control timing and deal with responses in a correct manner. It should be emphasized that the command text has not been modified, except by the introduction of the embedded macro calls. Also, in every case the occurrence of an end of line (RETURN character) is a signal for the command text to be transmitted to the remote system and also signals that a response from the remote system (specified by a previous $4 macro) is expected. Finally, the $1 macro allows a move to a new line of text without such transmission.

A number of new issues can be observed in Figure 3. For example, the mail command gives no prompt or response, so a $s macro is issued, which sends off the text (adding a RETURN) but does not wait for a response. In addition, in the case of the (CTRL-D) or ASCII EOT characters, no RETURN character is needed, but a response is expected, so the $8 macro call is used following the $e, and both are defined as the body of a $q macro call.

We intend to expand the interactive protocol interpreter considerably beyond its present capabilities to make it much more flexible. Planned expansions include the insertion of the contents of input files in the command sequence in order to allow login names, passwords, and text (to be transmitted to a remote computer) to be provided by a user who need not know the contents of the surrounding protocol. In addition, we must provide the capability for the user to route certain segments of incoming text (from the remote computer) into local files or subsystems, e.g., mail. In addition to the two present options (Abort or Proceed to next command) on timeout or recognition of a response string, we also intend to implement a Go-to capability, allowing more complex routes to be taken through the protocol, depending on the responses of the remote system. Presently the protocol interpreter is an experimental system, but with these additions we believe it will be ready for preliminary operational testing.

One other facility of the UNIX (Version 6) system at North Dakota State University is critical to the effective operational

```
12
gammill
password
who
mail shapiro
Have you heard from John yet?
(CTRL-D) (CTRL-D)
```

Figure 2—Commands for a UNIX login session

```
$0=r015$0=e004$0'q$e$8'$5000$2341007000$4'*'n
$4'SERVICE 12 UNAVAILABLE$r$n'a$4'login: 'n$3007n12
$4'Password: 'n$5001gammill
$4'% 'npassword
$0's$r$6$1'who
mail shapiro$s
Have you heard from John yet?$s
$q$q$1
```

Figure 3—UNIX command text with embedded macro calls

use of the interactive protocol interpreter. This is the "remind" command, which was first implemented at The Rand Corporation by Dr. S. Zucker. This command allows the user to specify when a task should be executed, either in relative (e.g., 1 hour from now) or absolute (e.g., 3 a.m. May 15) time. If the task scheduled by "remind" is not successfully completed at that time, it can reschedule itself at a later time (using "remind"). Since it can continue to do this until success is achieved, it is possible to run the protocol interpreter unattended but with great assurance of success!

DISTRIBUTING THE TASKS OF AGNET PROGRAMS

As mentioned previously, the increased demand for the services provided by AGNET and the increased burden on the physical resources of the system necessitate alternatives to provide the same quality of service in a more efficient and less costly manner. One approach would take advantage of the present three-part structure of AGNET programs and the increasing presence of personal computers on the farm for business purposes by splitting the programs into input, computation, and output sections. This natural division of labor allows the individual tasks to be completed on different computers. For example, the input and output sections, both communication-intensive, could be run on the farmer's home computer, and the model calculations could take place on a larger remote machine.

The input portion of any problem-solving program gathers information needed to calculate the answers to the problem the program is trying to solve. In current AGNET programs this is accomplished through a series of questions to which the user responds, having the option of changing previous responses at any time. The questions are usually asked in the following way:

Enter: 1) Tractor Price
2) Tractor Useful Life
3) Trailer Price
4) Trailer Useful Life

which takes a substantial amount of time to print out at the normal communication rate of 300 baud. This method will be referred to as the typewriter mode.

The input function on a small machine has two possible implementations. The first would run as AGNET programs now do, with the questions being asked in typewriter format. This task is supported by several input routines, written in C, which are modeled after similar AGNET subroutines. This method would always have to be available, since many terminals use hardcopy instead of a CRT. An alternative method of input, called screen mode, would be video-oriented. Many questions would be on the screen at once, and boxes would be left for the user to fill in the answer. Hitting a key such as TAB would allow the user to move from one field to another. The screen mode program, designed to be similar to VisiCalc (TM—Personal Software, Inc.), would have the capability of checking for illegal or out-of-range data and giving appropriate help messages. A special advantage of screen mode is that default answers can be provided in the boxes. Therefore, only changes need be entered by the user.

Regardless of which method is used to gather the user's information, a problem-solving routine is needed to calculate results. For small applications it is likely that this task could also be done on a user's computer. However, some programs attempt to answer rather complicated questions. To accomplish this, many equations may have to be evaluated, and databases may need to be used. This task often requires more resources than are available on most microcomputers. Therefore, the use of a larger computer is often required. After the relevant information has been gathered on the small computer, it can be sent off to the larger one to be processed. Another reason to keep the model calculations on a central machine is to allow the program author to have control over the maintenance and modification of the program. This allows the integrity of the model calculations and the reliability of the results to be certified by the author. Such issues are important when model results may be used in expensive agricultural decisions.

The output section of the program takes the results calculated by the model section and prints it in a form easily understood by the user. This printout is usually in table or chart form, which may be very lengthy. At the usual communication speed of 300 baud, this is a time-consuming process, which can be expensive during daytime hours over long-distance lines. If the output process were available on the farmer's own home computer, the calculated results could be sent to it in compact form, the long-distance connection dropped, and the results printed out on the screen or a printer if available. A hardcopy listing is often very useful in order to evaluate a possible solution to a problem. Since printers are still quite expensive, it is possible that not every user would have one. A possible solution to this would be to have a printer shared by several users, possibly located at a County Extension office. The results could be seen on the screen of the farmer's computer and then be routed to the printer to be picked up later.

PROGRESS

A test program using a North-Dakota-written AGNET program, SEMITRUCK, has been set up to test the concept of splitting an AGNET program into three tasks and having the tasks run on separate machines. Each section was translated to C and implemented on the PDP 11/45. The input task was accomplished by a routine in typewriter mode and a screen mode routine that uses the Heath H-19 video terminal. First, the modules communicated with each other on the same machine, showing that the program could indeed be split. The

next step was to have the model calculations carried out on the AGNET system in Lincoln, Nebraska, with the input and output functions on the PDP 11/45 in Fargo. The AGNET SEMITRUCK program written in FORTRAN was slightly modified to bypass the normal input and output sections of the code. The input task was accomplished using the typewriter mode routines. These data were sent to the AGNET computer with the aid of the previously mentioned auto-dial modem and protocol interpreter. The model calculations were carried out and the results sent back to the output program on the PDP11/45. The output worksheet was printed by the local machine.

RELATIONSHIP TO NETWORK OPERATING SYSTEMS

Splitting a task into parts that are distributed over several computers interconnected by a network is one of the concerns of computer scientists studying network operating systems. The goal of a network operating system is to manage resources and tasks among a variety of computers hooked together by a network, allowing the computers to share those resources (such as databases or special hardware) for better performance. Such work is complicated, especially because the machines and their individual operating systems are often heterogeneous and not designed with cooperation with other machines in mind. This paper describes work which is definitely *not* on network operating systems but is related to it. The goal here has been to distribute parts of a task on different computers where the network involved is the dial-up long-distance telephone network. In some ways the methodology used has been much different from that of workers in network operating systems, because we have assumed that the only control that can be exercised over computers "other" than ours is through timeshared login on the remote computer. This has advantages and disadvantages. It means that achievable data rates are low and that the class of services available are simply those available to the ordinary user. However, it also means that the remote computer need not even know (or be modified to support) the fact that it is being used in a distributed processing environment. In other words, the present work might be viewed as distributed processing managed at the applications programming level. To make an analogy, overlays in FORTRAN were an applications level solution to the problem of limited memory, which is generally being solved by virtual memory operating systems today. We view our work as an applications level solution to distributed processing which will ultimately find more refined solution in network operating systems. Given the environment in which we are working, that solution is not yet feasible here.

SUMMARY

A project which seeks to distribute the processing of agricultural-problem-solving tasks over two or more computers has been described. The vehicle for the project has been the experimental use of a PDP11/45 running UNIX at North Dakota State University to split the processing of AGNET timesharing programs into local and remote parts. The goal of the work is the dramatic reduction of long-distance line charges for users of AGNET by allowing interaction to take place locally, while allowing the use of high-quality problem-solving services that are available on the remote AGNET system in Nebraska. The support software used for this project is a timesharing protocol interpreter which allows UNIX to login on other machines and use their services. This method has some severe restrictions, primarily in the area of achievable data rates, but promises immense improvements in the costs of using remote computer services, since communication between computers can proceed at the full available data rate and at late hours. The cost improvement is estimated to be around a factor of 100 in long-distance charges. The present work is an early step in the development of systems where personal computers on the farm, small local timesharing services, and large remote computer services will work together to provide services to rural residents that no one of the services could provide alone.

REFERENCES

1. Anderson, D. North Dakota AGNET Programmer. Personal Interview, October, 1981.
2. Anderson, R. H., and J. J. Gillogly. "Rand Intelligent Terminal Agent (RITA): Design Philosophy." R-1809-ARPA, The Rand Corporation, Santa Monica, California, February 1976.
3. Anderson, R. H., and J. J. Gillogly. "The Rand Intelligent Terminal Agent (RITA) as a Network Access Aid." *AFIPS, Proceedings of the National Computer Conference* (Vol. 45), 1976, pp. 501–509.
4. Anderson, R. H., M. Gallegos, J. J. Gillogly, R. Greenberg, and R. Villanueva, "RITA Reference Manual." R-1808-ARPA, The Rand Corporation, Santa Monica, California, September 1977.
5. Cain, Steven. "Computer Age Farming at Your Doorstep." *Successful Farming,* 77 (1979), p. 46T.
6. "UNIX Time-Sharing System." *The Bell System Technical Journal,* 57 (1978), pp. 1897–2312.
7. Gammill, R. C. "GPMX—A Portable General Purpose Macro Processor Adapted for Preprocessing FORTRAN." *AFIPS, Proceedings of the National Computer Conference,* (Vol. 45), 1976, pp. 927–933.
8. Gammill, R. C. "Position Paper: Personal Computers for Science in the 1980's." *Proceedings of the Oregon Report on Computing,* IEEE-sponsored conference, March 20–22, 1978 (also reprinted in ACM SIGPC Notes, 1 (1978), pp. 18–28, and as Rand Paper P-5954).
9. Gammill, R. C. "A Tiny Portable Language Independent Macroprocessor and Some Applications." *AFIPS, Proceedings of the National Computer Conference* (Vol. 50), 1981, pp. 415–420.
10. Gammill, R. C. "Research on Rural Communication and the Microcomputer Controlled Modem and Telephone." *Proceedings of Oct. 1981 ACM SIGSMALL Symposium on Small Systems, SIGSMALL Newsletter,* 7 (1981), pp. 40–46.
11. Henkes, R. "Computers Come to the Farm." *The Furrow,* 84 (1979), pp. 2–5.
12. Isaacson, P., R. C. Gammill. "Personal Computing." *IEEE Computer,* 11 (1978), pp. 86–97.
13. Thompson, K., and D. M. Ritchie. "The UNIX Time-Sharing System." *Communications of the Association for Computing Machinery,* 17 (1974), pp. 365–375.
14. Waterman, D. A. "Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition." R-2171-ARPA, The Rand Corporation, Santa Monica, California, February 1978.
15. "HAMS." *Successful Farming,* 77 (1979), p. H25.
16. "Computer Marketing Looking Good in Debut." *Successful Farming,* 79 (1981), p. H10.
17. "Cattlex." *Successful Farming,* 78 (1980), p. B12.

RIPS net: The impact of an optical communication network

by KOJI YADA
Electrotechnical Laboratory
Tsukuba, Japan

and

MASANORI HONDA and SEIJI FUJINO
Fujitsu Limited
Tokyo, Japan

ABSTRACT

The new RIPS (Research Information Processing System), installed in the Tsukuba Research Center of AIST (Agency of Industrial Science and Technology) at Tsukuba Science New City in Japan, began operation in January 1981.[1,2]

AIST thought of the RIPS plan for the purpose of more creative, more advanced research activities, taking advantage of the possibilities for centralization, since a number of national research organizations were gathered in Tsukuba.

RIPS is a distributed integral system, decentralized in regard to the individuality of the laboratories or the researchers, but centralized by its very large-scale computer system.

The RIPS network, using optical fiber, is composed of three kinds of networks taking into account cost performance and technical innovations of the future. One is the MultiPurpose Data Highway loop, handling a mix of coded data and voice data. Another is the star-pattern High Speed Dedicated Network to expand the distance between the computer channel and I/Os. The other is the star-pattern Video Distribution Network, through which users can observe the state of the two networks and the conditions of the RIPS center system. Overall length of optical fiber is about 360 km.

INTRODUCTION

Local area networks using optical fiber cable have been developed and put into practical use all over the world.[3,4,5]

The basic aim of the RIPS network is not only the installation of a very large-scale computer but also the establishment of a total research support system, with the new local network enabling each laboratory to make the best use of the full power of the computer.

When a laboratory or a researcher needs individual research information, a decentralized system catering to individuality is needed. Furthermore, active exchange of information between researchers in the network is needed. It is also desirable to have software and databases to be used in common by all users, to control the computer and the mass storage system centrally, and to use the system's resources efficiently.

Taking these conditions into account in RIPS, we constructed a distributed integral system having the merits of both a centralized system and a decentralized system. In the RIPS network, the host computer is installed at the RIPS center, computer devices are installed in laboratories, and an optical communication network links them.

We adopted the optical communication technology generally to take advantage of its mass high-speed data transmission capabilities and noise immunity.

Use of the optical network system contributes to system flexibility so that users at distant locations can use the central computer easily.

SYSTEM CONFIGURATION

Figure 1 shows the configuration of RIPS. The RIPS center is located at the center of the premises, where the very large-scale computer system FACOM M-200 is installed.

The RIPS center is designed to operate 24 hours a day, 7 days a week.

The centralized supervisory system makes possible operation without special operators.

To access the host computer from the remote work stations and user terminals in the laboratories, there are three optical communication networks:

- The multipurpose data highway
- The high-speed dedicated network
- The video distribution network

Table I shows the scale of RIPS in October 1981.

THE MULTIPURPOSE DATA HIGHWAY

The data highway accommodates varied types of data terminals, facsimiles, and minicomputers with standard data interface.

The concentration of channels to the host computer, switched connection between arbitrary ports, and voice communication are the typical features of the network.

The requirements of the local network are many, depending on the application. But the following are generally common:

- It must cover widely scattered terminals and computers.
- It must place a computer facility at the user's premises (e.g., on a desk).
- It must accommodate various types of terminals and make it easy to switch accessing to computers.
- It must be sufficiently flexible to permit expansion or changes in the network.
- It must be expansible to include new services, voice communication, and image communication.
- It must be easily managed and easily maintained.

To satisfy the above requirements, we developed a data highway with the following features:

1. *Highly reliable transparent network.* The highway is a duplicated loop transmission highway operated at 16.896 Mb/s.
2. *Voice communication.* In this data highway voice communication is possible by interphone. This voice communication provides a new communication service such as program guidance and operation schedule reporting.
3. *Multiloop network.* To achieve a reliable and flexible network, building block construction is employed for each loop. The network is expansible up to five loops, and interloop communication is possible through the loop exchange node (LX). In the network's configuration, remote centers are connected in a loop, and a star connection is used to connect all offices in the building. Therefore, an end user can have a desk data terminal.
4. *Microprocessor-controlled distributed packet transmission.* Packet transmission integrates 64 kb/s PCM voice and data of various speeds such as 48 kb/s, 4800 b/s etc., for interloop and intraloop communication. One packet consists of 16 bytes, and in each loop throughput is 120 kilopackets/s. Thus, a large number of data terminals can be accommodated. The main loop frame format is shown in Figure 2.

A switching connection between arbitrary terminals is achieved in this network. Then, prior to the data communication, signaling packets are transmitted between terminal nodes (TN) by supervisory node (SV) control to set the line. The line is released by the same procedure. The signaling packet throughput is 8000 packets/ packets/s. Fujitsu's MB8861, an all-purpose processor,

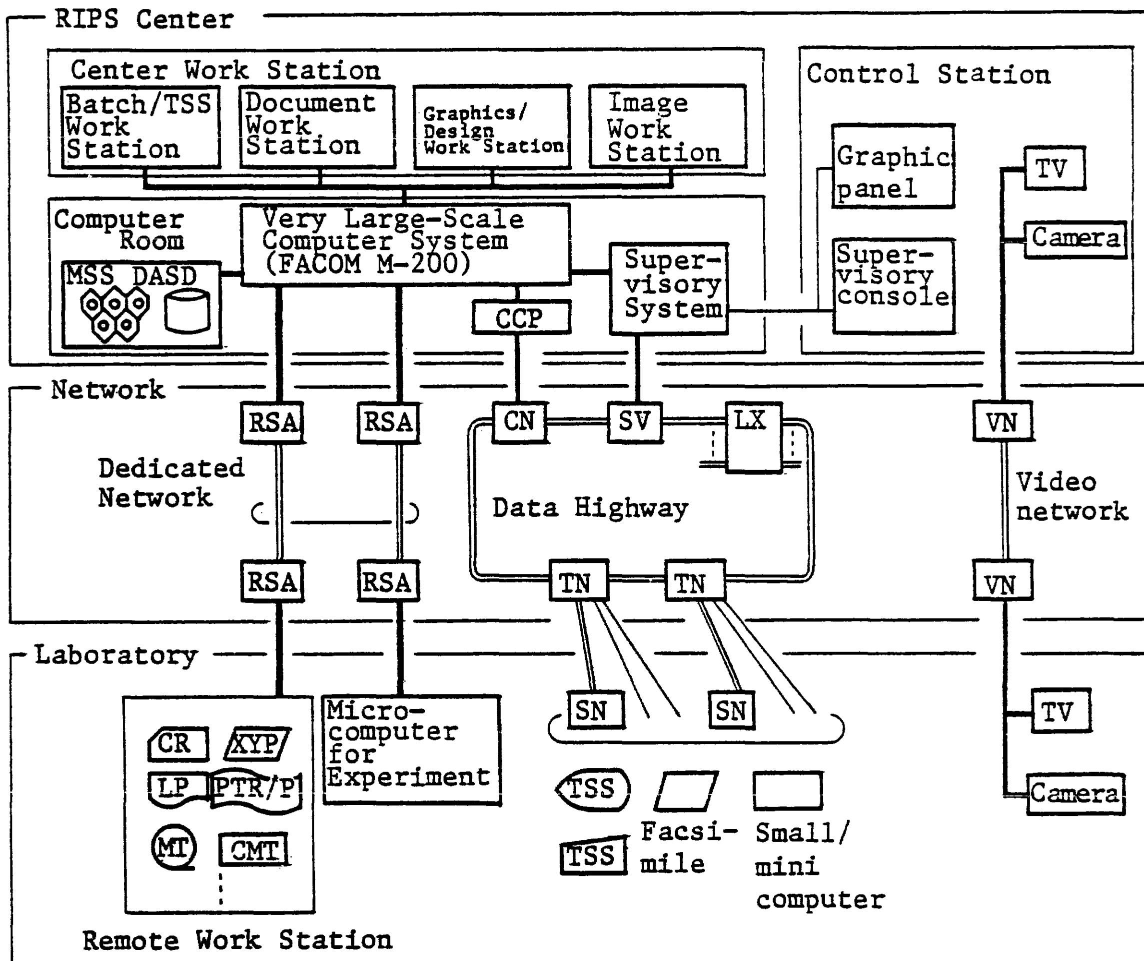


Figure 1—Configuration of RIPS

controls the network. Table II shows the specifications of the multipurpose data highway.

5. *Centralized network supervisory system.* In large computer networks, the network management facility is very important. The operation status of each loop and channel is transmitted to a network supervisory processor (NSP: PANAFACOM-U1500 minicomputer system). With NSP, it is possible to control the network operation mode and to realize remote maintenance. NSP handles large quantities of subscriber information such as data terminal attributes and can easily deal with channel attribute (e.g., transmission speed) alteration by software instructions from the center console.

The network is composed of several types of nodes, and optical fiber links connect these nodes, as shown in Figure 1. The nodes are as follows:

1. SV: The supervisor node is set in each loop. The SV controls the loop to localize malfunctions, to avoid overload, and to collect traffic statistics. SV operates the RAS function in three steps, as follows, when it detects an emergency status in the data highway: Normal/emergency line switching, auto loopback by loop reconfiguration, and separation of the abnormal terminal node.
2. CN: The center node is set up at the center. Ninety data channels are provided to interface the computer. Some of the data channels are connected in a time-sharing system to concentrate the line. Four CNs can be accommodated by each loop.
3. TN: Terminal nodes are installed in remote offices. There are 16 channels for interfacing the data terminals, minicomputers, and digital facsimiles directly or through subnodes (SN).
4. SN: Subnodes are set up at the users' offices. The subnode has a handset, a numeric key pad, a display, an audible-tone generator, and two low-speed (up to 9,600 b/s) data interfaces or one high-speed (48 kb/s) data interface. In the SN with two data interfaces, one is usually used for the conventional data terminal, and the other is for the image terminal. The SN is connected to the TN by optical fiber links, and the transmission speed is 768 kb/s. Figure 3 shows the SN equipment.

5. LX: The loop exchange node interfaces all loops to provide interloop communication; 1,200 data channels and 600 voice channels can be accepted simultaneously. The LX also has a master oscillator to which all loops are synchronized.

THE HIGH-SPEED DEDICATED NETWORK

The network connects the very large-scale computer (FACOM M-200) at the RIPS center with the laboratories in a star pattern of optical fiber transmission lines.

There are two types of Remote Station Adapters (RSA) used at transmission control units for the optical fiber transmission lines: RSA-C installed in the RIPS center and RAS-S installed in each laboratory.[6]

RSA-C is connected to the FACOM M-200 (host computer) channel. RSA-S is connected to various input/output devices or experimental minicomputers constituting a remote station. Interfaces between RSA-C and the channel and those between RSA-S and the input/output devices or minicomputers are equivalent to IBM360/370 I/O interfaces.[7] Major RSA specifications are shown in Table III.

TABLE I—Scale of RIPS

Item	Scale
Environment	
Laboratories and institutes	9
Researchers	2839
Number of researchers using RIPS	1613
Buildings	about 60
Total area	about 1.5km²
Host computer	
Main frame	FACOM M-200
Mass storage system	35GB
Disk volume	10GB
Optical fibers and devices	
LED	about 550
APD and Pin-PD	about 550
Optical fiber	about 360km
	Data Highway 197km
	Dedicated Network 136km
	Video Network 27km
Optical fiber connections	about 2800
	Optical Connectors 2000
	Splices 800
Data highway	
Loop	3
SV	3
CN	3
LX	1
TN	36
SN	220
Terminals	348
	TSS 319
	RJE 29
Lines for acoustic coupler terminals	9
Host-accommodated lines	218
Dedicated networks	19
Video networks	10

TABLE II—Multipurpose data highway specifications

Item	Specification
Loop composition	Maximum 5 loops/network
Number of nodes	TN—30 or less/loop CN—4 or less/loop SV—1/loop LX—1 or less/network
Number of connectable lines	Maximum 480 terminals/loop Maximum 240 subnodes/loop
Communication mode	Hot line connection Switched connection Broadcasting (voice only)
Connectable terminals	Low-speed lines: Asynchronous: Up to 1,200 b/s Synchronous: 1,200, 2,400, 4,800, 9,600 b/s High-speed line: Synchronous: 48K b/s
Call connection control	Pushbutton dial (built in subnode) Automatic call, interface (V24–200 series)
Interface	CCITT V24, V35, or equivalent Acoustic coupler FACOM low-speed I/O interface
Communication service	Calling party number display Representative number, abbreviated dialing Incompleted call detail display
Automation, labor-saving	Automatic power ON/OFF mechanism for nodes System automatic activate/deactivate function One-dimensional management of subscriber control table
Reliability	Full-duplex system for main transmission line System automatic/manual reconfiguration (switching to standby transmission line, bypass, loopback, etc.) Network multiloop configuration
Operation service	System configuration with NSP for the following functions: Collection of RAS information such as that for errors Collection of statistical information such as traffic volume Remote control for modification of line attributes, line loops, etc.

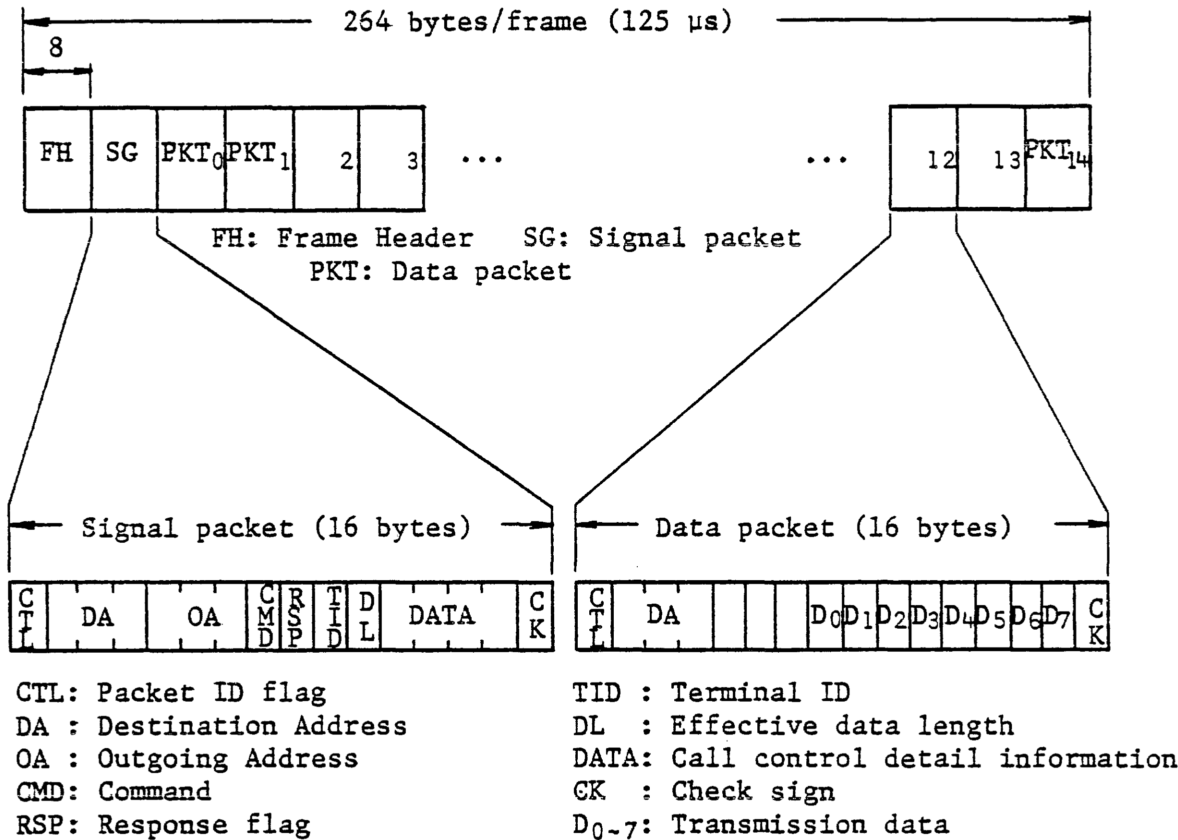


Figure 2—Frame format

The remote station adapter (RSA) converts I/O interface signals into serial signals and transmits them over a long distance. It also transfers data over a long distance at a high rate of speed.

To make possible the connection of an I/O device requiring high-speed transfer, such as magnetic tape unit, RSA makes high-speed transfer possible by blocking data with the use of a buffer memory, as shown in Figure 4.

Figure 3—Subnode equipment

Both RSA-C and RSA-S have three 64-byte blocks used as a cyclic buffer for high-speed data transfer.

Two-way serial interfaces are provided between the two RSAs, which continuously transmit frames. A frame consists of 36 bits (2 bits are used for synchronization, 1 bit is the frame parity bit, and the remaining 33 bits are data bits). Two data bits are used to specify one of the three data modes.

OPTICAL FIBERS AND DEVICES

The optical source is the highly reliable light-emitting diode (LED), operating at about an 830-nanometer wavelength.[8] It has a guaranteed lifetime of more than 1 million hours. The optical detectors avalanche photo diodes or PIN photo diodes, the choice depending on the range of automatic gain control.

TABLE III—RSA functions specifications

Interface between central computer and remote device	Equivalent to IBM360/370 I/O interface
Data transfer rate	Maximum 1.5M byte/s
Data transfer system	Block transfer/byte transfer
Transmission system	Frame synchronization

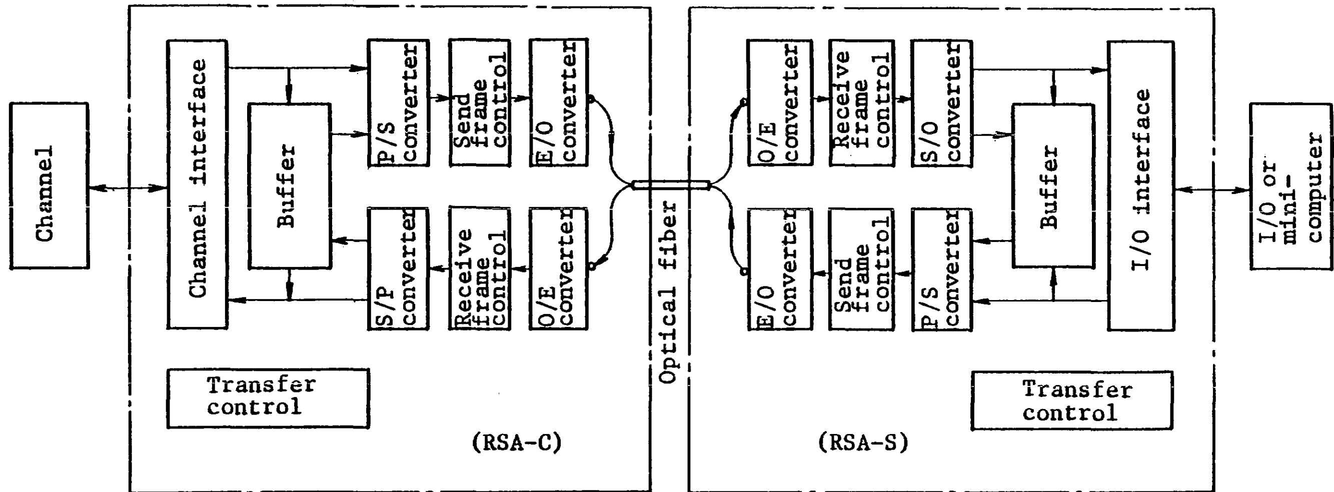


Figure 4—RSA block diagram

More than 600 optical source and detector pairs are used (See Table IV).

Step index fiber is used for indoor cabling, and graded index fiber is used outdoors. Table V shows the characteristics of the optical fiber.

RIPS OPERATION

Since RIPS began operation, an average of 11,590 batch jobs and 13,933 TSS sessions per month have been processed.

The data highway has 348 terminals; about 35 manufacturers supply the terminals. This network not only connects the host computer and the terminals; using the switching function, it also links the medium-scale computer with terminals connected to other nodes. Users can use TSS service and so on from the host computer and other computers.

In the dedicated network, 14 lines are used for remote batch stations at 9 laboratories. Five lines are used to connect minicomputers for experiments, and a measurement interface control unit (MICU) is connected to one of them.

TABLE IV—Design values of network optical transmission lines

Item	Multipurpose data network: Loop	Multipurpose data network: TN—SN	High-speed dedicated network	Video network
Light source	LED	LED	LED	LED
Photodetector	APD	PIN-PD	APD	APD
Transmission speed	16.9 Mb/s	0.768 Mb/s	33.3 Mb/s	6 MHz
Optical fiber cable	GI	SI/GI	GI	GI
Gain (dB)	39.3	29.0	36.3	24.0
Loss (dB)	24.0	16.1	24.0	20.7
Margin (dB)	15.3	12.9	12.3	3.3

THE IMPACT ON THE INFORMATION PROCESSING SYSTEM

Now RIPS is expanding the areas for researchers' information processing. For example, experimental data gained through MICU and the dedicated network can easily be displayed graphically by an extensible engineer language (AXEL).[9] By using MICU and AXEL, researchers can easily tabulate experimental data.

We are developing the electronic filing system (ELF).[10] Using facsimile, researchers can store image information in the host computer and retrieve it later. This system will help reduce the amount of paper in the office.

We have a translation system capable of translating simple English phrases into Japanese. Though it cannot translate all English sentences, it is satisfactory for translating titles of reports in the literature database.[11]

We are now studying and developing additional features, such as translation from Japanese to English and translation to and from other languages.

TABLE V—Design values of optical fiber cables

Item		Unit	Design value: Outdoor use	Design value: Indoor use	
Cable	Outside diameter	mm	17	10	
	Allowable tension	kg	200	50	
	Allowable bending radius	mm	100	60	
	Number of optical fiber cores	cores	2 or 4	2	
Optical fiber	Profile	—	GI	GI	SI
	Core diameter/clad diameter	μm	50/125	50/125	62.5/125
	Transmission loss*	dB/km	≤4	≤6	≤6
	Transmission bandwidth	MHz·km	≥200	≥200	≥30

*wavelength: 0.83 μm

We firmly believe that these RIPS-based applications will be a great help in researchers' activities.

CONCLUSIONS

This paper discusses the basic concept of the distributed integral research information processing system and the part of the system that has already been put into practical operation.

Optical fiber is most suitable for construction of a local network. The network will bring new applications to the office of the future.

In the next stage, we would like to employ new communication technologies, such as extension to a high-speed network integrated system that includes video and image information and optical atmospheric transmission.

In the future, efforts will be made to develop and commercialize research automation by upgrading design with CAD and by making the best use of trial production capability with CAM and office automation technology. Research results of the technical learning system and document-translating system will be tested with CAI, and wide-range system service using a communication satellite and DDX will be developed. Efforts will also be made toward further development of decentralized utilization and centralized high-level functions and high-level system integration.

REFERENCES

1. Yada, K., T. Ochiai, and M. Honda. "Optical Fiber Makes Research Information Processing System." *25th Computer Society International Conference,* San Francisco, 1981. Piscataway, New Jersey: IEEE Computer Society, 1981, pp. 450–453.
2. Yada, K., T. Tanaka, T. Hisano, and M. Honda. "Development of RIPS-Net Using Optical Communications." *Proceedings of the 23rd Annual Convention of the Information Processing Society of Japan,* Tokyo, October 10–14, 1981. Tokyo: IPSJ, 1981, pp. 605–606.
3. Yamaguchi, K., Y. Suzuki, and R. Yatsuboshi. "Microcomputer System Distribution by Optical Fiber Data Highway." *25th Computer Society International Conference,* San Francisco, 1981. Piscataway, New Jersey: IEEE Computer Society, 1981, pp. 454–457.
4. Toyoda, T., T. Inoue, S. Koganemaru, A. Izeki, M. Fujii, and K. Yanase. "Laboratory Automation with Optical Data Highway for Toyota Motor Co., Ltd., Higashi Fuji Technical Center." Fujitsu Limited, 1980 (vol. 31, no. 2).
5. Ceng, W. Y., S. Ray, R. Kolstad, J. Luhukay, R. Campbell, and J. W-S. Lin. "ILLINET—A 32 Mbits/sec. Local-Area Network." *AFIPS, Proceedings of the National Computer Conference* (Vol. 50), 1981, pp. 209–214.
6. Yada, K., H. Hasegawa, and M. Igawa. "Optical Dedicated Network in RIPS." In *Proceedings of Distributed Processing System of the Information Processing Society of Japan.* Tokyo: IPSJ, 1981, pp. 13–18.
7. IBM. "IBM-SYSTEM/360 and SYSTEM/370 I/O Interface Channel to Control Unit Original Equipment Manufacturer's Information." IBM Manual GA22-6974-4, File No. S360-S370-19, January 1978.
8. Abe, M., O. Hasegawa, Y. Komatsu, Y. Toyama, and T. Yamaoka. "Performance Optimization of GaA1As DH LED's." *Proceedings of the 36th Annual Device Research Conference,* Santa Barbara, California, 1978.
9. Koganemaru, S., M. Iijima, E. Hashimoto, and K. Hirai. "Interactive Data Handling System for Laboratory Automation." *Proceedings of the Computer Software and Applications Conference,* 1981. Piscataway, New Jersey: IEEE Computer Society, 1981.
10. Tanaka, T., K. Yada, T. Hisano, and T. Ochiai. "Test Use of Electronic Filing System." *Proceedings of the 23rd Annual Convention of the Information Processing Society of Japan,* Tokyo, October 10–14, 1981. Tokyo: IPSJ, 1981, pp. 1089–1090.
11. Nagao, M., J. Tsujii, K. Yada, and T. Kakimoto. "Document Information Retrieval System with English-Japanese Translation Capability." *Proceedings of the 23rd Annual Convention of the Information Processing Society of Japan,* Tokyo, October 10–14, 1981. Tokyo: IPSJ, 1981, pp. 829–830.

A coherent scheme to support location-independent references in internetwork environment

by RAY CHENG and J. W. S. LIU
University of Illinois
Urbana, Illinois

ABSTRACT

We describe in this paper a coherent scheme to support location-independent references of remote entities in an internetwork environment where entities may dynamically connect to computer systems in different networks. More specifically, we propose that name servers be provided within networks and in hosts to support user creation and use of symbolic internetwork names of entities. Message-forwarding mechanisms to support communication between entities that may migrate from network to network are designed.

INTRODUCTION

Since the early 1970s, user desire to access system resources provided in remote computer systems and to share information with other users motivated the development of data communication networks linking computer systems and terminal equipment. More recently these same considerations have generated a great deal of interest in the interconnection of individual networks to form joint networks of computer systems.[1–4] To access remote system resources and information in a network or a joint network, a user must know how to name remote entities. In most existing networks individual computer systems and networks have their own naming schemes. Many of the naming schemes are designed and implemented in an ad hoc way. Furthermore, the naming schemes are usually so ingrained in the individual systems that to modify the naming scheme of a computer system or a network in a joint network often leads to major upheavals. Therefore, design of a coherent naming scheme is one of the key issues to be solved to allow convenient remote accesses of resources and information in an internetwork environment.

The naming and addressing scheme in computer networks has not been treated in a coherent manner until recently.[5,6,7] In most current networks, addresses are used by system and application programs to refer to remote entities. With rare exceptions, the design of most networks is based on the assumption that communicating entities either do not move, or rarely move, from computer to computer. The use of addresses to refer to remote entities in networks is analogous to the use of physical core addresses in machine languages in the early stage of computer usage. In distributed systems where migration of files, data, server processes, and users between computers and networks are frequent, the tasks of generation and maintenance of directory information and the tasks of supporting location-independent references to all entities are usually carried out at the application layer.[15]

We describe in this paper a coherent scheme to support location-independent references of remote entities in an internetwork environment where entities may dynamically connect to computer systems in different networks. More specifically, we propose that name servers be provided within networks and in hosts to support user creation and use of symbolic internetwork names of entities. Message-forwarding mechanisms to support communication between entities that may migrate from network to network are designed.

Our assumptions and terminologies are discussed first. Naming and addressing schemes used in well-known networks are briefly described. In the next section the functions and structures of symbolic name servers that jointly support symbolic internetwork naming are discussed. Then a gateway level message-forwarding mechanism and two host-level message-forwarding mechanisms for internetwork entities that may move from network to network are described. An example illustrating how the schemes proposed here may be incorporated into transport layer and session layer protocols is described in the last section.

TERMINOLOGIES AND ASSUMPTIONS

In this paper we refer to a computer or a terminal that is attached to a network and that uses the basic services provided by the network as a *host*. It is irrelevant to our discussions to consider whether packets transmitted throughout the network are parts of messages exchanged because of interprocess communication or are parts of voice or facsimile transmissions. Similarly, there is no need for us to differentiate among the types of the sources and destinations of data packets. We refer to an element that is capable of generating, sending, or receiving information in the form of messages as an *entity*. We refer to the administrators who are responsible for the maintenance and operation of an entity as the *owner* of the entity. Entities reside in hosts.

We confine our attention to packet-switched networks. (Since there is no possibility of confusion, we simply refer to them as networks.) A *packet switch* is a collection of hardware or software modules that implements the basic network packet-switching functions. An entity gains access to its local packet-switched network via its local host-to-network interface. Messages to and from entities may be segmented into blocks either by local hosts or by the packet switches connected to local hosts. Such blocks of data transported by the packet-switched network are called *packets*. When networks are interconnected, we refer to the interconnected networks collectively as a *joint network*.

We view a *name* of an entity as a data item that designates the entity and distinguishes it from other entities. The *address* of an entity is viewed as a data item that indicates where the entity is located or can be reached.[6] A name is used either by human beings or by machines.[5,7] Names intended for the use of human beings usually have arbitrary lengths (within the limitations imposed by implementation). They are usually constructed with mnemonic character strings. We refer to a name that consists of a variable number of mnemonic characters as a *symbolic name*. Names consisting of bit strings for ease and efficiency of implementation are usually used by machines. We refer to names consisting of bit-strings as *bit-string names*.

We refer to a name used to identify an identity locally within its host as its *local name*. We refer to a name that is used networkwide to identify an entity within a network as its *C-name*,[8] and a name that is used in an internetwork environment to identify an entity as its *I-name*.

Naming and Addressing Scheme in Some Networks and Joint Networks

In most existing networks, addresses of entities rather than names are used. For example, in ARPANET,[9] before establishing a connection to a remote process, a user has to know the location of the remote process. In Cyclade,[10] an entity is supervised by a Transport Station (TS). If a TS moves from one host to another host within the same *region,* only the packet switches in this region have to be informed. Entities that refer to the moved entities are not affected in this case. However, if an entity moves from one TS to another TS, or a TS moves across regions, all entities and TSs that may want to communicate with the moved entities have to be informed of the move. In the Distributed Computing System (DCS),[11] location-independent reference using C-names is achieved with packet broadcasting. In Ethernet,[12] the station address of a remote entity has to be known before a user can use the services provided by the remote entity.

Addresses are also used for access of remote entities in interconnected networks. For example, while the CCITT X.25 recommendation allows each public data network to use its own addressing scheme when a virtual circuit is being set up, the X.121 recommendation that specifies the address format of an X.75 joint network interconnecting a group of X.25 public data networks has decreed the use of a numbering system similar to that of a public telephone network. With this scheme, an entity cannot move across a network. In a Pup joint-network,[13] a remote entity is referred to by a port address containing a hierarchical address consisting of a network number, a host number, and a socket number. In these networks all the programs or tables that refer to the moved entity have to be modified when an entity is moved.

Assumption and Objectives

To make our discussions of naming and addressing schemes clear, we summarize here our assumptions about the joint network. We consider here a joint network of an interconnected set of networks. The individual networks may support virtual circuit service or datagram service. Each network may have a different way to name and to address its local entities, may accept different packet sizes, and may have different flow control and error control protocols. Each entity in a network can be reached by a C-name within that network. We assume that the networks are interconnected via gateways. Since the level of interconnection is irrelevant to our discussion of naming and addressing schemes, we make no assumption about the manner in which networks are interconnected in our internetwork model.[2,3]

The internetwork naming and addressing schemes discussed here are designed to achieve the following goals:

1. All the entities in the internetwork environment can be referred to throughout the joint network. (We refer to these entities as internetwork entities.)
2. The internetwork naming and addressing schemes are independent of the naming and addressing schemes of individual networks.
3. The entities may move within a host, within a network, or across networks without any undue cost.
4. The naming scheme provides symbolic names and bit-string names.
5. The naming scheme supports the mechanisms of initializing and deleting I-names.
6. The internetwork naming and addressing schemes can be used both in datagram-oriented and in virtual-circuit-oriented internetwork architectures.

Relation between Symbolic Names and Bit-string Names

We note that a user can use either a symbolic name or a bit-string name to refer to a remote entity. However, symbolic names usually use more bits than bit-string names. If we use symbolic names in packet headers to identify the destinations of the packets, it costs more overhead in terms of numbers of bits in the packet headers and in terms of the amount of space required to store the routing directory. For the sake of efficiency in implementation, we assume here that when a remote entity known by its symbolic name is accessed, the symbolic name is first translated into the bit-string name or the address of the entity by its local host. The corresponding bit-string name or address is used in the packet header. If the naming scheme supports the use of location-independent bit-string names, the translation of symbolic names to bit-string names is not affected by the moving of entities. If the host does not translate symbolic names to bit-string names for its user, a user in this host is constrained to use a bit-string name to indicate the destination of a message.

SYMBOLIC NAME SERVERS

Since a symbolic name must be unique in an internetwork environment, a process that manages the registration of symbolic names is needed in a joint network. This process may also be responsible for the translation of symbolic names to bit-string names. We call such a process a *symbolic I-name server* (SNS). An SNS may reside either in a gateway or in a host. For the reasons of reliability as well as speed, a number of SNSs may be distributed throughout a joint network. In order to use fully the services provided throughout the joint network, information on what entities are available, what kind of services these entities provide, and how to reference these entities must be made available to internetwork users. Therefore we assume that when a symbolic I-name of an entity is registered to an SNS, a short description of the entity associated with the symbolic I-name should also be kept at the SNS. Moreover, the SNS includes a facility to allow users to query the database containing the names and descriptions of the entities.

In summary, an SNS provides to the internetwork users services needed for the translation of symbolic I-names to bit-string I-names, registrations of symbolic names, update of the information on the entities, and queries about joint-network entities. In addition, when a packet is received in a host, the SNS can be asked to translate a bit-string name into a symbolic name. We assume that each SNS has a database

(DB) that provides database services to the SNS. The internal structure of a DB is irrelevant to an SNS.

A host can provide a *local symbolic name server* (LSNS) to network interface routines and local users of the host. The LSNS can be viewed by its users as containing all information about symbolic I-names of the joint network. It provides such services as translation of symbolic I-names into bit-string I-names; registration of symbolic I-names; update of the information of joint-network entities; query services; and the creation, translation, and deletion of local representation of remote entities. The relation among LSNS, SNS, and DB; the algorithms of SNS and LSNS; and the database interface are described in the Appendix. In general, if a name used in layer i is a symbolic name and a name used in layer i-1 is a bit-string name, the services of a symbolic name server are provided in layer i.

MESSAGE-FORWARDING SCHEMES

When entities in a joint network are allowed to migrate from network to network, some mechanisms must be provided in order that messages intended for a moved entity may be forwarded to its new location. We describe here two message-forwarding schemes for this purpose. For the sake of concreteness, we assume that a message intended for an entity contain in its header the bit-string I-name of the entity.

When an entity is to be moved within a network or to a different network, the owner of the entity sends a move request to the supervisor gateway. The supervisor gateway then removes the C-name/I-name association from its memory. The entity, however, still has the same I-name.

After the entity has moved to another host and has obtained a new C-name, it sends a location report request to its new supervisor gateway. The new supervisor gateway keeps the C-name/I-name association in its memory and sends the location report with the I-name and supervisor gateway association to the registrar gateway of the entity. The registrar gateway then updates the association of I-name and supervisor gateway code in its memory.

When it is decided that an entity will no longer be accessible to other entities in a joint network, its bit-string I-name can be deleted by its owner by sending a termination request to the supervisor gateway. The supervisor gateway then deletes the I-name from its memory and sends the terminating request to the registrar gateway of the entity. The registrar gateway deletes the I-name from its memory. Depending on the pattern that the entity is referred to and on the implementation of the registrar gateway, the deleted I-name may be reused after a reasonably long period or never reused. One criterion for the reuse of a bit-string I-name is that there be a zero or a negligible possibility for internetwork entities to refer a name that designates another entity.

Gateway Mapping Scheme

We assume here that the bit-string I-name of an entity carries no information on the location of the entity. Among the gateways in a joint network, some are responsible for the functions of creation and maintenance of I-names and associations between I-names and C-names of internetwork entities. A gateway at which an internetwork entity registers to get a bit-string I-name is referred to as the *registrar gateway* (or name server gateway of the bit-string I-name) of that entity. A gateway that transmits internetwork packets for a joint-network entity is referred to as the *supervisor gateway* of that entity. In every network there are one or more supervisor gateways for the internetwork entities in the network. However, there might be no registrar gateway in some networks. A gateway can be both a supervisor gateway and a registrar gateway for some internetwork entities, but there are gateways that are neither a supervisor gateway nor a registrar gateway for any internetwork entity.

The bit-string I-name of an entity is created when it is decided that the entity will communicate with remote entities in a joint network. In this case the owner of the entity selects a gateway among the gateways in its local network as the internetwork supervisor gateway of the entity. All internetwork communication for that entity will be handled by its supervisor gateway. Conceptually, the supervisor gateway can be viewed by the entity as a host in the network, with all the remote internetwork entities as the local entities of that host. The owner of the entity sends a name registration request containing the C-name of the entity to the supervisor gateway. The supervisor gateway selects a registrar gateway and forwards the registration request to it. The registrar gateway finds an available I-name from its internetwork name space, keeps the association of I-name and supervisor gateway in its memory, and returns the I-name to the supervisor gateway. The supervisor gateway keeps the I-name/C-name association in its memory and returns the I-name to the owner of the entity. Without loss of generality, we assume that a bit-string I-name consists of two parts: the first part is the code for its registrar gateway, and the second part is a bit string that uniquely identifies the entity among all entities registered by the registrar gateway.

When an entity is to be moved within a network or to a different network, the owner of the entity sends a move request to the supervisor gateway. The supervisor gateway then removes the C-name/I-name association from its memory. The entity, however, still has the same I-name.

After the entity has moved to another host and has obtained a new C-name, it sends a location report request to its new supervisor gateway. The new supervisor gateway keeps the C-name/I-name association in its memory and sends the location report with the I-name and supervisor gateway association to the registrar gateway of the entity. The registrar gateway then updates the association of I-name and supervisor gateway code in its memory.

When it is decided that an entity will no longer be accessible to other entities in a joint network, its bit-string I-name can be deleted by its owner by sending a termination request to the supervisor gateway. The supervisor gateway then deletes the I-name from its memory and sends the terminating request to the registrar gateway of the entity. The registrar gateway deletes the I-name from its memory. Depending on the pattern that the entity is referred to and on the implementation of the registrar gateway, the deleted I-name may be reused after a reasonably long period or never reused. One criterion for the reuse of a bit-string I-name is that there be a zero or a negli-

gible possibility for internetwork entities to refer a name that designates another entity.

When an entity wants to communicate with a remote internetwork entity, it sends a data packet containing the bit-string I-name of the destination entity to its own supervisor gateway. By examining the first part of the bit-string I-name, the supervisor gateway can determine the registrar gateway of the destination entity and then transmit the packet to the registrar gateway. The registrar gateway of the destination entity finds the current supervisor gateway of the destination entity and transports the packet to the supervisor gateway. Having received the packet, the supervisor gateway of the destination entity transports the packet to the destination entity through its local network. This procedure is illustrated by the examples below.

Example 1. As shown in Figure 1, the internetwork entity with bit-string I-name 2x wants to send a packet to another internetwork entity with a bit-string I-name 4y. The numbers 2 and 4 indicate the registrar gateways of these entities. Gateway 1 is the supervisor gateway of entity 2x, and Gateway 3 is the supervisor gateway of entity 4y.

Entity 2x sends an internetwork packet with destination name 4y to the supervisor gateway 1 of the entity 2x. Gateway 1 forwards the packet to the registrar gateway 4 of destination entity 4y. Gateway 4 finds that entity 4y is currently connected to Gateway 3 and passes the internetwork packet to Gateway

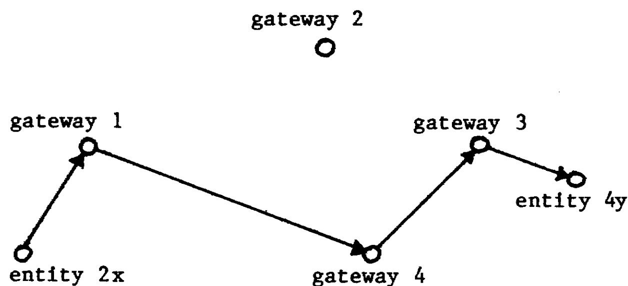


Figure 1—Entity $2x$ sends a packet to Entity $4y$

3. Gateway 3 then passes the internetwork packet to entity 4y through the local network.

Example 2. This example illustrates how the procedure described above can be simplified when the registrar gateway and the supervisor gateway of the entity are the same. As shown in Figure 2, Gateway 1 is both the supervisor gateway and the registrar gateway of the entity with a bit-string I-name 1x. Gateway 2 is both the supervisor gateway and the registrar gateway of entity 2y. Figure 2 shows the path of an internetwork packet sent from entity 1x to 2y.

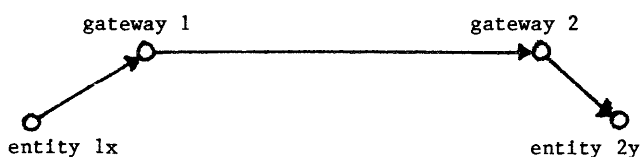


Figure 2—Entity $1x$ sends a packet to Entity $2y$

Packet Forward Scheme

We suppose here that bit-string I-names are managed by hosts in the network. In each host there is a process that assigns unused I-names to entities. We call this process an I-name server. To create an I-name, the owner of an entity requests an I-name from the I-name server in its host. Bit-string I-names may have a hierarchical structure consisting of a network number, a host number, and a socket number. Note that, in this case, the bit-string I-name of an entity also indicates the location of the entity when the name of the entity is created.

With this scheme, when an entity moves to a new host, its owner requests a new I-name and leaves this new I-name as a forward internetwork address in the old host. After an entity has moved, the packet containining the old I-name is sent to the old host. From the old host, the data packet is forwarded to its new host. Hence this entity can be referred to either by the old I-name or by the new I-name. (Figure 3)

If an entity moves several times, the easiest way to forward packets is to link the forward I-names serially, as shown in Figure 4. Because the time delay for packet transportations is significant, this method makes communicating with an entity that has moved many times very time-consuming. There are several ways to modify this method. The first modification is to forward internetwork packets through the original location,

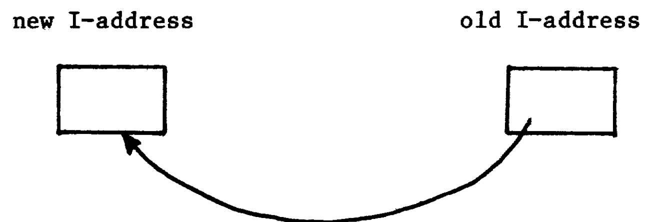


Figure 3—Packet forwarding

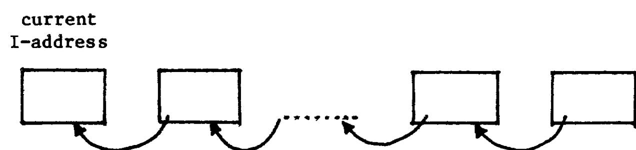


Figure 4—Cascade forwarding

where the original location of the entity is the host (or a protocol handler) in which the entity resides when the entity first acquires an I-name. As shown in Figure 5, all forwarding I-names are pointed to the original location. When a packet arrives at a host and it is found that the entity has moved, the packet is forwarded to the original location of the entity. When a packet is received at the original location and the destination entity has moved, the packet is forwarded to the current I-name location. The algorithms for this method are described in a PASCAL-like language, as follows:

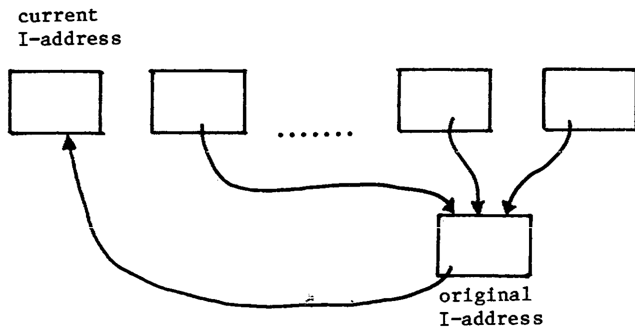


Figure 5—Forward through original host

```
forwardRecordType = record
                    current : boolean;
                    (* true if the entity is
                       in this host            *)
                    original boolean;
                    (* true if this host is the
                       original location of the entity  *)
                    reference : I-nameType;
                    end;
(* When an entity is created *)
new(I-name);
with I-name^ do begin
    current := true;
    original := true;
    reference := I-name;
end;
(* When an entity is moves from I-nameOld to I-nameNew *)
new(I-nameNew);
with I-nameNew do begin
    current := true;
    original := false;
    reference := I-nameOld^.reference;
    (* points to original I-name *)
end;
with I-nameOld^ do begin
    current := false;
    reference :=I-nameNew;
end;

(* Forwarding mechanism *)
if I-name^.current then
    passToEntity
else
    sendTo(I-name^.reference);
```

Using the method above, one packet transportation is needed to send a packet to the current I-name location of the entity. If a packet is sent to the original location and the entity no longer resides there, two packet transportations are needed. If a packet is sent to an old location that is not its original location, three packet transportations are needed. When an entity moves, only one message sent to the original location is needed.

The second variation is to forward internetwork packets with a feedback update. As shown in Figure 6, when an entity moves, it leaves a forward I-name in the previous location. When an internetwork packet arrives at a host and it is found

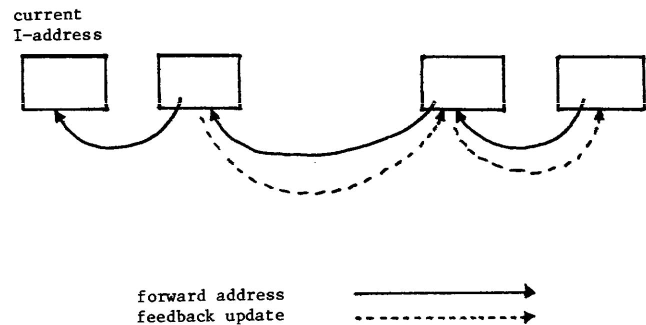


Figure 6—Forwarding with feedback updates

that the entity has moved to a new location, the packet is forwarded to the new location and an update message is sent back to the location that has forwarded this packet. Hence, besides the source and destination I-name field, there is one more address field in the internetwork packet to indicate the location that forwards this packet. The algorithms for this method are specified as follows:

```
forwardRecordType = record
                        current : boolean;
                        reference : I-nameType;
                    end;
(* When an entity is created *)
new(I-name)
with I-name^ do begin
    current :=true;
    reference := nil;
end;
(* When an entity is moved from I-nameOld to I-nameNew *)
new(I-nameNew)
with I-nameNew^ do begin
    current :=true;
    reference :=nil
end;

(* Forwarding mechanism *)
if I-name^.current then
    passToEntity
else begin
    sendTo(I-name^.reference);
    sendFeedbackTo(forwardSource, I-name^.reference);
end;
```

Using this method, if a packet is sent to the location where the entity currently resides, one packet transportation is needed. If a packet is sent to an old location where the entity does not currently reside, the number of packet transportations ranges from 2 to $n + 1$, depending on how up-to-date the forwarding addresses are in the previous locations, where n is the number of moves their entity has made.

AN EXAMPLE OF THE USE OF SYMBOLIC NAME SERVERS

We give here an example based on the draft reports on protocol feature specifications by the National Bureau of Standards, which uses the seven-layer architecture of the ISO Reference Model for Open Systems Interconnection.[3,14] In the reports of the National Bureau of Standards, the name supported by the session layer is a symbolic name, and the name supported by the transport layer is an address. In this example, we assume the name supported by the transport layer is a bit-string name. In this case, the symbolic name server is in the session layer.

In this example, among the *events* of *service primitives* provided in the session layer and the transport layer, we will use the "request" event of the TRANSACTION service primitive of the session layer and the "request" event of the TSEND service primitive of the transport layer. The TRANSACTION service primitive provides for single access data exchange for correspondent session users without establishing a session. The TRANSACTION(request) event notifies the session layer that it is to transfer the specified data as a transaction to the indicated session user. The TSEND service primitive of the transport layer provides for single-access data exchange for corresponding transport users without establishing a transport connection. The TSEND(request) event notifies the transport layer that it is to transfer the specified data as a transaction to the indicated transport user.

An entity with symbolic I-name "Nety.Second_Bank.Acct1" wants to send data to an entity with the symbolic I-name "Netx.Third_Bank.Supv15." Furthermore, the first entity has a local symbolic name "Acct1" under user "238." The message sent to the session layer, possibly having gone through the application layer and the presentation layer, is as follows:

```
TRANSACTION(request(
                    "Acct1",
                    "Netx.Third_Bank.Supv15",
                    mode,
                    data
                    ) )
```

The protocol handler of the session layer asks the LSNS to translate "Acct1" by calling 1sns.translate (see Appendix). The lsns.translate finds out that "Acct1" is a local symbolic name and returns the bit-string name of the entity, 1110001010110, to the protocol handler of the session layer. The protocol handler of the session then asks the LSNS to translate "Netx.Third_Bank.Supv15" into a bit-string name by calling lsns.translate. The lsns.translate finds out that "Netx.Third_Bank.Supv15" is not a local name, and then asks SNS to translate it by sending a message to sns.translate. The sns.translate queries its database by calling db.query to find the associated bit-string I-name, 1010011011001. SNS then returns the bit-string I-name, 1010011011001, in a message to LSNS. LSNS finally returns the bit-string I-name to the protocol handler of the session layer. The session layer passes the data transfer message to the transport layer by calling the transport layer primitive:

```
TSEND(request( 1110001010110,
               1010011011001,
               precedence,
               security level,
               compartment,
               data
               ) )
```

We ignore the procedures that pass the message down to the network layer, the data link layer, and the physical layer, as well as the procedures that pass the message up to the remote entity, since they are irrelevant to the use of symbolic names. In the destination host, when the message is passed to the session layer by the transport layer, the session layer may ask symbolic name servers to recover the symbolic name of the entity that sends this message.

ACKNOWLEDGMENTS

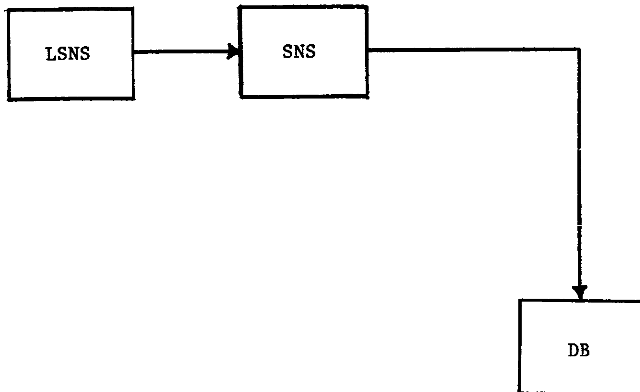
This work was partially supported by the U.S. Army, CORADOM, Fort Monmouth, N.J., Contract No. US ARMY DAAK 8080K0060.

REFERENCES

1. Cerf, V. G., and R. E. Kahn. "A Protocol for Packet Network Intercommunication." *IEEE Transactions in Communications,* COM-22, 1974, pp. 637–648.
2. Cerf, V. G., and P. T. Kirstein. "Issues in Packet-Network Interconnection." *Proceedings of the IEEE,* 1978, pp. 1368–1408.
3. National Bureau of Standards. *Features of Internetwork Protocol.* Draft Report, Report No. ICST/HLNP-80-8, 1980.
4. Postel, J. B. "Internetwork Protocol Approaches." *IEEE Transactions in Communications,* COM-28 (1980), pp. 604–611.
5. Saltzer, J. H. "Naming and Binding of Objects." *Operating Systems, An Advanced Course.* Berlin, New York: Springer-Verlag, 1978.
6. Shock, J. F. "Inter-Network Naming, Addressing, and Routing." *Proceedings of the 17th IEEE Computer Society International Conference (CompCon),* September 1978, pp. 72–79.
7. Watson, R. W. "Naming in Distributed Systems." *Advanced Course on Distributed Systems.* Berlin/New York: Springer-Verlag, 1980.
8. Pouzin, L., and H. Zimmermann. "A Tutorial on Protocols." *Proceedings of the IEEE,* 66 (1978), pp. 1346–1369.
9. Carr, C. S., S. D. Crocker, and V. G. Cerf. "HOST-HOST Communication Protocol in the ARPA Network." *AFIPS, Proceedings of the Spring JCC* (Vol. 36), 1970, pp. 589–597.
10. Pouzin, L. "CIGALE, the Packet-switching Machine of the CYCLADES Computer Network." *Proceedings of the IFIP Congress,* Stockholm, Sweden, August 1974, pp. 155–159.
11. Farber, D. J. "A Ring Network." *Datamation,* February 1975, pp. 44–46.
12. Metcalfe, R. M., and D. R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Network." *Communications of the ACM* 19 (1976), pp. 395–404.
13. Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe. "Pup: An Internetwork Architecture." *IEEE Transactions on Communications,* COM-28 (1980), pp. 612–623.
14. Zimmermann, H. "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection." *IEEE Transactions on Communications,* COM-28 (1980), pp. 425–432.
15. Lindsay, B. "Object Naming and Catalog Management for a Distributed DataBase Manager." IBM Research Report RJ2914 (36689), 1980.

APPENDIX

In this appendix we describe the relation among LSNS, SNS, and DB, the algorithms of SNS and LSNS, and the database interface.




```
module sns;

procedure translate(sname, bname, error_code);
(* procedure translate returns the associated bit-string I-name
of a symbolic I-name or returns the associated symbolic I-
name of a bit-string I-name. *)
  db.query(snamekey,bname,answer,next);
  if answer = nil then
    prepare_error_code(error_code);
end; (* translate *)

procedure register(entity_record,error_code);
(* procedure register inserts the information of a symbolic
*i-name into the database. *)
  db.new_record(entity_record,error_occurred);
  if error_occurred then
    prepare_error_code(error_code); (* name conflict *)
end; (* register *)

procedure update(key,update_record,error_code);
(* procedure update updates the information of a given sym-
bolic *i-name *)
  db.query(key,entity_record,query_id,more_answer,
  error_occurred);
  (* update the fields of entity_record that appears in
  update_record *)
  update_record(update_record,entity_record);
  db.update_record(key,entity_record, error_occurred);
  if error_occurred then
    prepare_error_code(error_code);
end; (* update *)

procedure query;
(* procedure query reads query commands from the sub-
scriber, queries the database, and writes the answers to the
subscriber until it reads a 'bye' command.*)
  repeat
    read(command)
    if command < > 'bye' then begin
      analyze_command(command,key,result_kind);
      db.query(key,answer,query_id,more_answers,
        error_occurred);
      if error_occurred then begin
        prepare_error_code(error_code);
        write(error_code);
        end
      else begin
        write(answer);
        while more_answers do begin
          db.get_answer(query_id,answer,
            more_answer);
          write(answer);
        end;
      end;
    end;
  until command = 'bye';
end; (* query *)

end; (*module sns *)

module db;

(* since the structure is irrelevant to name servers, we describe
here only the assumed database interface. *)

procedure query(key,answer,query_id,more_answer);
(* procedure query searches the database with the given key
and returns the result to 'answer'. if there are more than one
answer matches the key, it sets_more_answer to true, and
assigns a query_id to this query. the caller uses this query_id
to get next answer by calling procedure get_answer. *)

procedure get_answer(query_id,answer,more_answer);
(* procedure get_answer gets the answer from previous query
identified by query_id. if there are more answers,
more_answer is returned true. *)

procedure new_record(entity_record,error_occurred);
(* procedure new_record inserts the record into the data-
base. *)

procedure update_record(key,entity_record,error_occurred);
(* procedure update_record updates the record of the given
key. *)

procedure delete_record(key,entity_record,error_occurred);
(* procedure delete_record deletes the record of the given
key. *)

procedure reset_db:
(* procedure reset_db initializes the database. *)

end; (* module db *)

module lsns;

procedure translate(sname,bname,error_code);
(* procedure translate returns the associated bit-string I-name
of a symbolic I-name or returns the associated symbolic I-
name of a bit-string I-name. *)
  if not is_local(sname,bname) then
    sns.translate(sname,bname,error_code);
end; (* translate *)

procedure register;
(* procedure register lets the user to register a new symbolic
i-name. *)
```

```
  (* interactively ask user the information of the entity_record *)
  get_register_info(entity_record);
  sns.register(entity_record,error_code);
  if error_code <> nil then
    write(error_code);
end; (* register *)

procedure update;
(*procedure update updates the entity_record. *)
  (*interactively ask the user the key and the fields to be updated
    and prepares an update_record which contains only the
    field to be updated.*)
    get_update_info(key,update_record);
  sns.update(key,update_record,error_code);
  if error_code <> nil then
    write (error_code);
end; (* update *)

procedure query;
(* procedure query connects the query routine of a symbolic
i-name server to the user. *)
  sns.query;
end; (* query*)

procedure make_local(user_id,sname,bname,error_code);
(* procedure make_local makes an association between the
symbolic local name 'sname' and the given bit-string i-name
'bname' under the naming environment of the user
'user_id'. *);
  insert_pair(user_id,sname,error_code);
end; (* make_local *)

procedure delete_local(user_id, sname,error_code);
(* procedure delete_local deletes the association between the
local name 'sname' with the bit-string name from the naming
environment of the user 'user_id'. *)
  delete_pair(user_id,sname,error_code);
end; (* delete_local *)

end; (* module lsns *)
```

Issues and methods for practical distributed data processing applications—I

by MAURICE BLACKMAN
Arthur Andersen & Co.
Houston, Texas
and
HUGH RYAN
Arthur Andersen & Co.
Chicago, Illinois

ABSTRACT

This paper presents methods for one of two key activities in the creation of practical distributed data processing (DDP) systems for business computing. The first activity, discussed in this paper, is to select a configuration of hardware and software to support the implementation of a multiapplication plan. The second, discussed in a companion paper, is to select data distribution and manipulation approaches for one application within the limits set by the results of the first activity. To establish a justification for the methods, the paper selects a definition of DDP, discusses the alternatives to DDP for reaching the objectives of the enterprise, and identifies the design issues to be solved or avoided in a practical system for a commercial establishment.

INTRODUCTION

Distributed data processing (DDP) seems to have been selected as the computing industry's newest panacea for business computing. DDP has very attractive characteristics, accompanied by some significant technical challenges for the pioneer. This situation would be acceptable if the decision to use distributed processing were always made by computing professionals. Unfortunately, the spectacular improvements in the price/performance ratings of small computers has led to a significant number of articles in the business press[1–4] that promote the use of small computers in DDP configurations. As a consequence, the initiative in using DDP is taken by the chief executive, and the MIS director is frequently faced with a directive to "implement distributed processing."

Principal Tasks

In our experience and that of our colleagues, there are two activities crucial to a successful installation. The first is the preparation of a plan or strategy for the disposition of functions, data, and hardware and the identification of the necessary supporting software. The second is the disposition of data and the associated choice of approaches to the timing and synchronization of processing for each specific application within the constraints set by the choices of the prior plan.

The two activities fit into the context of a complete method for systems development.[5] They draw upon data developed in earlier tasks and provide input to tasks that follow. The two tasks are discussed because they result in key design decisions that control the impact of logical problems and technical gaps that we see in the present environment.

Each of the two papers discusses one of the two activities. The first is called strategy selection and the second data design. A method for collecting, organizing, and analyzing relevant data for each is presented. The methods, intended to be responsive to the current state of the art in DDP products, are aimed at minimizing present risk for the commercial user of computing. As logical challenges are overcome and more sophisticated products are introduced, the methods will evolve.

Issues

There are two groups of issues that can make DDP implementation a formidable task: logical puzzles that spring from the very nature of allocating processing between several processors[6] and gaps in the available technology[7] that make some approaches more costly than others. The second group of issues will lose their significance as the gaps are filled, but the first group may persist. In any event, the MIS director needs to adopt an approach to DDP that avoids both groups in order to succeed in creating a DDP environment within constraints of cost and time.

Definition

There are many interpretations of the term *distributed data processing.*[6,8,9] To place this contribution in context, we define DDP as follows: A data processing technique that provides access to computing power for end users by means of multiple processors interacting through the planned exchange of data over communication lines.

The definition is explicit, because DDP systems have many sources of variation:

- The processing capability of the devices at the remote location
- The extent to which data are distributed to terminal locations
- The discipline used to communicate
- The number of terminal locations
- The frequency of the processing at terminal and central locations
- The frequency with which the processes are synchronized
- The compatibility of the devices used for implementation

Our definition varies from others in certain key respects. It is more restrictive than some in implying storage capability and hence data residence at each node. It is less restrictive in not being limited to online or real-time interaction between processors and in not implying any dynamic allocation of tasks.

We wish to stress the definition question for two reasons. First, a clear definition helps the DP professional explain how a directive to "use DDP" has been interpreted. Second, the definition can be used as a template or profile for comparison with a proposed configuration. A conforming configuration may suffer from some of the logical problems or gaps in technology that exist. The implementer of a conforming configuration will be alert to these issues and take steps to avoid them.

STRATEGY SELECTION

User Objectives

Our earlier definition hinted at one of the key objectives of the organization examining DDP: to provide access to com-

puting power for the end user. *Distributed* means that some function of the data processing department is reassigned to a person closer to the mainstream of the business.

There are many means by which that function distribution may be achieved, including online terminals and decentralized computers. The many variations of DDP fill the middle ground. It is the quality of direct access, independent of the data processing department, that is the principal objective of the requester or commissioner of DDP. We therefore take the objective of strategy selection to be the configuration of a combination of hardware and software components that delivers access while optimizing an objective function addressing performance, development cost, operating cost, delivery date, and ease of use.

General Approach

The strategy selection activity can be decomposed into a sequence of tasks, as follows:

- Identification of applications
- Identification of distributed functions
- Analysis and selection of a configuration of hardware and data communications facilities
- Selection of standards for data distribution
- Development of a catalogue of software components
- Selection of products

Strategy selection occurs during a planning phase. The results must stand for five, ten, or even more years. The selected strategy must be responsive to requirements that are imperfectly defined and likely to change. The selection approach must therefore rely heavily on the experience, knowledge, and judgment of the selectors. The steps here presented are precise in their identification and sequence but will be subjective in their execution.

Application Identification

If a strategy is to be durable, the selectors must have a good appreciation of the applications to be implemented. From the point of view of the user, computing services are more convenient if they are accessible through a single work station. From the point of view of the enterprise, product acquisition is more economical if negotiated for the long term. For such reasons, this first task identifies as many potential uses for the configuration as possible.

Useful tools for this purpose are models of general processing and data structure requirements specific to the industry to which the enterprise belongs. These charts identify the components of a total information system to serve all the principal business functions for transaction processing, reporting, and operational and strategic planning. The components are connected to show how information flows between them. The charts are personalized to an industry to show its unique requirements and to help communicate with the executives who must determine which systems are likely candidates for automation. The charts form a useful checklist to help ensure that all requirements are identified. Characteristics of the enterprise discovered by interview and research, such as size, growth, and markets sought, help to develop a plan with a sequence and time frame that reflects the enterprise's priorities and resources. (See Figure 1.)

Function Distribution

Candidate functions capable of distribution at this time are the generic functions of a data processing department rather than the specific functions of single applications. This task sets limits on the maximum distribution in order to identify needed software support in the configuration and to establish overall quantities and capacity of equipment.

Table I gives examples of generic functions in three groups: operations, development, and management.

TABLE I—Generic data processing functions

Operations	Development	Management
Inquiry	Identification	Network design
Data entry	Justification	Capacity plan
Validation	Data design	Equipment acquisition
Correction	Screen/report	Software selection
File analysis	Process design	Standards
Printing	Programming	Network operation
Posting	Testing	Performance report
	Conversion	Data administration

Decisions about the dispersion of functions are responsive to characteristics of the enterprise and are made according to simple guidelines.

Operations

In general, operating functions are prime candidates and are considered first. If operating functions are not dispersed, the dispersion of any other functions is very unlikely. Operations functions are dispersed when the local business unit has autonomy in its own day-to-day operations, when its operations are on different business cycles from those of its parent or headquarters, and when its operations efficiency or effectiveness can be improved by rapid access to the results of computing. The only common exception would be a lack of the necessary skills or working conditions to operate computer peripherals.

Development

Development functions are also good candidates, for the reasons commonly cited for any form of end user participation in development: knowledge of the business, responsiveness of the design to user needs, and enhanced acceptance of the system. In addition, if the sites considered for function dispersion are widely separated, centralized development becomes difficult as a result of communications difficulties. Dispersed development is almost a necessity if the management style of the enterprise favors local autonomy as shown by a

INFORMATION REQUIREMENTS PLANNING CHART
PROPERTY AND CASUALTY

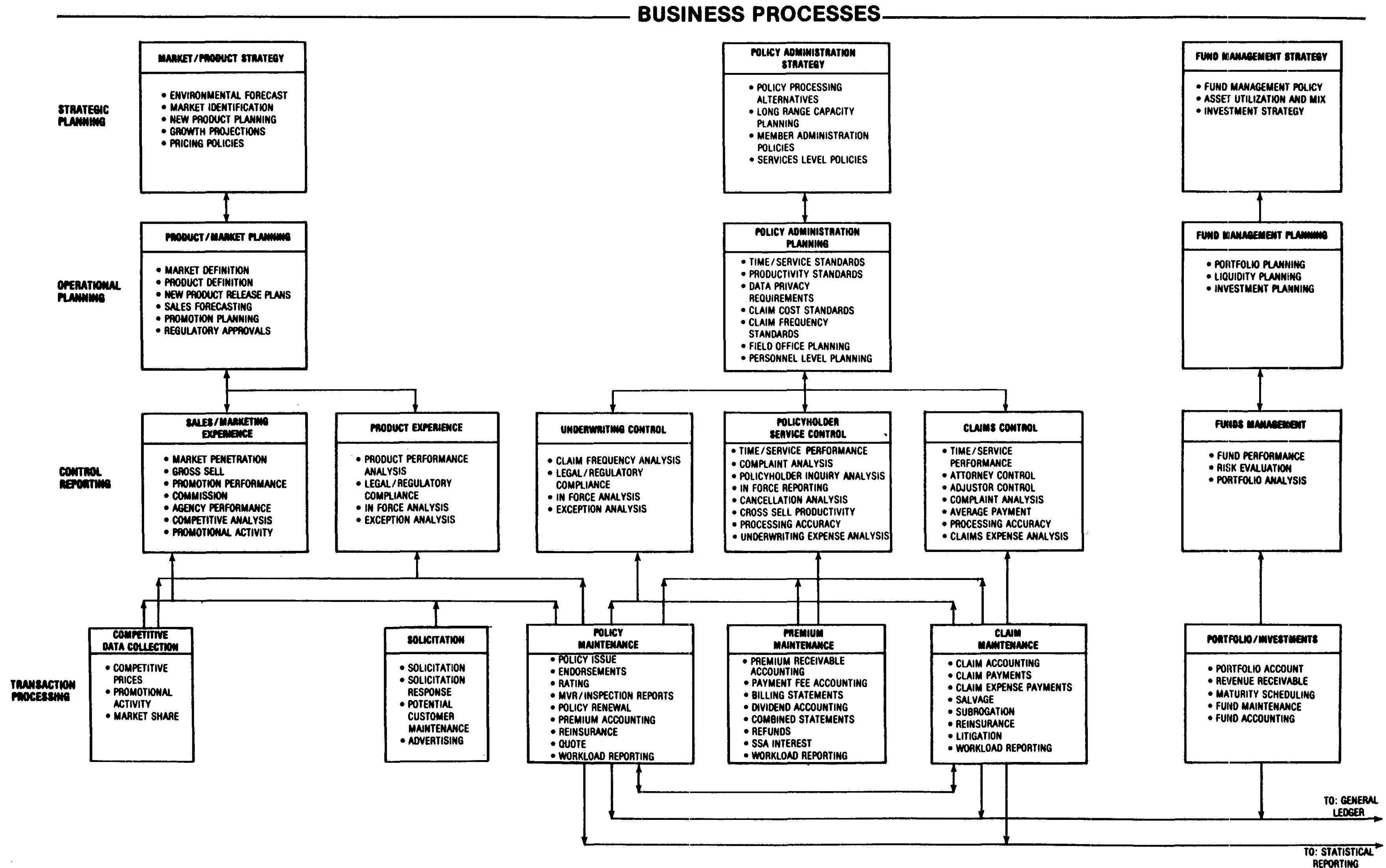


Figure 1—Industry system schematic—detail

profit center policy or if the products, markets, and businesses of the units are different. As with operations, the principal impediment to dispersion will be a lack of available skills. It should be noted that the dispersion choices being made here do not imply that all applications will be equally dispersed. The objective is to determine the degree of dispersion under the most favorable conditions so as to uncover software and equipment needs.

Management

Management functions are least likely to be dispersed, because the resources being managed are those shared by all users. Their proper selection and administration requires that the needs of all groups be blended and that facilities capable of working together be obtained. Conditions for the dispersion of management functions are the dispersion and local control of most of the assets of the enterprise and the dispersion of other service functions such as accounting, personnel administration, and research and development. Such an enterprise is really several distinct enterprises needing only occasional contact.

Hardware and Data Communications

Function distribution, although subjective and judgemental in its methods, is performed early, because its outcome may limit the hardware options considered. If only the simplest operating functions are distributed, then an online centralized solution is indicated. On the other hand, if extensive distribution of functions occurs, many strategies across the dispersed-access spectrum are possible.

This task may be subdivided into the following smaller tasks:

- Selecting access styles and modes
- Identifying access locations
- Configuring peripherals at each location
- Determining the processing/communications strategy
- Configuring data storage

Access modes

Access styles and modes include choices of online or batch access techniques and such special requirements as OCR, MICR, microfiche, or badge readers. The distributed work stations' complement of devices can be deduced by examining the likely input and output media of each application.

Locations

The access locations to be given direct service must be reviewed for instances of very low volume not justifying equipment. Lack of volume may be a temporary condition. Plans for network expansion at a later time may be an important input to the more precise steps discussed later in this section.

Peripheral configuration

The configuration of peripherals at each site is the first precise step. The different types of peripherals need to be treated separately. Methods for establishing keyboard screen and printing requirements to satisfy random and scheduled demand are well established from online system experience.[5]

Requirements for specialized peripherals must be considered carefully when unit capacity and cost are both high. Those forms of peripheral are more usually centralized for all but the largest sites.

Processing/communications strategy

We now come to the crux of the hardware question: should processing be centralized or distributed? The analysis consists of computing one-time and operating costs for alternative solutions and comparing them. The variety of alternatives—simple online terminals with a value-added network, smart terminals with a multidrop leased line network, intelligent terminals (stand-alone or clustered) with dialed WATS lines, minicomputers and mainframes with auto dial/auto answer—is usually too many. A technique of sampling, say three alternatives, followed by successive refinement, is preferred. The cost analysis must recognize that increased dispersed intelligence reduces data transmission volume and allows the use of low-tariff periods for transmission if the remote and central processing cycles can be decoupled.

The analyst must also consider geographically intermediate dispersion. There may be a case for sharing processors between several remote sites. This is an example of refining a strategy once a degree of distribution has been determined to be advantageous.

Data storage

The final step is to estimate data storage requirements *if* a strategy of physical distribution has been selected. This step is performed before more precise allocation of data within applications, since that cannot be done with certainty until designing each in turn. At this time we are concerned only with gross needs. Data storage capacity is needed for development as well as production, for programs as well as data, for system software as well as applications, for backup as well as mainline data, and for inefficiencies of use. In short, rather large estimates are indicated in preference to rather small.

Data Distribution Strategy

The final stage is to consider whether to place any limitations on the application designer so far as the complexity of data arrangements is concerned. Our principal target is to decide what forms of data synchronization are allowed and what forms, if any, of data directory are required.

Our first task is to determine whether master data centralization is indicated. Under current technology constraints, data needing to be up to date for all locations must be central-

ized. Other indicators include any suggesting the need for full-function database management system software. Such software needs the capacity and skills of a central site.

As a rule of thumb for strategic planning purposes, the designer should avoid data distribution configurations that require online posting within one commitment unit at more than one node. The designer should be wary even of online posting at a node other than the one at which a transaction originates. Paper II explains the background for this guideline.[10]

Software Configuration

Once the decisions about function, hardware, and data distribution are made, the generic components of software must be identified.

The key components are those associated with the fact of distribution:

- Data communication, including the appropriate online or offline protocols
- Message-routing logic for bringing data into the center and putting it out again, including distribution list interpretation logic, logical address to physical line mapping, alternate routing, etc.
- Data location logic in the case of split data files, including directory maintenance
- Data conversion logic to map data characters, fields, records, files, and databases from one format to another
- Development aids in each node type expected to support local development
- Program and dictionary distribution logic if one site is developing on behalf of others
- Remote job request submission and acceptance

and many others.

This list shows how the function, data, and hardware distribution decisions have consequences in the complexity of the environment.

To complete this task with any reliability, the designer must understand the software components necessary to the functions of operating, application development, and resource monitoring. Each of these divides and subdivides until a portfolio of components is developed that may be used as a checklist.

Vendor Selection

At the end of the strategy development, the designer has sufficient information to consider the products necessary for creating the target environment. This step will have been in view throughout the development of the strategy, since limitations in products available are the reasons for some of the rules of thumb suggested.

The principal components to be obtained will be the processing systems of the different nodes and the communications subsystem between them. By far the most productive rule at this time is to obtain processors at all nodes from the same vendor so that much of the potential software complexity can be subcontracted to the vendor. Many vendors have defined and implemented network architectures of richness sufficient to support cooperative working among multiple processors. The decision to use different vendors will inevitably involve the enterprise in the definition of its own architecture and the software to implement it.

SUMMARY

The objective of this paper was to describe a sequence of steps for developing a DDP strategy. The steps have been described only to show their purpose. The material is offered to help identify the components of the strategy and the dependencies between them. The resulting strategy is a high-level plan adequate for costing, for vendor selection, and for commencing the development of any necessary software.

In Paper II, we consider the detail of data design for an application within the constraints of a strategy.

REFERENCES

1. Brancatelli, J. "Office of the Future." *Texas Flyer,* 10 (1981), pp. 57–72.
2. Van Rensselaer, C. "Centralise? Decentralise? Distribute?" *Datamation,* 25 (1979), pp. 88–97.
3. Withington, F. G. "Coping with Computer Proliferation." *Harvard Business Review* (May–June 1980), pp. 152–164.
4. Buchanan, J. R., and Linowes, R. G. "Understanding Distributed Data Processing." *Harvard Business Review* (July–August 1980), pp. 143–153.
5. "Method/1." Arthur Anderson & Co., Chicago, 1980.
6. Kohler, W. H. "A Survey of Techniques for Synchronization and Recover in Decentralized Computer Systems." *ACM Computing Surveys,* 13 (1981), pp. 149–183.
7. Tozer, E. "Is Distributed Data Base Yet Mature Enough To Consider?" *22nd GUIDE Conference,* May 1981.
8. Loren, H. "Distributed Processing: An Assessment." *IBM Systems Journal,* 18, pp. 582–603.
9. Scherr, A. L. "Distributed Data Processing." *IBM Systems Journal,* 17, pp. 324–343.
10. Blackman, M., and Ryan, H. "Issues and Methods for Practical Distributed Data Processing Applications—II." *AFIPS, Proceedings of the National Computer Conference* (Vol. 51), 1982.

Issues and methods for practical distributed data processing applications—II

by MAURICE BLACKMAN
Arthur Andersen & Co.
Houston, Texas
and
HUGH RYAN
Arthur Andersen & Co.
Chicago, Illinois

ABSTRACT

This paper presents methods for the second of two key activities in the creation of practical distributed data processing (DDP) systems for business computing. The first activity, discussed in a companion paper, is to select a configuration of hardware and software to support the implementation of a multiapplication plan. The second, discussed in this paper, is to select data distribution and manipulation approaches for one application within the limits set by the results of the first activity. The paper assumes the definition of DDP established in the prior paper. It identifies some of the issues that constrain a commercial establishment with limited research funds and that justify the limitation of practical alternatives assumed as a basis for the methods.

INTRODUCTION

This paper continues the discussion of methods for designing practical DDP systems in view of the proven technology available to the commercial user. This discussion commenced in Paper I of this two-paper series. In the first paper a definition of DDP systems was selected to include all systems that would be recognized as DDP by a commercial user. A method for developing a DDP strategy was described. The definition is assumed for this paper, and the results of the strategy selection are assumed to be implemented. This paper addresses the second key task: data design.

Challenges and Gaps

The basis for the method proposed in this paper is a desire to select practical approaches to applications. We are seeking to avoid substantial effort in two areas:

- Overcoming the logical challenges of subdivided but dependent processing through additional application development
- Filling technical gaps with substantial system software development

The challenges and gaps include, for example:

- The correct handling of transactions found to be invalid at one node after having been validated, posted, and acted on at another node earlier in a cycle
- The maintenance of data integrity in answers to inquiries needing reference to several nodes while the late nodes in the sequence are continuing to post other transactions, unaware of the impending inquiry
- The preservation of the integrity of locally developed files related to local copies of centrally maintained files
- The preservation of the integrity of a distributed application that posts transactions on line at more than one node, so as to be able to back out a whole transaction upon the failure of any node, its database, or the communications network between them

The list goes on. These issues are substantial. In some cases they are only imperfectly understood. In time some will be routinely handled by software, especially when high-capacity data communications reduces the time for internode communication. For the present we believe that the average MIS director is better advised to avoid them.

The Complexity of Design for Distributed Data Processing

The system development process for a centralized environment includes the following segments:

- User requirements and functions are identified.
- A data design is defined on the basis of the requirements and functions.
- A technical architecture is designed by consideration of data design and business functions.
- A systems design is detailed from the technical architecture, using a combination of data and function-driven structured design.
- The entire design is implemented and converted.

This process is not trivial for a centralized design. The distributed environment adds an entire dimension to the design problem (Figure 1): the geographically distributed nodes of the network.

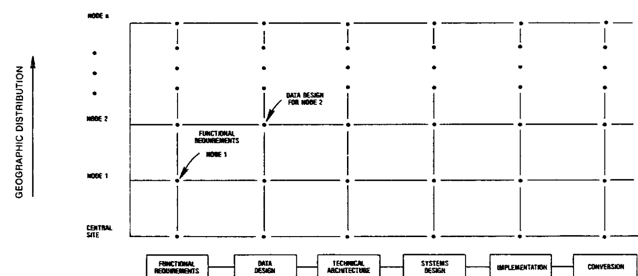


Figure 1—Design process in a distributed development

The design process must be applied to each class of node in the network. It is not sufficient to repeat the design process for each distinct type of node, because a design decision about one type may affect a decision about another. For example, a decision to allow changes of hourly wages at a central node may affect the timing, and even feasibility, of computing and printing paychecks at remote nodes.

Design in a DDP environment presents a set of complex design problems that require a well-structured approach to make the required decisions in a logical sequence. The methods presented address techniques for defining data allocation and operational data movement for a DDP system. However, they also affect functional analysis and the person/machine boundary.

These tasks take place in a context of other activities in the design process, as shown in Figure 2.

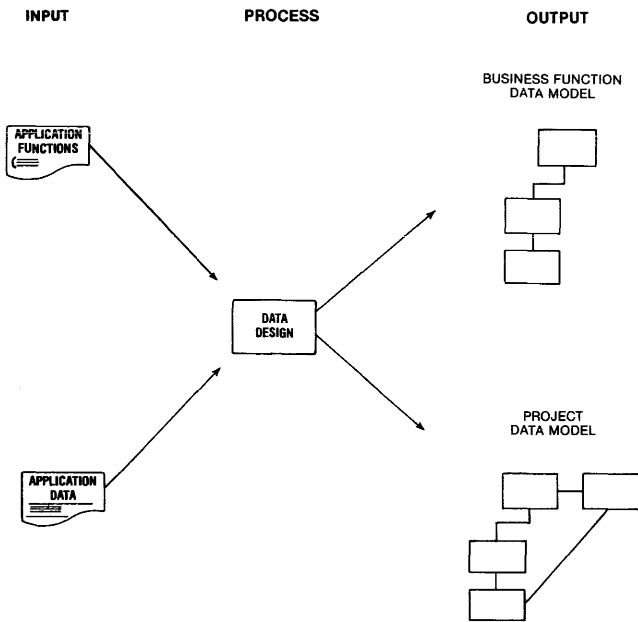


Figure 2—Inputs and outputs of data design

DATA ALLOCATION

The Process of Data Design

The designer begins the process of data design by identifying an initial data model. Then, for each business function to be supported by the machine, the designer must do the following:

1. Identify the aggregates and relationships required to support the function as a business function data model (BFDM).
2. Identify fields required to support the business function.
3. Assign fields to appropriate data aggregates in the BFDM.
4. Merge the BFDM into the prior project data model. This may require identifying new aggregates and/or relationships in the project data model.
5. Merge the fields into the appropriate aggregates of the project data model.

These steps of data design apply in both a centralized and a distributed environment.

Data design requires one additional step for a distributed environment:

6. On completion of the project data model (exhausting all business functions), minimize communication across nodes within each business function data model. (The basis for this additional step will be discussed in the section "Data Movement.")

The following is an example of the use of the method of data design for a distributed processing environment. Although it has been simplified by reducing the number of functions considered, it illustrates the key points of the approach. The example is based on parts distribution, in which computers at widely separated warehouses are used to keep track of local inventory. Stock status is reported centrally each week to support purchasing and allocation to warehouses. Predicted delivery data is then returned to the warehouses.

Example of Data Design for Parts Distribution Control

Consider two of the system's business functions: relieving inventory and purchasing new items (Figures 3 and 4).

RELIEVE INVENTORY STOCK — FILE; INDEX USE 213; ITEM FCT. 1.0

Function		I/O X-REF.	MODE	WHERE
1.0 Relieve Inventory Stock	1.1 Find Appropriate Stock Description	SCR100	On-Line	Distributed
	1.2 Review Inventory Status for Available On Hand	SCR200	On-Line	Distributed
	1.3 If Stock Available Relieve Stock	SCR200	On-Line	Distributed

Figure 3—Function chart for "Relieve Inventory Stock"

ORDER NEW STOCKS — FILE; INDEX USE 213; ITEM FCT. 2.0

Function		I/O X-REF.	MODE	WHERE
2.0 Order New Stock	2.1 Review Warehouse On Hand Stock			
	2.2 If On Hand and Committed Order Level Add to Purchase Amt. Note Allocation			
	2.3 On End of Warehouse Issue Purchase Order	Form100	Batch	Central

Figure 4—Function chart for "Order New Stock"

The function descriptions are developed in a prior task, "Identify Functional Requirements and Information Needs." The I/O decision is set in a task, "Define Process Function," but it is preliminary and subject to change.

As an initial guess for a project data model, we assume a single aggregate "warehouse stock status."

Consider the function 1.1, "Find Appropriate Stock Description," for the first step of data design. The appropriate screen layout shows that the function requires data describing the stock. The fields can be assigned to a single aggregate "stock description," which is the only component of this BFDM.

Finally, merging the BFDM and the current project data model gives an updated project data model of two aggregates, "warehouse stock status" and "stock description."

Next consider function 1.2, "Review Inventory Stock Status for Available On Hand." The BFDM (Figure 5) suggests that once the appropriate stock description is found, the warehouse stock status record is obtained. Fields are assigned to the warehouse stock status, including:

- Stock item identity
- Warehouse
- Short description
- Amount available

The existing project data model can satisfy this function, although the entities are now required by the local site.

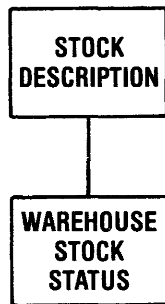


Figure 5—Business Function Data Model 1

Next consider the purchase function 2.0, "Order New Stock" (Figure 4). Examination of the related screen layouts and business function shows that a warehouse stock status entity and a purchase order entity are required. The BFDM suggested by this function is shown in Figure 6. The warehouse stock status gives the amount on hand and records the amount allocated to the warehouse. The purchase order entity records the purchase. In addition, it is related to the warehouse stock status. The relationship tells what purchase orders exist for a given item.

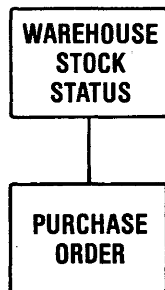


Figure 6—Business Function Data Model 2

Merging this business function data model (Figure 6) with current project data model gives a new project data model (Figure 7).

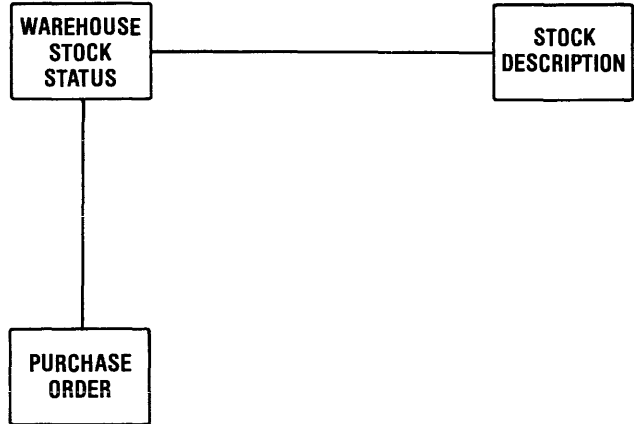


Figure 7—Project data model as of purchasing function

Assigning Data

If one reviews the business function data models, it can be seen that the warehouse stock status is needed both at a local node, for relieving inventory, and at the central site, for purchasing. If only one version of the warehouse stock status entity is defined, one of the BFDMs will need to cross nodes.

Few approaches are available for minimizing the number of cross-node communications. For the business functions that require crossing nodes, the options are as follows:

- Replicate the data aggregate at each node.
- Copy a portion of the data aggregate at a node.
- Partition the data aggregate across nodes.

In replicating the data aggregate, one stores a copy of the data aggregate at all sites where it is required. In our example, we could replicate the warehouse stock status at the local and central sites.

Replicated data are appropriate in cases where any of these conditions applies:

- Most of the data of the aggregate are used.
- Planned data use is periodic.
- Noncurrent data have small impact.

Copying a portion of the data aggregate is a variation of the replication option. The identity of the data aggregate may be lost when a portion is redundantly stored. For example, we can store the purchase quantity allocated to a warehouse in the warehouse stock status record. Thus, data from the purchase order entity are redundantly stored in the inventory stock status entity. The identity of the purchase order entity, however, is lost at the remote site. Copied data is appropriate in cases where only a small portion of a data aggregate is used.

The third option is partitioning data, i.e., storing the data for a node only at the node. For example, the warehouse stock status could be partitioned by warehouse. Partitioned data are appropriate when the data can be clearly identified with a

given node type. The partitioning option often seems desirable, although it can increase the complexity of the design effort significantly.

The above suggests the project data model shown in Figure 8. Note that the geographic dimension has been introduced into the project data model.

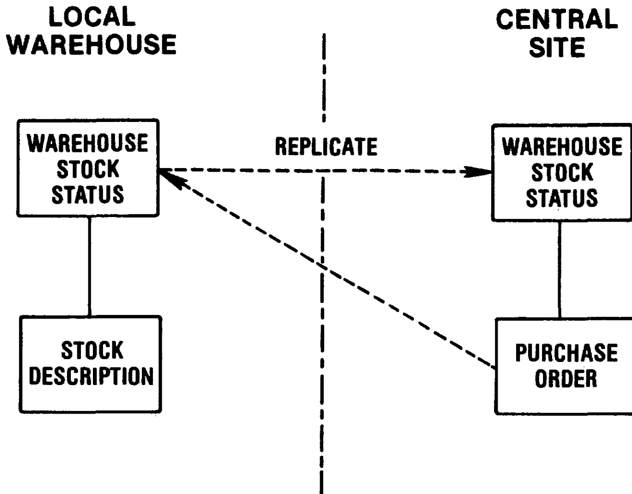


Figure 8—Project data model for distributed inventory processing

Structuring the Decision Process

We are using a relatively simple data design problem for illustration purposes. Full-size problems need a more formal documentation tool. The form shown in Figure 9 has proved effective in the mapping process.

To use the form, the data aggregates of the project data model are listed on the form. Then all BFDMs that use the data aggregate are listed and cross-referenced to the appropriate source. Next, the sites where the business functions are performed are identified. When there is a mix of nodes in the "Where" column, a mapping approach must be defined. As noted, the options are as follows:

- Replicated data
- Redundant data
- Partitioned data

In addition, one can choose not to avoid crossing nodes and use messages instead. Messages are online transmissions of data between nodes. They are appropriate when use of the data is unpredictable and the data must be current.

Concerns in Mapping the Project Data Model to Nodes

A casual reading of the above would suggest that the DDP environment is accommodated by only filling out a few forms.

HWR
7/26/81

DAT545

DATA AGGREGATE	BUSINESS FUNCTION DATA MODEL				MAPPING			
	X-REF.	DESCRIPTION	WHERE	MODE		NOTE		
Warehouse Stock Status		Relieve Inventory	Local	On-Line	Partition	1		
		Create Purchase Order	Central	Batch	Replicate			
Purchase Order		Create Purchase Order	Central	Batch				
		Check Available Stock	Local	On-Line	Redundant	2		
Stock Description		Relieve Inventory	Local	On-Line				
		Stock Description Change	Central	On-Line	Redundant			

Figure 9—Inventory control technical architecture

This is not the case. As has been discussed, the DDP environment is quite primitive today compared to centralized online/DBMS environments. The system software, when available, is not very competent. Performance, when moving data between multiple machines, is often inadequate. Restart/recovery typically consists of having the user repeat work or try another approach, such as the telephone. The mapping suggested by Figure 9 must be performed in full awareness of node-machine capabilities, transmission software capabilities, business function volumes, and business function requirements (versus desires).

Data allocation is a key step in the definition of a DDP design. The data design approach needs a single extension for mapping the data design to the distributed network. The objective recommended is to minimize cross-node communications for BFDMs. The next step is to define the approach to data movement for a DDP environment.

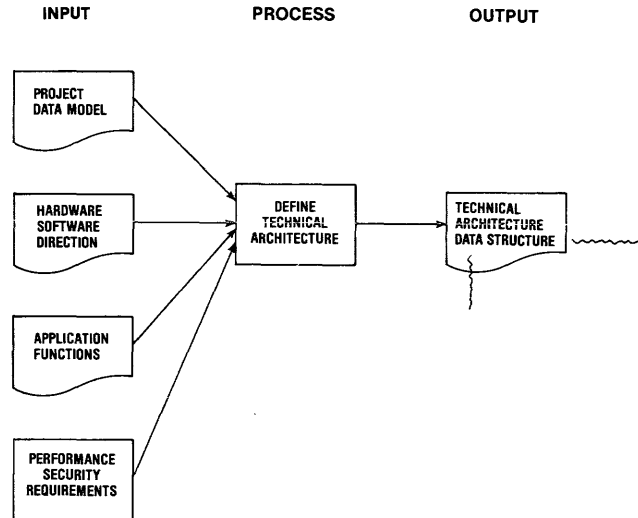


Figure 10—Inputs and outputs for defining data movement

DATA SYNCHRONIZATION

Selecting Data Movement Approaches

Data movement is defined for this paper as follows: a description of machine processes that relate data structures. The description specifies what the processes are, when the processes are run, and where the processes are run.

Inputs and outputs from the selection of data movement

The key inputs and outputs for selecting data movement shown in Figure 10, are as follows:

- The project data model
- Definitions of the application functions
- A description of performance and security requirements
- A description of the hardware/software environment from the strategy selection, that identifies hardware at each site, software (compilers, online monitors, database management systems) at each site, transmission hardware between sites, and transmission software to move data between use.

A thorough understanding of the capabilities and limitations of the hardware and software is necessary. Because of the rapidly changing hardware/software environment and the high cost of software development and maintenance, it is worthwhile to stay inside the constraints of available products.

The process of selecting data movement approaches

Relating data structures in a distributed environment means keeping the data synchronized (i.e., ensuring that at certain times the data at different sites have consistent and reasonable values). Thus, the process of selecting data movement for a DDP system is equivalent to defining the synchronization approach to be used for the system.

This task can be time-consuming, involving many decisions and reconsiderations. At this stage in a design, functions or features may need to be changed, reduced in capability, or completely foregone. Success is the result of creativity and compromise. Since the task is iterative, it is worthwhile to structure the design process.

The technical architecture form previously illustrated (Figure 9) is a means for structuring the process. As shown in Figure 11, the form can be extended with two additional columns. One column describes the synchronization requirement; the second describes the synchronization approach.

There are relatively few approaches available for synchronization:

- Transmit a message.
- Transmit a transaction file.
- Transmit a master file.
- Use the telephone.

Synchronization using messages

Synchronization using messages refers to a method of immediate transmission of data to other sites.

An example of the software and hardware for this approach is shown in Figure 12. The example is based on a DDP network with IBM 8100s and a 370. In this environment, the 8100 online monitor DTMS provides an interface to CICS or IMS on a 370. A message can be sent from DTMS to CICS. The message is processed in CICS and a reply sent to the 8100.

The transmission of messages presents some significant problems in today's environment. One problem is record protection. When a record is acquired for update at a node, some currently implemented mechanisms provide no lockout protection for accesses from other nodes. Some implemented protection mechanisms fail to handle a program failure at one of the participating nodes. Finally, implemented schemes may require so many exchanges to acknowledge approval that performance is not acceptable.

HWR
7/26/81

DAT545

DATA AGGREGATE	BUSINESS FUNCTION DATA MODEL				MAPPING		SYNCHRONIZATION REQUIREMENT	SYNCHRONIZATION APPROACH
	X-REF.	DESCRIPTION	WHERE	MODE		NOTE		
Warehouse Stock Status		Relieve Inventory	Local	On-Line	Partition	1	Data should be current as of when data is removed from inventory.	Synchronous on-line updating of transaction.
		Create Purchase Order	Central	Batch	Replicate		Data should be current as of end of business on the day when purchasing runs.	Transfer entire file as of end of day.
Purchase Order		Create Purchase Order	Central	Batch			Data should be current as of end of day.	Do updates in batch.
		Check Available Stock	Local	On-Line	Redundant	2	Data should be synchronized with last purchase order.	When purchasing runs, create a trans file of P.O. to send to local site for updating.
Stock Description		Relieve Inventory	Local	On-Line			Access local stock.	Desc. as a part of on-line processing.
		Stock Description Change	Central	On-Line	Redundant		Data needs to be current as of beginning of business at all local sites.	Transmit trans file of changes. Local site will post change transactions on overnight batch run.

Figure 11—Technical architecture extended

Performance, even without regard to protection, is a concern when using messages. As the example illustrates, a message involves the overhead of entry into two online monitors. In addition, there is the time for transferring data over lines between the sites. For large volumes of data, the line time can be significant. Thus the total time for a response from a message between processors may quickly become unacceptable to the user.

A third problem with messages is sensitivity to a node failure. In Figure 12, if the central site fails, the remote 8100 programs using the central site cannot run. One of the benefits of DDP, reliability through independence of sites, is lost when sites are connected through the use of messages. The problems posed by messages in DDP are significant. Messages should be used primarily for exception conditions. If, as the design evolves, it becomes obvious that many transactions require messages, use of a centralized system should be reconsidered.

Synchronization using a transaction file

Synchronization using a transaction file involves sending a queued set of transactions to other sites for asynchronous processing. Figure 13 gives an example of such a configuration:

- Personnel data are maintained at the remote site.
- Changes are posted to the local site database (1).

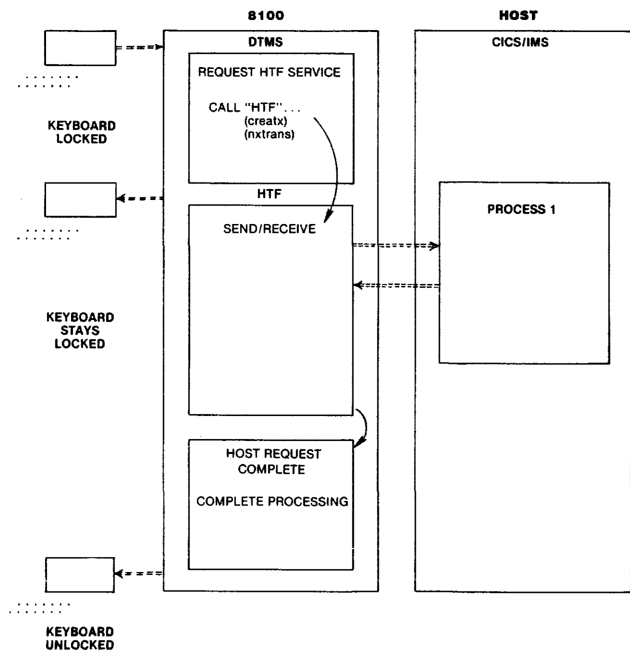


Figure 12—Synchronous message transmission between 8100 and 370 host

- A transaction file is maintained of personnel data changes made at the local site (2). An example of such change is the change of contact in event of an emergency.
- Periodically, the changes are sent to a central site for updating (3) and retransmission to all other sites (4).

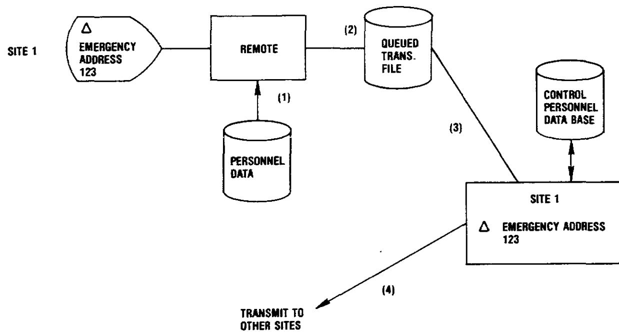


Figure 13—Example of transaction file synchronization

Transaction file—asynchronous processing

In a DDP environment, this technique for synchronizing data structures is common. However, it presents some significant problems.

First, since data are posted immediately at the local site, they are not synchronized with the rest of the network. This method may be acceptable or even desirable, but it does mean that while the transaction is queued, different sites will possess discrepancies in data. The user needs to understand this and recognize it as a planned part of the design and not a system error.

Second, is a potential for conflicting transactions. In Figure 14, two sites are entering an emergency contact address for the same individual. It is accepted at each site. However, at the central site it is not clear which transaction is correct. This is a common problem with the use of transaction files, and many approaches exist to deal with it, such as designating the site with primary responsibility or timestamping. Every conflict should be reported, whatever the technique.

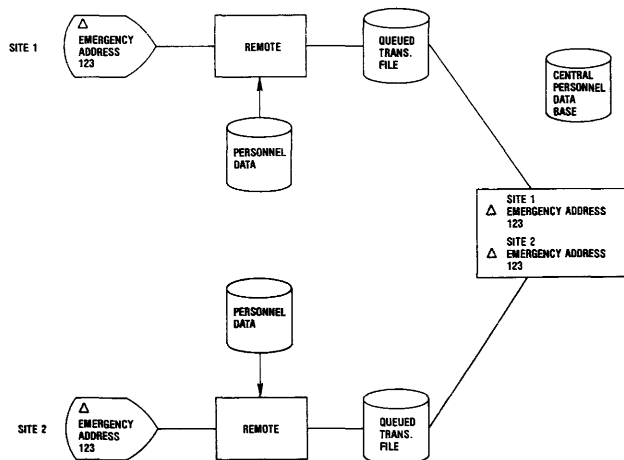


Figure 14—Transaction file asynchronous update problems

Another problem then arises: who is to receive the error report when two sites are involved. Once again, many approaches are possible:

- Report to the site causing the problem.
- Report to the site receiving the problem.

No general solution exists. In any approach, the impact of asynchronous processing of the transaction needs to be evaluated. In each case, responsibility for the resolution of problems should be identified and some means of follow-up defined.

Transaction file—transmission software

A second major problem with transaction files involves the development of transmission software to move the data files around the network. Such software must be able to do the following:

- Preschedule file transmissions.
- Send and receive sequential files between sites.
- Detect and report on error transmissions.
- Provide some form of a queuing mechanism to hold files until a site is prepared to accept them.

IBM's DSX software has some of these features and can be viewed as a representative example. This software has been undergoing design enhancements for over three years, which indicates that such transmission software may be a major effort by itself.

Synchronization using master files

Synchronization using master files resembles the use of transaction files, except that master file records are sent rather than the transactions causing the changes.

This technique does have some drawbacks. First, if the master file is large, performance may not be acceptable. The second problem involves the discontinuity of the file transmission. If files are not synchronized between two sites, the sudden revision of the file at a site may be disconcerting to a user. For one company, the central inventory files were used to maintain inventory balances. These balances were assumed to be correct and were used each month to update remote inventory balances. When the update occurred, an error report was produced of items with differing amounts, and an inventory check was made to resolve discrepancies. The discrepancies created a loss of confidence in the system, and there was user resistance to having "their" data overwritten by central-site data.

Another problem is the need to halt the application. Typically, while master files are being loaded, there are oper-

ational and application problems in running the application. Specifically, if changes are made while a file is being loaded, it may be difficult to predict which sites have what master file. To avoid this problem, the entire application must be stopped while loading occurs. Since users are reluctant to lose use of a system, even for short periods, and the software and operations procedures used to shut down the system may not be adequate to prevent concurrent online entrees, an effort should be planned to explain to users the need to halt the system on occasion. A subsequent effort during implementation is then needed to ensure that no one tries to enter "just one more."

CONCLUSION ON DEFINING DATA MOVEMENT

A means for documenting the relations of data structures is the technical architecture form (Figure 11) identifying synchronization requirements and synchronization approaches. The limited number of approaches include the following:

- Transmit a message.
- Transmit a transaction file.
- Transmit a master file.
- Use the telephone.

All these techniques present significant problems and require evaluation based on application specifics. Figure 15 summarizes the tasks that have been discussed in the course of this paper to develop DDP design.

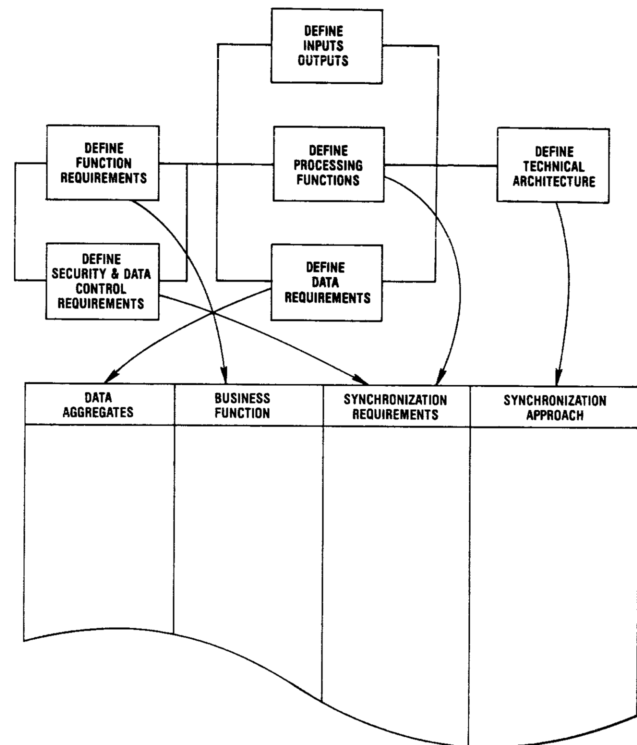


Figure 15—Summary of tasks to develop DDP design

PIONEER DAY

A technological review of the FORTRAN I compiler

by F. E. ALLEN
IBM Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT

The FORTRAN I compiler functions and organizations are described and shown to form the basis for many of the techniques used in modern compilers.

INTRODUCTION

Early in 1954, the FORTRAN I project was formed by John Backus. A fundamental question posed by the project was "... can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?"[1] A major goal was to provide an automatic programming system which "... would produce programs almost as efficient as hand coded ones and do so on virtually every job."[1] This seemingly impossible goal was met to an astonishing degree. In some cases, it produced code which was so good that users thought it was wrong, since it bore no obvious relationship to the source. It set a standard for object program efficiency that has rarely been equalled. The FORTRAN I compiler, completed in 1957, established modern compiler tasks, structure, and techniques.

The compiler was developed for the 704, an IBM machine introduced in 1954 featuring built-in floating point and indexing capabilities. It compiled the FORTRAN I language, which was defined as part of the project and evolved considerably as the project progressed. In order to achieve its efficiency goals, the high-level arithmetic statements in the source program had to be translated to minimize storage references, and, even more important, subscripts and their control had to make maximal use of the machine's three index registers. The way in which this was achieved is described by Backus, and other project members;[2] formalized by Sheridan;[3] and reviewed by Backus and Heising[4] and Backus.[1] The latter paper, presented at the 1978 Conference on the History of Programming Languages, contains a penetrating analysis of the project, its origins and development. The purpose of this paper is to assess the technological impact of the FORTRAN I compiler on compiler construction, theory, and practice as it has evolved over the last 25 years.

COMPILER FUNCTIONS AND ORGANIZATION

The basic function of the FORTRAN I compiler was, of course, to translate the source program to an object program for loading and executing on the target machine. However, confronted with a belief that compilers could only turn out code intolerably less efficient than hand coding and confronted with a machine that would make small lapses in arrangement of coding show up as sizeable inefficiencies, the primary goal of the compiler, and indeed of the whole project, was to produce very efficient code.

The greatest potential source for inefficiencies was believed to be the address calculations rather than the code generated for the arithmetic expressions. Thus, while the translator part of the compiler was designed to produce excellent code for the arithmetic expressions, the design and organization of the entire compiler was driven by the need to produce nearly perfect code for array addressing on the three-register 704. Consider the FORTRAN program fragment in Figure 1.

```
     DIMENSION A (10,10)
     DIMENSION B (10,10)
     .....
     DO 1 J = 1,10
     DO 1 I = 1,10
   1 A(I,J) = B(I,J)
```

Figure 1—FORTRAN program to move array B to array A

Remembering that FORTRAN stores arrays column-wise, the expansion of the subscript on array B (as well as on A) is $(I - 1) + (J - 1)*10$. Clearly such a computation inside the DO loops was intolerable—and certainly not two such computations, one for A and one for B. It is interesting to note however that some current, nonoptimizing compilers do perform variants of this computation and are tolerated quite happily.

To achieve a modicum of efficiency, there was also a need to utilize the 704 index register instructions to increment, test and branch to control the execution of the DO loops. Furthermore, the registers had to perform dual functions when possible, controlling the looping and indexing the arrays in the loops. Assembly language programmers did this all the time, and if compiled code was to compete, the FORTRAN translator had to also. The primary criterion which dictated the design of the compiler was, therefore, the need to produce excellent addressing code. In fact it is still the case today that the biggest payoff for optimizing compilers for languages at the FORTRAN level (e.g., PL/I and Pascal) is in generating good addressing code.

The compiler was divided into six sections (phases in today's terminology):

1. A statement identifier and arithmetic statement translator
2. A subscript and DO statement analyzer
3. A transformer which interfaced sections 2 and 4
4. A control flow analyzer
5. A global register allocator
6. Final assembly

As John Backus makes clear,[1] this organization evolved as the problems associated with assigning index registers became clear. The initial intent was to have translation and code generation, including register allocation, complete by the end of Section 2 so that all that was left was final assembly; i.e.,

producing the binary code, load maps, etc. Section 1, the translator, was to classify statements, compile object instructions for the arithmetic formulas, and partially compile or record information about the remaining statements (the I/O, DO, GO TO, IF, DIMENSION, and function definition statements). Section 2 was to compile the instructions associated with subscripting and DOs. When it became clear that the task of Section 2 was too complex, Sections 4 and 5 were created and then Section 3 to glue everything together.

It is worthwhile looking in more detail at what went on here because it presents a model of an approach to solving very complex problems—the use of a divide and conquer strategy. Stated in today's terminology and from today's perspective (after 25 years of problem partition and solution), the problems being solved were the following:

1. (Section 2) Assuming an unlimited number of index registers, to create optimal code for addressing and loop control. In today's terms this meant: (a) reassociating the subscript expansions to collect constants and make them part of the base address and to group subexpressions to minimize the computation required in the loop; (b) finding common subexpressions; (c) moving computations out of loops; (d) performing strength reduction so that subscript calculations become index register increments and decrements; (e) folding constants; and (f) replacing loop tests by tests on registers required for addressing within the loop, i.e., linear function test replacement.
2. (Section 4) To perform the control flow analysis and identify (probabilistically) the relative frequency of program regions.
3. (Section 5) To assign real registers to the symbolic registers in order to minimize, using the control flow based frequency information, storage references and register–to–register moves.

How does the overall organization of the FORTRAN I optimizer (Sections 2 through 5) differ from today's optimizing compilers? Today we would probably do control flow analysis first and use it as a basis for performing Section 2's optimizations. Separating register assignment from the problem of optimizing code involving symbolic registers is now considered a good strategy,[5] though many optimizing compilers have not exposed loads, symbolic registers, and all of the addressing code to their optimizing sections and have ended up with most of the problems originally faced by the FORTRAN project when doing register allocation!

How does the overall organization of the rest of the FORTRAN I compiler compare with today's compilers? The translation phase is typically broken into several subparts today; syntactic analysis, semantic analysis, and code generation are common partitions, although the evolution here is by no means complete.

Overall, the organization of the compiler was surprisingly simple. Most of the complexities arose from the desire to produce object programs competitive with hand code and the consequent need to gather information and postpone producing code until the analysis necessary to produce efficient code had been performed.

We now turn to a closer examination of the significant sections of the compiler (Sections 1, 2, 4 and 5) in order to assess their technological impact in more detail.

TRANSLATION

Today's compilers often use elegant, language-independent translator systems. The theory behind these systems did not really start to develop until the 1960's, but the problem appeared in its full form in this system. Given an arithmetic expression, the translator first created a sequence of arithmetic instructions, then transformed this sequence to eliminate redundant computations arising from the existence of common subexpressions (their term) and to reduce the number of accesses to memory. These transformations have been the subject of numerous investigations (Aho[6] gives a good set of references), and we now know that an optimal solution is inherently hard. It is interesting to note, however, that the compiler designers felt that "the near-optimum treatment of arithmetic expressions is simply not as complex a task as a similar treatment of 'housekeeping operations'."[2]

In addition to parsing and producing good code for the arithmetic expressions, the translator identified the other statements and transformed complex I/O lists into their component DO nests for treatment by the regular mechanisms of the rest of the compiler. The attempt here and in numerous other parts of the compiler to seek common mechanisms rather than create special case mechanisms is interesting in light of the overall complexity of the task and the amount of invention required for every part.

SUBSCRIPT AND DO STATEMENT OPTIMIZATION

The translator did not complete the translation of DO statements and subscripts; that was the function of Section 2. A symbolic index register corresponding to each particular subscript combination of a variable was created by the translator and existed until Section 5 had assigned registers. The function of Section 2 was to optimize the calculation of subscripts and DO control statements. The constant parts of the calculation were incorporated into operand addresses, operations involving DO control variables were transformed into index register increments when possible, loop independent parts of the calculation were removed from the loop, and the loop exit test was transformed to use one of the registers needed for indexing. A nest of DO loops for array calculations was sometimes replaced by a single loop in the generated code! Some of these transformations are now subsumed in more general optimizations, but today's production compilers rarely do as well.

FLOW ANALYSIS

The function of Sections 4 and 5 of the compiler was to assign real registers to the symbolic registers. Except for the symbolic registers and the assumption that they could all be assigned to real registers, the program on entry to Section 4 was complete. The basic task, therefore, was to assign the sym-

bolic registers to real registers in order to minimize the time spent loading and storing index registers. Section 4 of the compiler did a flow anlaysis of the program to determine the pattern and frequency of flow for use in Section 5, where the actual assignment was made.

Basic blocks ("a basic block is a stretch of program which has a single entry point and a single exit point"[2]) were found and a table of immediate predecessor blocks constructed. Here, then, is the beginning of the elegant and fast control flow algorithms of today. Basic blocks and predecessor (successor) relationships are inputs to these algorithms.

The other task performed by Section 4 was the computation of a probable frequency of execution of every predecessor edge. To do this a Monte Carlo "execution" of the program with initial weights assigned to each edge was developed. This method is no longer commonly used to identify frequently executed areas of a program; rather the program topology is used more directly but with less resultant precision.

REGISTER ASSIGNMENT

Using the edge execution frequencies, regions were formed so that registers could be assigned to the most frequently executed areas (usually innermost loops), then to the next most frequently executed areas, etc. until the entire program had been treated. When a region had been processed, its entry and exit conditions were recorded, i.e., the values which needed to be loaded on entry and stored on exit. A processed region was not reexamined when its containing region was processed, but the entry and exit conditions and whether or not it had any unassigned registers were used. The assignment of registers within a basic block used the "distance to next use" criterion to determine which register to displace when out of registers. "Activity bits" were used to determine the necessity of storing a value in a register for subsequent use if the register had to be reused. In case of register assignment mismatches across basic blocks, an attempt was made to permute the assignment.

This register assignment method was a phenomenal piece of work. The displacement algorithm for straight-line code was later proved optimal[7] for the "one-cost model";[8] a displacement costs the same whether you need to store the register contents or not. Until 1980, when Greg Chaitin[9] successfully applied a graph coloring algorithm to the global assignment of registers, most global assignments were essentially variants on the FORTRAN I approach.

RESULTS

Perhaps the best way of demonstrating the results of this project is to show an example of its output—an output which startled this author. The FORTRAN program in Figure 1 moved array B to array A in a double nest of DO loops. The assembly program in Figure 2 is the FORTRAN I compiler's output for this program and shows the move being done with one loop instead of the two expected from the source. (FORTRAN stored its arrays column-wise and backwards; the 704 subtracted the value in the index register from the address.)

```
        LXD ONE,1          load 1 into reg1
LOOP    CLA B + 1,1
        STO A + 1,1
        TXI * + 1,1,1      add 1 to reg1 and
                           goto next inst
        TXL LOOP,1,100     if reg1 ≤ 100
                           goto loop
        ...
ONE     , ,1               data value one
A       BES 100            reserve 100 locs,
                           ending with A
B       BES 100            reserve 100 locs,
                           ending with B
```

Figure 2—FORTRAN I translation of array move

In general the code produced by the compiler was not only locally efficient but globally as well. The output program did not contain long, precoded, predictable sequences but contained code optimized to run efficiently in its context (where the context included the whole program). The project was a success.

Jean Sammet states, "Its major technical contribution was to demonstrate that efficient object code could be produced by a compiler; as a result, it became clear that productivity of programmers could be significantly improved."[10] Another major contribution of the project is the influence it has had on compiler structure and techniques. Overall "... FORTRAN has probably had more impact on the computer field than any other single software development"[10]—because of the language, the technological impact on subsequent compilers, and the impetus it gave to widespread use of higher-level languages.

REFERENCES

1. Backus, John. "The History of FORTRAN I, II, and III." ACM SIGPLAN History of Programming Languages Conference, *SIGPLAN Notices* 13, 8 (August 1978), pp. 165–180.
2. Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. "The FORTRAN Automatic Coding System." *Proceedings Western Joint Computer Conference,* Los Angeles, 1957, pp. 188–198.
3. Sheridan, Peter B. "The Arithmetic Translator-Compiler of the IBM FORTRAN Automatic Coding System." *CACM* 2,2 (Feb. 1959), pp. 9–21.
4. Backus, J. W., and W. P. Heising. "FORTRAN." *IEEE Transactions on Electronic Computers,* EC-13,4 (August 1964), pp. 382–385.
5. Auslander, M. A., and M. E. Hopkins. "An Overview of the PL.8 Compiler." *Proceedings of the SIGPLAN Symposium on Compiler Construction,* June 1982 (to appear).
6. Aho, Alfred V. "Translator Writing Systems: Where Do They Stand?" *Computer,* 13,8 (August 1980), pp. 9–14.
7. Belady, L. A. "A Study of Replacement Algorithms for a Virtual-Storage Computer." *IBM Systems Journal,* 5,2 (1966), pp. 78–102.
8. Horwitz, L. P., R. M. Karp, R. E. Miller, and S. Winograd. "Index Register Allocation." *JACM* 13,1 (Jan. 1966), pp. 43–61.
9. Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register Allocation via Coloring." *Computer Languages* 6 (1981), pp. 47–57.
10. Sammet, Jean E. "History of IBM's Technical Contributions to High Level Programming Languages." *IBM Journal of Research and Development,* 25,5 (Sept. 1981), pp. 520–534.

Computing prior to FORTRAN

by R. W. BEMER
Honeywell Information Systems
Phoenix, Arizona

ABSTRACT

The life of the programmer in pre-FORTRAN days is characterized in modern terminology, indicating how strongly FORTRAN has changed the programmer's condition and working habits.

The 25 years since the introduction of FORTRAN covers most of programming as we know it, certainly in volume of usage. To minimize any possible communications gap, I have chosen to describe how it was before that watershed event by means of some of the terminology and buzzwords of today:

1. Conferences and published papers
2. Computer science education
3. Stored programming
4. Structured programming
5. Program portability
6. Performance measurement
7. Communications and timesharing
8. Compilers
9. Data independence
10. Software piece parts
11. Software packages

The technical history of early programming languages has been covered by many authors (it became a popular subject), so I'll confine my contribution to more general areas.

CONFERENCES AND PUBLISHED PAPERS

Publication of software papers in pre-FORTRAN days was far less prolific than now. And it wasn't yet "software." Papers on software techniques prior to FORTRAN are given,[2–42] as found (mostly) in Youden's "Computer Literature Bibliography 1946 to 1963."[1] They're given in best chronological order. To avoid duplication, sources with multiple papers are referenced separately, and the individual papers are given decimal notation.

Doing an analysis of the paper content of the early Joint Computer Conferences (the only continuing national meetings of that era) yields the counts shown in Table I. The last entry is the meeting at which FORTRAN was presented.

The summary pre-FORTRAN count is that of Table II.

TABLE I—Paper distribution of early JCCs

Year	JCC	Hardware	Applications	Software
1951	Eastern	16	2	0
1952	Eastern	26	0	0
1953	Western	8	11	0
1953	Eastern	18	4	1
1954	Western	8	14	0
1954	Eastern	9	7	2
1955	Western	6	16	1
1955	Eastern	6	9	1
1956	Western	18	10	6
1956	Eastern	29	0	0
1957	Western	28	4	3

TABLE II—Paper distribution by conference location

JCC	Hardware	Applications	Software	H/A	H/S
Eastern	104	22	4	4.7	26.0
Western	68	55	10	1.2	5.5
Total	172	77	14	2.2	12.3
%	65	29	5		

COMPUTER SCIENCE EDUCATION

This was just starting, and in just a few schools. When you hired a programmer then, you didn't ask about a degree in computer science; there weren't any. IBM used its Programmer's Aptitude Test as one screening method, and it worked somewhat, but people had a tendency to read more into it than was warranted.

A lot of us had our own pet questions, for we were taking them off the street. Magazine writers were curious about how one became a programmer. Dave Sayre had been a crystallographer, and Sid Noble and Art Bisguier were hired when I, an ex-movie set designer, advertised for chess players.

Although there may not have been enough collected theories to support specific degrees, the university people were all busy creating courses. The summer sessions at MIT and Michigan brought many practioners together. Language processors were being built there and at Purdue, Pennsylvania, Carnegie Tech, Case, UCLA, and many others.

STORED PROGRAMMING

Programs have always been "stored programs." The only difference is in where they were stored. In desk calculator days—in our heads. To program the IBM 601, one had to file notches in a phenolic strip, and they were stored in a box or hung on the machine. The IBM 604 was programmed by wires placed in plugboards, and often we stored them for reuse, if they were general enough. More often they were unwired for a new program (I wired about 700–800 60-step boards for the 604).

For the CPC the program was obviously in the cards. Bob Bosak and I devised a card system with 4 different tracks of 3-operand instructions, and so could feed a deck of cards continuously in a loop.

STRUCTURED PROGRAMMING

Structure in programs is generally ascribed to Wilkes, Wheeler, and Gill,[5] in their book on programming for the EDSAC. The subroutine was the first element of structure, and was generally accepted by programmers, particularly those writing interpretive systems.

We had no DO UNTILs or semaphores at our disposal, but many programs had a structure that's all but forgotten now. It was called "optimum programming," a method of placing sequential instructions just right on a magnetic drum, so they would be ready to read just after the previous instruction was completed.

PROGRAM PORTABILITY

The first way used to reconcile the differences between two types of computer was to recode the problem. The second way was to write a programmed interpretive emulator for one machine in the code of the other. When this resulted in performance degradation of 100:1 up to 1000:1 it lost a certain amount of favor.[43,44]

The third way was to use the source language of the interpreter and write another interpreter for the second machine. This had some success, because the degradation was often not very high (except for extremely dissimilar machines), and it could even run faster! Several of these were made.[44] If machines of today's speeds had suddenly been introduced then, this may have become commonplace; compilers might have a different role. Even now, after thousands of compilers, interpreters still enjoy a considerable vogue. The fourth way, with different compilers, did not to my knowledge receive substantial usage until FORTRANSIT, and even there the portability path from a 704 to a 650 was difficult because the 650 supported fewer index registers.

PERFORMANCE MEASUREMENT

Although no hardware instrumentation was available for probes, much performance measurement did occur. It was vital because the computers were too slow for the amount of calculation waiting to be performed. While working at Marquardt, I was chastised one day by my boss, for not shaving. It was caused by being up since the previous morning running a trajectory simulation on the CPC. Under such circumstances, everyone wanted programs to run as fast as they could. That was why the program optimizers for drum machines (like SOAP) were so heavily used.

When the 701 superseded the CPC, the balance between user and machine changed. One man at the RAND Corporation took two years to program a problem that ran in two minutes. He experienced considerable culture shock.

There was competition everywhere to have the fastest program for a given task, quite often a mathematical subroutine. When published, those subroutines always had timing associated so the user could plan wisely. The situation was much the same as in the early days of microcomputers. Jewel work was needed, and the domain was small enough to see and measure something. There was even competition between software and hardware people. The 705 engineers were shocked when a programmed divide ran faster than the hardware instruction —without firmware, they could not program a Newtonian iteration.

I suspect that FORTRAN itself had much to do with the temporary hibernation of performance evaluation. After programming in the other languages, it gave so much power because of the ease of use (and the efficiencies were incorporated for you in the compiler), that the number of user of computers could expand much more rapidly. It wasn't until operating systems came into heavy use that we rediscovered the need to prevent waste.

COMMUNICATIONS AND TIMESHARING

It wasn't Ethernet, but George Stibitz had tied into a relay computer by way of a Teletype—in 1940. SAGE was one of the first major projects to use direct inputs from communications lines. FORTRAN wasn't available when it began, and couldn't have been used for much of the job if it had, for it wasn't just a scientific problem.

Timesharing was just talk. The first time I find the word appearing is in a J. W. Forgie paper on the input-output system for the Lincoln TX-2 computer, concurrent with the 1957 FORTRAN paper. I proposed such usage in an article the next month; it was suggested that IBM should fire me, because that wasn't in line with their policy.

COMPILERS

Compilers existed before FORTRAN, but they were all rudimentary in comparison. Grace Hopper, chief pioneer of the concept, might have gone faster further if she had had the type of support given to Backus and his group. IT, A2 and A3 were true compilers, but they avoided interactions and optimization.

DATA INDEPENDENCE

This concept arose with the commercial compiler languages. Grace Hopper and company wrought the Data Division concept. Scientific languages all stuck to floating point, with integers for loop control.

Data structure was usually built into the program, and it didn't seem important, because hardly any interchange of programs took place between different computers. Even if that were possible one could not necessarily get the same answers due to different hardware characteristics.

SOFTWARE PIECE PARTS

Piece parts for software first came to attention at the first Software Engineering conference in 1968, proposed by Doug McIlroy. However, Bob Glass makes a convincing case[45] that they were in existence before FORTRAN, certainly via the SHARE organization. Indeed they were necessary to counteract the inefficiencies of working without such compilers.

SOFTWARE PACKAGES

In the modern sense the software package did not exist, for today they cost money. Before FORTRAN it was unthinkable to sell software, although the packages did exist. They were traded or given away. Examples are several general CPC boards, plus the many 650 packages published in the IBM Technical Newsletter No. 10.[27]

There is no doubt that packages existed. They were source programs for interpretation, not compiled source as today. A buzzword of the times was "abstraction." Douglas Aircraft had a "matrix abstraction," for example.[23] It manipulated matrices and performed combinatory functions. Ergo, if your problem could be expressed in matrix form, it could be solved. So it was urged that all problems be expressed this way, a not altogether natural way of use. But many of today's software packages have similar contortional requirements upon the user.

Codes for nuclear computation also fell in the category of software packages, even if they were exchanged in machine language form. Hundreds of these codes were disseminated.

SUMMARY

I'm enjoying the developments of today, but my pleasure is a bit spoiled by the terrible waste in software development, and so much poor software. It's tempting to recall Miniver Cheevy, who loved "the medieval grace of iron clothing." Software before FORTRAN could be considered quite medieval, even primitive, but there were certain graces.

From my starting in the computer field in early 1949, until FORTRAN arrived, I was either working too hard to see the Peter Principle in effect, or else it didn't exist in such a virulent form. It was exciting to build software then. We had management support and trust for whatever we thought was possible. The number of levels of management was low, and the control tenuous. I reported to John Backus in FORTRAN days, but never felt the slightest pressure. I looked upon him as a friend, not a menace. So today we have better tools and knowledge, and theories of program correctness and such. I don't think that they have added to the fun and excitement of Computing Prior To FORTRAN!

REFERENCES

1. Youden, W. W. "Computer Literature Bibliography 1946 to 1963." US Natl. Bur. Standards Misc. Publ. 266, 1965 Mar 31.
2. Wilkinson, J. H. "Coding on automatic digital computing machines." Report Conf. on High Speed Automatic Calculating-Machines, Univ. Math. Lab., Cambridge, England, 1949 Jun 22–25, 28–35.
3. Huskey, H. D. "Semiautomatic instruction on the Zephyr." Proc. 2nd Symp. on Large-scale Digital Calculating Machinery, Cambridge, MA, 1949 Sep 13–16, 83–90, Harvard U. Press, 1951, Annals Vol. 26.
4. Stowe, L. "Programming." Office of Naval Research Seminar on Data Handling and Automatic Computing, Washington, DC, 1951 Feb 26–Mar 6, 79–84.
5. Wilkes, M. V., D. J. Wheeler, S. Gill. "The Preparation of Programs for a Digital Computer." Addison-Wesley Press, Cambridge, MA, 1951.
6. Proc. ACM Conf., 1952 May, Pittsburgh, PA.
6.1 Adams, C. W. "Small problems on large computers." 99–102.
6.2 Lipkis, R. "The use of subroutines on SWAC." 231–234.
6.3 Wheeler, D. J. "The use of subroutines in programmes," 235–236.
6.4 Carr, J. W. III. "Progress of the Whirlwind computer towards an automatic programming procedure." 237–242.
6.5 Hopper, G. M. "The education of a computer." 243–250.
7. Proc. ACM Conf., 1952, Toronto.
7.1 Ridgway, R. K. "Compiling routines." 1–5.
7.2 Isaac, E. J. "Machine aids to coding." 17–28.
7.3 Strachey, C. S. "Logical or non-mathematical programmes." 46–49.
7.4 Bennett, J. M., D. G. Prinz, M. L. Woods. "Interpretative subroutines." 81–87.
8. Rutishauser, H. "Automatische Rechenplanfertigung bei programmgesteuterten Rechenmaschinen." Mitteilung aus dem Institut fur angewandte Mathematik, Basel, 1952, 1–45.
9. Rochester, N. "Symbolic programming." PGEC (IRE Transactions on Electronic Computers), Vol. EC-2, No. 1, New York, 1953 Mar, 10–15. LC Card 57-39723.
10. Hopper, G. M. "Compiling routines." Computers and Automation 2, No. 4, 1953 May, 1–5.
11. Hopper, G. M., J. W. Mauchly. "Influence of programming techniques on the design of computers." Proc. IRE 41, No. 10, 1953 Oct, 1250–54.
12. Bouricius, W. G. "Operating experience with the Los Alamos 701." Proc. Eastern Joint Comput. Conf., 1953 Dec 8–10, 45–47.
13. Bennett, J. M., A. E. Glennie. "Programming for high-speed digital calculating machines." In "Faster than Thought." B. V. Bowden, London, Pitman, 1953, 101–116. LC Card 54-15305.
14. Wilkes, M. V. "The use of a 'Floating Address' system for orders in an automatic digital computer." Proc. Camb. Phil. Soc., 49, Part I, 1953, 84.
15. Laning, J. H., N. Zierler. "A program for translation of mathematical equations for Whirlwind I." Engg. Memo. E-364, M.I.T. Instr. Lab., 1954 Jan.
16. Backus, J. W. "The IBM 701 speed-coding system." J. ACM 1, No. 1, 1954 Jan, 4–6.
17. Symposium on Automatic Programming for Digital Computers, Office of Naval Research, Washington, DC, 1954 May 13–14. LC Card 56-60789 rev.
17.1 G. Hopper. "G. M. Automatic programming definitions." 1–5.
17.2 Moser, N. B. "Compiler method of automatic programming." 15–21.
17.3 Waite, J. "Editing generators." 22–29.
17.4 Goldfinger, R. "New York University compiler system." 30–33.
17.5 Holberton, F. E. "Application of automatic coding to logical processes." 34–39.
17.6 Adams, C. W., J. H. Laning, Jr. "The M.I.T. systems of automatic coding: Comprehensive, Summer Session and Algebraic." 40–68.
17.7 Muller, D. E. "Interpretive routines in the Illiac library." 69–73.
17.8 Gorn, S. "Planning universal semiautomatic coding." 74–83.
17.9 Brown, J. H., J. W. Carr III. "Automatic programming and its development on the MIDAC." 84–98.
17.10 Livingston, H. M. "Automatic programming on the Burroughs Laboratory computer." 99–105.
17.11 Backus, J. W., H. Herrick. "IBM 701 speedcoding and other automatic-programming systems." 106–113.
17.12 Elmore, M. "The LMO edit compiler," 114–116.
17.13 Keller, A., R. A. Butterworth. "Programming for the IBM 701 electronic data processing machine with repetitively used functions." 117–149.
18. Jones, J. L. "A survey of automatic coding techniques for digital computers." M.S. Thesis, M.I.T., 1954 May.
19. Proc. Eastern Joint Comput. Conf. 1954 Dec 8–10.
19.1 Rice, R. Jr. "Why not try a plugboard?" 4–10.
19.2 Krider, L. D. "Applications of automatic coding to small calculators." 64–67.
20. Rutishauser, H. "Some programming techniques for the ERMETH." J. ACM 2, No. 1, 1955 Jan, 1–4.
21. Herbst, E., N. Metropolis, M. B. Wells. "Analysis of problem codes on the MANIAC." M.T.A.C. 9, No. 49, 1955 Jan, 14–20.
22. Hopper, G. M. "Automatic programming of digital computers." Proc. High Speed Comput. Conf., Baton Rouge, LA, 1955 Feb 16, 113–118. LC Card 57-63206.
23. Denke, P. H., I. V. Boldt. "A general digital computer program for static stress analysis." Proc. Western Joint Comput. Conf., 1955 Mar 1–3, 72–78.
24. Bradshaw, T. F. "Automatic data processing methods." Proc. Auto.

Data Proc. Conf., Cambridge, MA, 1955 Sep 8–9, 3–27, Harvard U. Press, 1956.
25. "Automatic programming the A-2 compiler system." Parts 1 and 2, Computers and Automation, 4, Nos. 9 and 10, 1955 Sep and Oct.
26. Electronic Digital Computers and Information Processing, Darmstadt, Germany, 1955 Oct 25–27, F Vieweg, Braunschweig, 1956. LC Card 59-18764.
26.1 Rutishauser, H. "Methods to simplify programming, 5 years work with the Z4 computer" (German), 26–30.
26.2 Samelson, K. "Problems of programming techniques" (German), 141–142.
26.3 Lehmann, M. J. "Automatic computer programming" (German), 143.
26.4 Loopstra, B. J. "Processing of formulas by machines." 146–147.
26.5 Thuring, B. "The automatic programming of Univac by the A-2 compiler system." (German), 154–156.
27. Technical Newsletter No. 10, IBM Applied Science Division, New York 1955 Oct.
27.1 Ruthrauff, R. E. "Symbolic coding and assembly for the IBM Type 650." 5–14.
27.2 Horner, J. T. "Relative programming for the IBM Type 650." 15–27.
27.3 Bosak, R. "Development of a floating decimal abstract coding system (FACS)." 28–30.
27.4 Bemer, R. W. et al. (Lockheed MSD). "A general utility system for the IBM Type 650." 31–48.
27.5 Mandelin, A. R., K. D. Weaver. "A selective automonitoring tracing routine called SAM." 49–62.
27.6 Battin, R. H., R. J. O'Keefe, M. E. Petrick. "The MIT Instrumentation Laboratory automatic coding 650 program." 63–79.
27.7 Titus, C. K. "An integrated computation system for the IBM 650." 80–89.
28. Hume, J. N. P., B. H. Worsley. "TRANSCODE, a system of automatic coding for FERUT." J. ACM 2, No. 4, 1955 Oct, 243–252.
29. T. Gorman, T. P., G. Kelly, R. B. Reddy. "Automatic coding for the IBM 701." J. ACM 2, No. 4, 1955 Oct, 253–261.
30. C. Adams, C. W. "Developments in programming research." Proc. Eastern Joint Computer Conf., 1955 Nov 7–9, 75–79.
31. Gordon, B. "An optimizing program for the IBM 650." J. ACM 3, No. 1, 1956 Jan, 3–5.
32. Proc. High Speed Computer Conference, Baton Rouge, LA, 1956 Feb. LC Card 57-63206.
32.1 Hopper, G. M. "Automatic coding techniques 1955." 6–12.
32.2 Heller, J. "Mathematical service routines." 151–153.
32.3 Perry, D. P. "Specifications for an automatic matrix program." 210–215.
33. Proc. Western Joint Computer Conf. 1956 Feb 7–9.
33.1 Ross, D. T. "Gestalt programming, a new concept in automatic programming." 5–9.
33.2 Grems, M., R. E. Porter. "A truly automatic programming system." 10–20.
33.3 Moncrieff, B. "An automatic supervisor for the IBM 702." 21–25.
33.4 Bemer, R. W. "PRINT I, a proposed coding system for the IBM Type 705." 45–48.
33.5 Goldfinger, R. "The IBM Type 705 Autocoder." 49–51.
33.6 Meek, H. V. "An experimental monitoring routine for the IBM 705." 68–69.
34. Brooker, R. A. "The programming strategy used with the Manchester University Mark I computer." IEEE Conf. on Digital Computer Techniques, Suppl. Part B, Vol. 103, London, 1956 Apr 9–13.
35. Perkins, R. "EASIAC, a pseudo-computer." J. ACM 3, No. 2, 1956 Apr, 65–72.
36. Bauer, W. F. "An integrated computation system for the ERA-1103." J. ACM 3, No. 3, 1956 Jul, 181–185.
37. Blum, E. K. "Automatic digital encoding system II (ADES II)." Proc. ACM Conf. 1956, 29.
38. Chipps, J., M. Koschmann, S. Orgel, A. J. Perlis, J. Smith. "A mathematical language compiler." Proc. ACM Conf. 1956, 31.
39. Symp. Advanced Programming Methods for Digital Computers. 1956 June 28–29, ONR Report ACR-15, 1956 Oct.
39.1 Hopper, G. M. "The interlude 1954 to 1956." 1–2.
39.2 Wegstein, J. H. "Automatic coding principles." 3–6.
39.3 Thompson, C. E. "Development of common language automatic programming systems." 7–14.
39.4 Benington, H. D. "Production of large computer programs." 15–28.
39.5 Jones, F. "SHARE, a study in the reduction of redundant programming efforts through the promotion of inter-installation communication." 29–34.
39.6 Carr, J. W. III, B. Arden, "Advanced programming techniques with smaller computers." 35–38.
39.7 Goldstein, M. "Computing at Los Alamos, Group T-1." 39–44.
39.8 Wells, M. "Coding for the MANIAC." 45–48.
39.9 Holberton, F. E. "Proposed advanced coding system for the UNIVAC-LARC." 49–56.
39.10 Waite, J. H. Jr. "RCA approach to automatic coding for commercial problems." 57–66.
39.11 Selfridge, R. G. "The PACT compiler for the 701." 67–70.
39.12 Blum, E. K. "Automatic digital encoding system II." 71–76.
40. J. ACM 4, No. 4, 1956 Oct.
40.1 Melahn, W. S. "A description of a cooperative venture in the production of an automatic coding system." 266–271.
40.2 Baker, C. L. "The PACT I coding system for the IBM Type 701." 272–278.
40.3 Mock, O. R. "The logical organization of the PACT I compiler." 279–287.
40.4 Miller, R. C. Jr., B. G. Oldfield. "Producing computer instructions for the PACT I compiler." 288–291.
40.5 Hempstead, G., J. I. Schwartz. "PACT loop expansion." 292–298.
40.6 Derr, J. I., R. C. Luke. "Semiautomatic allocation of data storage for PACT I." 299–308.
40.7 Greenwald, I. D., H. G. Martin. "Conclusions after using the PACT I advanced coding technique." 309–313.
41. Symp. on Automatic Coding, Franklin Institute, Philadelphia, PA, 1957 Jan 24–25, Monograph No. 3.
41.1 Petersen, R. M. "Automatic coding at G.E." 3–16.
41.2 Katz, C. "Systems of debugging automatic coding." 17–28.
41.3 Bemer, R. W. "PRINT I, an automatic coding system for the IBM 705." 29–38.
41.4 Kinzler, H. M., P. M. Moskowitz. "The procedure translator, a system of automatic coding." 39–56.
41.5 McGee, R. C. "Omnicode, a common language programming system." 57–70.
41.6 McGinn, L. C. "A matrix compiler for Univac." 71–86.
41.7 Perlis, A. J., J. W. Smith. "A mathematical language compiler." 87–102.
41.8 Yowell, E. C. "A mechanized approach to automatic coding." 103 ff.
42. Proc. Western Joint Computer Conf. 1957 Feb 26–28.
42.1 Grems, M. D., R. K. Smith, W. Stadler. "Diagnostic techniques improve reliability." 172–178.
42.2 Baskus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, R. Nutt. "The Fortran automatic coding system." 188–197.
42.3 Newell, A., J. C. Shaw. "Programming the Logic theory machine." 230–240.
43. Table, Comm. ACM 1, No. 2, 4.
44. Tables, Comm. ACM 1, No. 11, 5–6.
45. Glass, R. L. "Software parts nostalgia." in Reader's Forum, Datamation 27, No. 12, 1981 Nov, 245–247.

History of FORTRAN standardization

by MARTIN N. GREENFIELD
Honeywell Information Systems, Inc.
Billerica, Massachusetts

ABSTRACT

The history of FORTRAN Standardization, ranging from the original efforts in the early 60s up to the present, is presented. Some of the precedent-setting development during the initial cycle in handling problems common to all language standardization is discussed. The background in introducing some of the features in FORTRAN 77 is covered. The nature and reasoning behind the current activity are described.

There is an interesting and appropriate introduction in my daughter's college text on FORTRAN. It reads, "After you have learned some of the language, you will show off your sophistication by knocking its lack of elegance. Everybody does. After you learn a little bit more, you will appreciate that it is the way to really get your work done." FORTRAN has for most of life been the blue-collar worker of the programming language set. What it lacked in savoire-faire and style, it returned in cost effectiveness. Those working with FORTRAN pioneered the way for the acceptance of higher-level languages and their standardization. Those who have influenced its development were continually aware of the underlying fact that the language, first and foremost, must remain an efficient tool for producing results.

FORTRAN standardization dates back to early 1960. The language had just been selected by industry over ALGOL as the language for scientific and engineering work. The major vendors recognized the requirement to provide FORTRAN compilers in order to compete with IBM. The general strategy was to provide a compiler with the functionality of the 704/709 FORTRAN and to add features as a competitive inducement. The impact of these added features was two-edged. Although they contributed to the development of the language, they threatened to splinter it into a myriad of uncontrolled dialects. Adding to the problem, a rigorous definition of the language did not exist, even within IBM.

Fortunately, at that time ASA (subsequently to become ANSI) and BEMA (subsequently, CBEMA) undertook sponsorship of a massive standardization effort covering a broad variety of data processing areas. Someone had the brave idea of including languages. The ASA X3.4 committee conducted a survey of existing programming languages. FORTRAN, COBOL, and ALGOL were selected as the candidates for standardization. X3.4 at their May 1962 meeting established the X3.4.3 committee and directed it to standardize the FORTRAN language.

INITIAL STANDARDIZATION (1962–1966)

Bill Heising, of IBM, was appointed as the initial chairman of X3.4.3. Bill sent invitations to potentially interested groups to attend a formation meeting. Accompanying the invitations was a document written by Bill together with Dick Ridgeway that was proposed as the starting draft for the standardization effort. This Heising-Ridgeway FORTRAN was based upon the forthcoming FORTRAN IV.

The initial meeting of X3.4.3 was held at the BEMA Headquarters in New York City on August 14, 1962. This makes 1982 both the twenty-fifth anniversary of FORTRAN and the twentieth anniversary of the start of its standardization. At this August 1962 meeting, there was a consensus to undertake the standardization work. The scope and criteria of the effort were established.

X3.4.3 assumed the role of parent and policy maker and delegated all the chores below that to two working subcommittees. As such, X3.4.3 met only about twice a year. X3.4.3 originally had about two dozen regular members. All the major hardware vendors were represented. A number of user groups (SHARE, Honeywell Users Association, USE, VIM, IBM 1620 Users, CO-OP) participated. Some software houses (CSC, CUC) and universities (Wisconsin, Penn State, UCSD) had members.

The parent X3.4.3 did thrash out some very controversial issues. One of recall concerned a proposal from those working with the then new character-addressable hardware. They could save much space by not allocating the same space to integer and logical data as was allocated to reals. In fact, they preferred not to have any fixed storage relationship between the data types. Logicals could be packed into one byte or less. Double precisions could occupy just two or three more bytes than reals. Their arguments centered about the concept that a language standard should not be as hardware biased as the word-storage-unit relationship implies. After some impassioned discussions the heavy dependence of FORTRAN on storage association for efficiency and the dominance of word-addressable processors won.

Most of the actual standardization work was handled by the two subcommittees. X3.4.3-IV was responsible for the standardization of the language based on FORTRAN IV, while X3.4.3-II was to do the same for FORTRAN II.

The subcommittees were small compared to the size of groups currently developing draft standards. It was fortunate, because it provided an efficient working arrangement and uninterrupted participation. Little time was lost in having to bring new members up to date. The regular members of X3.4.3-IV were

Martin N. Greenfield, Honeywell, chairman
Richard K. Ridgeway, IBM, editor
Caral Sampson (Giammo), Philco, secretary
Tom Martin, SHARE and Westinghouse
Geraldine Zimmerman (Bowen), UNIVAC
Lou Gatt, CSC
Ken Tiede, CDC
Carl Bailey, CO-OP and Sandia
Bob Mitchell, CO-OP and UCSD

Along with the X3.4.3 chairmanship responsibilities, Bill Heising was a very active participant in the effort of the

X3.4.3-IV subcommittee. Others from X3.4.3 participated from time to time, but the bulk of the effort was done by the group above.

The work proceeded during the following two years. Although some meetings were hosted at the sites of the different members throughout the country, the bulk of the sessions were either at BEMA headquarters or at the IBM program development center in the Time-Life building, both in New York City.

The initial FORTRAN IV compilers were all under development while the work of X3.4.3-IV was in progress. The members of X3.4.3-IV were all either responsible or could direct changes in their language specifications for these implementations. It was a unique situation, where language changes adopted by the subcommittee were incorporated into the compilers almost immediately. I have always felt that the actual standardization of FORTRAN stemmed from the discussions, understandings, and agreements of X3.4.3-IV rather than from formal text that followed some years later.

The undocumented agreement within X3.4.3-IV was that the standard would not incorporate any feature that was not planned for all the implementations. Since the starting point for all of our language specs was the IBM-proposed language, it followed that the draft most closely represented the IBM implementation. It was by no means a slavish copy. For one thing, there were no rigorous specifications within IBM of much of FORTRAN IV that could have been copied. This was particularly true in the input-output area. There were some features that IBM meant to carry into FORTRAN IV from their FORTRAN II implementations in order to protect their user's investment. Unfortunately, some of their FORTRAN II implementations contained some objectionable shortcuts. For example, a constant could precede a variable and imply a multiplication operator (5L meant 5 * L). To their credit, there was never much of a hassle with those from IBM in deleting features that were objectionable carryovers from existing implementations of FORTRAN II. I believe they were sincerely motivated in working toward the best long-term interests of the language. Another change of note was that the DATA statement syntax was altered from the way IBM was implementing it. It was originally specified with parentheses rather than slashes as the delimiter for the list of constants.

Having no precedents, X3.4.3-IV had to address numerous problems common to all language standardization. Much of this we take for granted now, but there was nothing to turn to at the time. There were discussions as to whether there should be a standard. There is a penalty. The presence of a standard implies the pressure of conformance over a long period to a static document. This could certainly serve to limit the growth and development of the language. Even if motivated, the implementor, constrained to conform, would be prohibited from adding extensions. Programs requiring nonstandard functionality could not be developed. Unanticipated requirements could not be satisfied until after the many years needed for a new revision had elapsed. The difficulties of specification of a standard could artificially limit the functionality because it might be too difficult or unwieldy to word the true restrictions. Once a feature was standardized, its life would be semi-eternal even if the feature were a mistake. The result is that generally a very conservative posture is assumed in deciding what is to be included. The potentially useful but untested functionality usually doesn't make it. These are all penalties to be weighed against the advantages of portability and communication that standardization could provide.

A partial answer to these objections to having a standard was worked into the interpretation section of the standard and has been carried into all the subsequent revisions of FORTRAN standards. The standard is to be interpreted as permissive. That is, that the standard serves only to specify a part, not all, of the language. Anything not specified isn't unclean, bad, immoral, or even not kosher. It is simply not specified. Similarly, things that are prohibited are things that are simply uninterpreted when violated. A standard program must be limited to what is specified in order to conform, but the same is not true for a processor. A processor may provide array processing, but it must handle standard subscripting in the conforming manner. Thus, an experimental extension can be available in a standard processor. The processor must be able to properly interpret standard programs, but may also provide interpretation to a nonconforming program. The choice is then available to conform or not as the economics dictate. Some nonconformance is encouraged.

The subcommittee decided that the target audience for the standard would be compiler implementors or those on users' staffs who were the FORTRAN support experts. It was felt that this latter group were competent in being able to implement a compiler; so, in effect, there was just the implementor that characterized the audience. It was felt that the standard should specify the requirements for a standard conforming program rather than a compiler, but I don't believe this was apparent in the document.

The decision was made to use English rather than some metalanguage. This was in the belief that the description of the semantics was the difficult problem. Use of a metalanguage would not help there. A metalanguage was at best only assisting in tackling the easiest part of the description. It was felt that its precision did not compensate for the need to become familiar with the added formality. Interestingly, the one most useful area that could have been served by a precise description using a metalanguage is the FORMAT statement. There was actually an error in the way it was specified in the standard. I am still unaware of a complete and precise description of that statement using some metalanguage.

There were many challenges to our ability to describe. CDC had proposed that the new logical IF be a two-way branch analogous to the arithmetic IF. This would have saved us much descriptive grief in handling the concept of a compound statement that had in this one place crept into the language. For example, we could no longer accurately state that every statement could have a label. It also led to an unduly harsh restriction prohibiting some forms of the logical IF from being the terminal statement of a DO loop.

The greatest challenge to our descriptive capabilities was presented by the extended range of a DO loop. (There are some who would claim that this honor should go to the concept of second-level definition). All the implementations of FORTRAN IV being developed allowed a more liberal extended range than the one appearing in the standard. The committee would have been amenable to a less restrictive extended range if it could only have been appropriately de-

scribed. Everyone tried at least twice. Any definition that included statements about the sanctity of the contents of index registers, although reflecting the real concern, was inappropriate. The definition finally adopted was an accurate subset of what everyone was providing. The definition was felt to be reasonably understandable. Those of you who have struggled with that definition and its prerequisite concept of completely nested nest might quibble about the description being reasonably understandable. This is only because you did not struggle with some of the descriptions that were rejected. This was certainly an instance where the ability to describe limited the technical content. I believe that there is some of this effect in most standards. It is deluding not to admit it.

There were a surprisingly small number of new terms that had to be coined. Terminology common to several manuals was preferred, since it would already be familiar. The, usually missing, rigorous definitions of these terms had to be developed. Among the newly coined terms were *definition and undefinition* and their related states of being *defined or undefined, reference* as applied to data and to procedures, and *intrinsic function.* The term *intrinsic function* had its birth at a bull session during one of our meetings. We had been discussing the classification of functions, using the then customary terms *open and closed functions. Open functions* meant in-line code; *closed* meant some internal procedure. There was the concern that the absolute function (ABS), generally thought of as the obvious prototype of in-line code, was no longer such when the argument was of complex data type. Further, the tightening techniques being developed for some codes might make it attractive to put more formerly closed functions in line for greater speed. Besides, the terminology smacked of a particular implementation consideration. Lou Gatt piped up with the idea that the basic characteristic of these functions was that they were cast into or intrinsic to the processor, and that therefore we should call them *intrinsic functions.* So credit for this term belongs to Lou.

We were later to find that a subtle side benefit of our standards work was the widespread use of the terminology used in the standard. Our terminology was generally accepted and replaced the proliferation of some terms for certain actions and objects that were in use before without any rigorous and agreed-upon definitions.

The subcommittee gave some consideration to how to enforce the standard through use of acceptance procedures. Two hurdles caused us to turn away from further work in this area. We realized that an exhaustive verification was not possible. It might be misleading to develop some partial verification package that might be construed as being total. Any such official package might be misused as a standard performance benchmark. The second hurdle was simply not having the manpower to do the work. It was hoped that market pressures would lead to some accepted verification means, but we didn't have the resources.

The subcommittee X3.4.3-II drafting the specification based on FORTRAN II was even smaller than that of X3.4.3-IV. Their membership, as I recall, was

Jack Palmer, IBM, chairman
Irwin Boris, Honeywell
Charles Davidson, University of Wisconsin, 1620 Users Group
Don Laird, Penn State University
Bob Brunelle, Honeywell Users and NIH
Bernice Weizenhoffer, IBM
Robert Hux, RCA

Partly because their target was better defined, X3.4.3-II completed their work and the first draft FORTRAN standard almost a year before X3.4.3-IV finished. They were directed by X3.4.3 to keep the draft on hold until X3.4.3-IV had its draft ready. There was still the hope at that time that a compatible standard representing FORTRAN II and FORTRAN IV could be produced.

Subsequently, X3.4.3 decided that there should be a standard for the full language and a standard that was a proper subset of the full language. It was not possible to use the X3.4.3-II draft as the subset because of the number of totally incompatible differences between FORTRAN II and FORTRAN IV. The result was that the work of the X3.4.3-II was discarded. The subset was created by deleting text from the X3.4.3-IV draft. I hope that the draft produced by X3.4.3-II finds its way into the archives of FORTRAN history. Through no fault of its own, the effort of X3.4.3-II was not incorporated. Their work is historically significant in that it was the first completed draft of any language standard.

In October 1964, the two proposed draft standards were published in the *Communications of the ACM.* These were the first standards ever proposed for a programming language. They severely taxed the editing and approval mechanisms of ASA and BEMA. Draft standards up to then rarely needed more than a page of text and that page usually had room for the diagrams of the screw thread. The inability to rigorously check for conformance was shattering. It is little wonder that it took almost a year and a half before final approval was obtained in April 1966. The full language standard was designated ASA X3.9–1966 FORTRAN and the subset, ASA X3.10–1966 Basic FORTRAN.

Early in the standardization effort, the European Computer Manufacturers Association (ECMA) submitted a proposed draft of what they felt the full language should contain. Since they were separated from the developments in this country, their proposal fell between the Basic FORTRAN and the full FORTRAN. X3.4.3 voted to standardize on only two levels. When FORTRAN standardization was considered by the International Standards Organization, the ANSI form and content was chosen as the basis. The ECMA subset in ANSI form was added as the intermediate of three levels.

INTERPRETATIONS PERIOD (1967–1970)

Late in 1967, the then disbanded X3.4.3 was recalled primarily through the urging of the National Bureau of Standards. NBS, and in particular, Betty Holberton, was attempting to produce a Federal standard for FORTRAN. Betty's examination of the X3.9–1966 FORTRAN standard led her to submit a few dozen questions on interpretation. Other clarification inquiries were received from other sources. The FORTRAN

group was revived as the only body that could authoritatively provide the clarifications. This process turned out to be more tedious and demanding than the standardization effort itself. Because we were dealing with an approved standard, not a single comma could be altered without going through the same long approval cycle. Interpretations had to be based on a rationale developed from the standard's actual wording and not from what even the original authors felt it should have been. Two interpretation reports were published, but they took over three years of meetings to produce. The difficulty of that interpretation effort has had its impact on the form of the standard for FORTRAN 77. Those who participated in both efforts took pains to carefully examine every phrase to reduce to a minimum the chance of misinterpretation.

By 1968, enough extensions had appeared in the more current implementations to have the FORTRAN group appoint someone to study whether these extensions should be standardized. Frank Engel was selected as the one to conduct this study. Following Frank's report, in January 1969, the committee voted not to reaffirm X3.9–1966 when its review period came up, but to provide a new draft standard.

The committee had a succession of chairmen during this period. Bill Heising was replaced by Dick Ridgeway. Heising later returned as chairman prior to having Dennis Hamilton assume the position. In September 1970 Frank Engle assumed the chair and was to last throughout the development of FORTRAN 77. Frank's tenure, the longest of any chairman, ended in October 1977 when Jeanne Adams, the current holder was appointed.

FORTRAN 77 (1970–1978)

By early 1970 the interpretation activity had had it. There were unresolved issues that could not be handled within the wording of X3.9–1966. They decided that since the standard had to be reviewed and replaced or reaffirmed by 1971, it would be more productive to abandon the clarification work and devote their energy to a replacement. It is interesting that the most pessimistic schedule proposed at that time had the draft available by the end of 1971. The initial effort did not sharpen the ability to predict the time required to develop a standard.

Criteria and goals were drawn up for what would become FORTRAN 77. Their jist was to evolve the language, keep it approximately the same "size," and be sure that its efficiency features would not be impaired. It was important that the standard should be in a much more expository form and be meaningful to a larger and less knowledgeable audience. The form of the revision was chosen to be a single standard containing two subset levels. A later decision removed the intermediate subset. Because of the single standard approach, ASA X3.10–1966 Basic FORTRAN would be discarded (i.e., not reaffirmed).

They further voted that the new draft standard would be an evolutionary development that would not invalidate programs written in the language of the 1966 standards. This position was subsequently modified in two significant areas. The Hollerith data type was deleted because it was replaced by the more functional and machine independent character data type. The zero trip DO loop was specified. Actually, the control conditions for a zero trip DO were conditions that were nonconforming to the 1966 standard. However, since many implementations interpreted these conditions by executing the statements in the range once, many programs would have to be adjusted. There were objections even though the issue related to programs that were technically not standard conforming.

Six years of effort went into FORTRAN 77. That standard represented work on over two hundred technical proposals from all over the world. The cost of the effort was in excess of two million dollars. The text was almost six times the size of X3.9–1966. While some very significant language additions are present, the expansion was largely attributable to the effort to make the document more understandable. The draft had a completely different organization than the 1966 standard. Emphasis was on clarity rather than compactness and nonredundancy. Extensive use was made of word processing, a concordance tool (KWIC), computer graphics, and direct transcription to hard copy and fiche facilities. The very extensive editing, consistency checking, rewriting, and the distribution of the numerous interim drafts were made possible only by some herculean efforts of the two editors, Lloyd Campbell and J. C. Noll. The editorial staff of ANSI was presented with a camera-ready copy of the draft for publication, thus avoiding the errors that might have been introduced by an ANSI stage of processing.

The features of the draft standard were publicly presented by X3J3 members at the West Coast FORTRAN Forum held in Anaheim, California, in February 1976. The following month, the draft standard appeared in a special edition of SIGPLAN Notices. An East Coast FORTRAN Forum was later held at the National Bureau of Standards in Gaithersburg, Maryland. Smaller groups of X3J3 members presented sessions on the new language standard at meetings of professional societies, user groups, and at conferences. The public review was initiated and comments were solicited.

During the period of public comment and review 289 responses consisting of 1225 pages were received. This was probably the largest outpouring to any proposed standard as of that time. It took almost a year for the committee to complete the responses. The number of public comments was evidence of the large, present, and continuing interest in the language and the understandability of the document. Despite the earlier extensive checking by the committee, there were a number of changes and corrections incorporated because of the comments.

The major issue, as measured by the volume of comments received, was to add some facility in support of structured programming. There were a significant number of preprocessors available that enabled FORTRAN programmers to develop programs using statements such as IF . . .THEN . . .ELSE, DO WHILE, DO UNTIL, CASE statements and the like. These preprocessors would convert the source into valid FORTRAN statements. There was a clear requirement to place some of the facility directly into the language. In responding, the committee felt that although some facility should be added, there were many syntactic variations and an insufficient experience basis to select and standardize many of the constructs. They took an appropriately conservative ac-

tion of adding only the BLOCK IF constructs. This addition, as specified by Walt Brainerd, provided most of the important capability requested. It avoided adding and being stuck with some of the other constructs such as DO UNTIL that are already falling into disuse because of superior forms.

The reaction of X3J3 to the structured programming requests is a good example of how a responsible committee should avoid an over reaction that would prematurely add features that it would shortly regret. Unfortunately, there are counter examples in FORTRAN 77 such as the ENTRY statement and the alternate RETURN that should not have been included.

Approval of the standard came in April 1978. The official designation is American National Standard programming language FORTRAN X3.9–1978. In March 1980, an International Standards Organization FORTRAN based upon the ANSI standard and known as ISO 1539–1980 was approved by twenty-one countries. This document is essentially a cover that references ANS X3.9–1978 for the English text and the French standard NF Z65–110 for the French text. In September 1980, the US Federal Standard for FORTRAN (FIPS PUB 69), incorporating by reference X3.9–1978, was approved.

Next Revision (1978–Present)

Following the approvals of the FORTRAN 77 standards, the expected lull in the standardization activity did not materialize. There was pressure to consider the additions received during the public response to FORTRAN 77 that were rejected as premature. New FORTRAN implementations were incorporating additions such as a free form for statements. CODASYL had established a group (FORTRAN Data Base Language Committee, FDBLC) to provide a foundation for the addition of a major database augmentation to the language. ISA and the Purdue Workshop had developed standards addressing issues of tasking, file synchronization, and event management. An interest in a graphic addition was looming.

The committee devoted its time during 1978 to the planning for the future direction of the language. They solicited the thoughts of many other interested groups such as ISA, CODASYL, IEEE, and SIGNUM who were known to be interested in FORTRAN extensions. The level of interaction with international bodies was dramatically increased. International meetings under the informal structure of ISO FORTRAN Experts Group were convened in Europe during 1977, 1978, 1979, and 1980. All of this activity was in the attempt to obtain a broad basis of experience upon which to develop the successor standard.

X3J3 felt confident it could manage desired additional language features such as free form for statements, new control and data structures, and even most of the array handling. They even felt comfortable in handling the removal of some of the basic restrictions such as dynamic storage allocation, recursion, identification via storage association, and storage related precision. However, they were unsure of how to cope with major augmentations such as the database and graphics handling. The additions would be expensive, not only in the cost of the processors, but in the breadth of the language that would be impacted. Even those not interested in these features would be paying a price in terms of what they would have to know to work with the language. The committee knew it did not have the expertise to select among the competing forms of database and graphics facilities. It wanted to be able to responsibly control these augmentations and yet didn't see how a single committee could commandeer all of the expertise needed for this development and management.

The answer is one that is still evolving and is a change in the architecture of the language. It is called the core plus modules approach. The plan for the language revision, called FORTRAN 8X, is to specify a relatively small, general purpose, self sustaining core language. There would be added features that would modernize and streamline the language. The size of this core language would not exceed that of FORTRAN 77 because there would be compensating deletions. The core would be provided with very strong facilities to be able to interface with modules whose use could be selectively chosen. These modules would have to follow some broad conventions established by the committee to qualify as part of the FORTRAN family.

There would be two classes of modules, language extension modules and application modules. A language extension module would be developed by X3J3 and would represent features that exceed the general purpose scope of the core. It might also consist of features that were desirable for addition, but that had not been subject to sufficient implementation or usage experience. An extension module could not be modified and approved for standardization without reconsideration of the core and all of the other language extension modules.

One special language extension module would be called the Obsolete (Transition) Features Module. This module would contain all of the features needed for compatibility with the previous revision (FORTRAN 77). Features being dropped in a revision would survive for one cycle in this module. When this module was employed, it would override any incompatible features of the current language.

An applications module would probably be specified by some group external to X3J3 and would address features of some special domain. Examples might be one (or more) of the database facilities, a query capability, or a graphics addition. These would probably take the form of a collateral standard so its maintenance could be managed independently. The hope is that through use of modularity, the heart of what is identified as FORTRAN might remain small.

FUTURES

Over this period of twenty years of standardization we have been through two complete cycles and are in the midst of a third. How long does this go on and when does it end? Jean Sammet once asked me if it weren't time for the FORTRAN gurus to get together and call an end to the effort so people can get on with the using of the good languages. I have reservations over which of the current choices should be crowned the good languages. There should be something fundamentally different and better to justify dropping the huge

investment in the current languages. The replacement should have features that defy compatible inclusion in what we have.

Until this revolutionary development makes its appearance, interest in FORTRAN will remain. There is the story of the farmer who was asked by one of his eager turks why he didn't replace his old burro with one of the younger, sleeker, more highly tuned and spirited steeds. He looked at the young hand with wrinkled and wizened eyes and said, "When you have something yeh gotta be sure gets done, yeh goes with what you knows." So be it with FORTRAN.

REFERENCES

1. Heising, William P., and Richard K. Ridgeway. "FORTRAN." Proposal distributed to ASA X3.4.3, June 1962.
2. Heising, William P. "History and Summary of FORTRAN Standardization Development for the ASA." *Commun. of ACM* (Vol. 7, No. 10) October 1964, 590.
3. ASA X3.4.3. "FORTRAN vs. Basic FORTRAN." *Commun. of ACM* (Vol. 7, No. 10) October 1964, pp. 591–625.
4. ASA. American Standard FORTRAN (ASA X3.9–1966).
5. ASA. American Standard Basic FORTRAN (ASA X3.10–1966).
6. USASI. "Clarification of FORTRAN Standards—initial progress." *Commun. of ACM* (Vol. 12, No. 5) May 1969, pp. 289–294.
7. ANSI. "Clarification of FORTRAN Standards—second report." *Commun. of ACM* (Vol. 14, No. 10) October 1971, pp. 628–642.
8. Greenfield, Martin N. "FORTRAN—A History of a Pragmatic Language," HLSUA 1974 Meeting, June 11, 1974.
9. Greenfield, Martin N. "Background and Interpretation of the FORTRAN Draft Proposed Standard." The WEST COAST FORTRAN FORUM, Anaheim, California, February 9, 1976.
10. ANSI. "Draft Proposed ANS Fortran." *SIGPLAN Notices* (Vol. 11, No. 3), March 1976.
11. Brainerd, W. editor. "Fortran 77." *Commun. of ACM* (Vol. 21, No. 10), October 1978, pp. 806–820.
12. ANSI. American National Standard programming language FORTRAN, ANSI X3.9–1978.
13. ISO. Programming languages—FORTRAN. ISO 1539–1980.
14. US Department of Commerce National Bureau of Standards. FORTRAN. FIPS PUB 69. September 4, 1980.
15. CODASYL FDBLC. Fortran Data Base Facility, Journal of Development, January 1980.

DYSTAL: nonnumeric applications of FORTRAN

by JAMES M. SAKODA
Brown University
Providence, Rhode Island

ABSTRACT

This paper presents an explanation of how FORTRAN was used to write a list-processing language, DYSTAL, which uses linear arrays rather than linked word lists. Three basic features are dynamic storage allocation, integer array names as pointers, and a seven-word head for each array.

INTRODUCTION

I was in the Psychology Department of the University of Connecticut when IBM set up a computation center at MIT for use by New England colleges and universities. I attended the first summer institute offered at MIT in 1956, I believe, and struggled through the assembly language programming course. At the end of the session a young man, who I believe was Sheldon Best, got up and announced that they were working on an automatic programming system called FORTRAN. The following year when FORTRAN was made available, I attended a short course on it in Boston. As a research associate to the MIT Computation Center I began to work on statistical programs in FORTRAN, and since then it has been the only language in which I have programmed.

My encounter with nonnumeric programming came in 1963 when I attended a summer institute on the use of IPL-V[1] for simulation at the Rand Corporation. The session was organized by Bert Green. I found that IPL-V provided dynamic storage allocation, list-processing operations, such as insertion and deletion, and list-structures and procedures for handling them which could not be normally performed in FORTRAN. On the other hand, data handling was almost nonexistent, input-output was difficult, and even a simple device like a checkerboard could not be easily represented by linked-word lists. Moves on a checkerboard could not be specified by incrementing two subscripts as one could in FORTRAN, but instead lists of possible moves were utilized. Furthermore, programs written in IPL-V were reputed to be slow, and I attributed this to the linked-word list which required sequential rather than direct access to the middle of a list.

LINKED-WORD LISTS VS. LINEAR ARRAYS

Before the institute was half over I decided to write a list-processing language using FORTRAN subroutines and functions. I was not aware of Gelernter's FLPL. Joseph Weizenbaum's SLIP[2] had just been announced, and to me it appeared to be IPL-V operations written as a series of FORTRAN subprograms, with a few primitives written in machine language. I decided that in order to preserve many of FORTRAN's efficient features lists should not consist of linked words but a linear string of words. My task was to find ways of providing dynamic storage allocation at runtime, list-processing operations and creation and operation of arrays connected into tree structures. I was able to provide all of these using procedures written as FORTRAN functions. I then proceeded to add string-processing routines, sorting operations and statistical and matrix operations, aiming for a general purpose language. The first DYSTAL Manual[3] was completed in 1956. After the 1967 IFIP Working Conference on Symbol Manipulation Languages[4], I decided to make arrays relocatable, using a directory to hold the names of arrays and allowing arrays to move to a disk file as room in memory was depleted. A manual incorporating this improvement was put together in 1970[5].

My approach was that of an amateur, unaware of the niceties of computer language design, doing what appeared to be necessary to achieve features which FORTRAN did not normally provide. Much of this would not even be of historical significance, since DYSTAL was not widely used. But some of it is pertinent to the present-day effort to provide a more general-purpose language via FORTRAN. The X3J3 FORTRAN Committee is discussing setting up a core FORTRAN and extensions into different application areas. It is my belief that the core should be relatively flexible to allow for a variety of extensions. I would like to point out how I was able to make use of FORTRAN IV to accomplish unFORTRAN-like operations, while integrating numeric and nonnumeric procedures.

ESSENTIAL FEATURES

Three features were important to my effort to provide list-processing and list-structuring operations in FORTRAN. The first was dynamic storage allocation. The second was the name of an array which was separate from its content. In FORTRAN a variable, whether subscripted or note, referred to its content or value. To create tree structures or to chain arrays it was necessary to be able to use names of arrays as pointers. This called for a new data type—array name—which was different from integer and real variables. The third feature was required to provide the flexibility inherent in linked-word lists. I found this in the five-word head, which I later increased to seven words. These features were not independent of one another. I began with dynamic storage allocation, which brought into play the need to keep track of the location of an array and its features.

DYNAMIC STORAGE ALLOCATION

To implement dynamic storage allocation of linear arrays a single storage area was created and from it all arrays were allocated at runtime. To accomplish this three variables were dimensioned a maximum amount and made equivalent to one another and stored in COMMON. Later a disk file was added when arrays were made relocatable:

```
DIMENSION LOT (5000), FLOT (5000), GLOT (5000)
EQUIVALENCE (LOT, FLOT, GLOT)
COMMON GLOT
DEFINE FILE 4 (1000, 80, U, JFI)
```

The equivalencing of the three arrays made it possible to cut out any type of array from the same storage area and even to store different types of variables on the same array. The EQUIVALENCE statement therefore played a central role in providing a flexible dynamic allocation system. The use of COMMON allowed each function to have access to the entire dynamic storage area without need to enter LOT or FLOT as arguments each time. GLOT was placed in COMMON to fool the compiler into believing that LOT and FLOT in COMMON were not being modified by a FORTRAN function. This rigid requirement was encountered in Basic FORTRAN when working with the IBM 1130 computer, and I would deem that as overprotection of the user. He is better served by permissiveness to change values in COMMON as needed.

ARRAY NAME

It was the development of dynamic storage allocation that permitted and also required a name separate from the content of the array. It was necessary to keep track of the position within LOT or FLOT where the next array was to start. This location was returned and used as the name of the array. If LOCA was the name of an array, LOT (LOCA + 1) or FLOT (LOCA + 1) referred to the value of the first word of that array. Thus LOT and FLOT came to mean "the content of" a word at a given location within the dynamic storage area. In the meantime, it was possible to use LOCA as a pointer to the array and store it on other arrays, making possible chains of arrays or tree structures. Below is shown a simple tree structure with an array called NAME holding the names of three arrays, LSTA, LSTB, LSTC. These in turn hold character strings, which have been read into created arrays:

```
NAME:  LSTA, LSTB, LSTC .
LSTA:  D, O, G .
LSTB:  C, A, T .
LSTC:  H, O, R, S, E .
```

It was a great day when I realized that to create a tree structure it did not matter where the arrays were stored. All that was necessary was to be able to store array names on the same name array.

Arrays were later made relocatable and an array called MAP served as the directory.

```
LSTA = MAPL (3, 10)
```

created an array named LSTA for real numbers of length 10. The name of the array was then the location on the directory. The directory in turn held the current location of the array.

THE HEAD OF AN ARRAY

I learned the use of the attached head of an array from IPL-V. Instead of a limited amount of information, I stored the length of the array, the count of items stored on it, the mode of the array (1–7), the node to be used to store pointers in creating chains of arrays or alternatively as the row size of a matrix, an alphabetic identification, a reference count, and the directory address. The head was positioned just before the array itself so that it could be accessed by means of a zero or negative subscript. LOT (LOCA) referred to the array counter, LOT (LOCA-1) to its length and LOT (LOCA-2) to its mode—i.e. the data type stored on the array. To a considerable extent list-processing type of operations were performed with the aid of information stored in the head of an array. LOAD (WD, LSTA) could be used to store a word at the end of the line and the counter increased by one. FITEM (− 9, LSTA) was used to take off the last word on the list. If the capacity of the array was exceeded when using LOAD, the array was moved automatically to a new location and enlarged by 20 percent and the routine continued. Routines for insertion and deletion required that words be moved to make room or eliminate an empty position.

To create and operate list structures names of arrays were placed on arrays with the data type of 1 (names of arrays), which distinguished them from integer arrays with a data type of 2. This distinction was desirable in writing a routine to walk through the list structure. Each of the seven modes was associated with an input-output format so that it was possible to print out an array with the simple instruction IDUMP (LSTA) or to print out all of the arrays in dynamic storage with the instruction CALL KDUMP. Thus, when creating an array its mode and dimensions were declared numerically and retained in the head of each array. In matrix operations, such as matrix multiplication, it was not necessary to specify the row and column sizes, since these could be calculated from information in the head of the arrays involved. The head was made possible by implementation of dynamic storage allocation and by use of the EQUIVALENCE statment.

The role of EQUIVALENCE is crucial in adding the head to each array. The information in the head could be handled as integer variables using LOT. The head could be attached to any array, whether they held integer, real or literal words. In developing DYSTAL for the IBM 1130 using Basic FORTRAN, I managed to equivalence two-byte integers with four-byte real words. I did not get around to adding double-precision words as data types, but that could have been managed. The ability to equivalence different data types and the addition of a head to each array greatly contributed to relieving the programmer of many bookkeeping chores.

RECURSION

FORTRAN subroutines are not recursive—i.e. they are not allowed to call themselves. Recursive routines are desirable in symbolic manipulation of formulas and in tracing through list structures. Recursion can be simulated in DYSTAL using the approach used by IPL-V. Within a procedure dynamic storage allocation can be used to provide a pushdown stack to store intermediate information. The necessary operations can then be performed in reverse order using information in the pushdown stack. At the end of the procedure the pushdown stack can be erased. Here it is dynamic storage allocation which permits an unFORTRAN-like operation.

VIRTUAL MEMORY

Virtual memory, if it exists, is generally provided by the computer system rather than by a compiler for a particular language. For smaller machines, however, virtual memory is generally not available, and using FORTRAN to provide it greatly adds to the flexibility of writing and running large programs. The implementation of virtual memory required the setting up of a directory as an array to hold the current location of each array. This could be in memory or on a disk file. Three types of arrays were distinguished: permanent arrays, which remained in memory at the low end of the storage area, temporary arrays which were created at the upper end, and semi-permanent ones which began where the permanent ones ended. When the free space reached the end of the storage space, it was allowed to wrap around to the beginning of the semipermanent arrays. Thus it was possible to move whole arrays each time to the disk file without fragmenting the storage space. Virtual memory also neatly solved the problem of garbage collection, since it was possible to allow unwanted arrays to move to the disk file and remain there.

ACCESS TO ARRAY ELEMENTS

Creating a name of an array required adding its location to the subscript for LOT or FLOT. Making the arrays relocatable further complicated the problem of access. When an array was created its name was saved in a FORTRAN variable or placed on an array:

```
LSTA = MAPL (3, 10) .
```

To get its location, the function LOCAL was called:

```
LOCA = LOCAL (LSTA) .
```

LOCA could then be used in the subscript of LOT to access the Ith element of LSTA: LOT (LOCA + I).

Retrieval was made simpler, but not efficient, by using retrieval functions ITEM (I, LSTA) and FITEM (I, LSTA). For storage the function IPUT (X, I, LSTA) was developed. Here X is the word to be stored in the Ith position of LSTA. FORTRAN, in spite of its rule that real and dummy arguments have to agree in number, order and type, allowed me to use IPUT for storing either integers or real words. There were further complications when arrays were made relocatable, since it was necessary to insure that accessing one array, which might be on the disk file, did not kick out another one that was needed in the same part of the program. One solution was to create such arrays early and declare them to be permanent. The other was to clear sufficient free space to make sure that there was sufficient free space for the required arrays. A routine called ICHEK (LSTA, LSTB, LOCA, LOCB) brings into memory LSTA and LSTB and provides their locations. Such procedures were most helpful at the beginning of subroutines to insure that both were in memory at the same time.

My general approach was to write frequently-used subprograms as efficiently as possible by subscripting LOT and FLOT. Retrieval functions, on the other hand, were used initially to write application programs. There was discussion fairly early in the game of the desirability of a precompiler which would take the less efficient functions and replace them with direct subscripts.

STRING PROCESSING

DYSTAL's string processing operations could be applied to arrays of single characters or to words. It was hampered by the lack of literal constants, and it generally had to be assumed that character strings were read into dynamically-created arrays. It was possible to perform the basic operations of hunting for a character or a string of characters and to remove a substring or replace it with another substring. For example,

```
LOC = MASK (LSTB, LSTA, I)
```

searched for the location of LSTB within LSTA, beginning at the Ith position.

```
CALL ISWAP (LSTC, N, LSTA, LOC)
```

replaced with LSTC, the substring of N characters of LSTA beginning at LOC. Character strings stored on DYSTAL arrays had array names which could be placed on name arrays, thus making it possible to create list structures, which were needed in analyzing sentence structures. As with other data types character strings had heads, including the length of the array and the current number of characters on it. In DYSTAL single characters could be packed into a word or the word unpacked into single characters, using integer arithmetic.

FORTRAN 77 introduced the CHARACTER data type, which greatly aids string-processing in FORTRAN. The literal constant enclosed in quote marks can now be written directly into a program. But character strings can no longer be equivalenced with other data types, and hence new ways must be found to provide character strings with more flexibility, including an integer name. One method of doing this is to provide a separate dynamic storage area for character strings in a CHARACTER data type named CHAR. A function, such as LOC ('CAT', CHAR) can be used to store 'CAT' in the next available position of CHAR and return as its value the beginning and end positions within CHAR, I and J. The two numbers can be packed and stored into a single integer word:

```
LCAT = I * 1000 + J .
```

This integer value, such as 1003, can be stored on arrays whose mode specifies names representing character strings. Names of such arrays in turn can be placed on name arrays to form list structures representing, for example, sentence structures. Knowing the name of the character string, such as 1003, it is possible to retrieve the characters through the substring reference provided in FORTRAN 77: CHAR (1:3) or its equivalent value CHAR (LCAT/1000 : MOD (LCAT,1000)). The MOD function returns the remainder term needed as the designation of the end of the substring. In sorting the strings of characters into alphabetic order, it is possible to compare

character strings, but move the positions of the names of the strings rather than the strings themselves. Here again dynamic storage allocation produces a reference to the position within the area which can be treated as an integer name.

PERMANENT FILE

A more recent addition to DYSTAL has been a save file and get file instructions to save the entire dynamic storage area on the disk file at the end of a run and to recall the same storage area at the beginning of another run. All of the important words in a program, including names of arrays, can be saved from one run of a program to another by equivalencing them to a public location in the first parameter array. In my cluster analysis-factor analysis program I can first run the clusters, examine them, and if satisfied run the program a second time beginning from the point where the clustering procedure ended. It is also possible to write a program to selectively print out any of the arrays in dynamic storage. This facility provides a means of periodically updating a complex data structure constructed as a tree structure or a chain of arrays. An error made during the course of a run may result in the file not being properly stored. By saving the previous copy of a file, it is possible to go back to an earlier version.

FORM OF THE DYSTAL LANGUAGE

A language written as FORTRAN subprograms might be imagined as a series of explicit calls to subroutines. Early in the development of DYSTAL, I realized the advantage of using functions rather than subroutines. Practically every DYSTAL routine uses the name of at least one array and it was possible to allow the name of one of the arrays to be the returned value for most of them, except those retrieving values from an array. This permitted the nesting of functions within a line of the program. This gave DYSTAL a function form of specifying a series of procedures. For example, to create an array, read 10 words into it and print it out one could write:

```
LSTA = IDUMP (LRD (NRD, 1, 10, (MAPL (3, 10))) .
```

The matrix operation

$$T' (T T')^{-1} = T^{-1}$$

can be written in DYSTAL as

```
MATN = IDUMP (MPTRA (MTRAN (ICOPY (MATT)),
  MINV (MPTRA (MATT, MATT, 0)), 0))
```

MPTRA performs matrix multiplication of the first array by the transpose of the second and stores the resulting matrix in a newly created array and returns the name of this array. Although the function form is somewhat confusing because of the many parentheses, it does allow the stringing together of several routines on a single line. One can easily see that this ability is dependent upon the use of an integer name for an array. The nesting of functions makes the one-line arithmetic function quite useful. When the returned value of a function is not needed FORTRAN allows the use of the explicit CALL. For example, one can write CALL IDUMP (LSTA) even though IDUMP is a function with a returned value.

CONCLUDING REMARKS

DYSTAL used linear arrays in place of linked words and was therefore better able to take advantage of FORTRAN's desirable features—flexible input-output operations, use of subscripts, use of two-dimensional arrays, and arithmetic capabilities. The development of DYSTAL as a general purpose language encompassing nonnumerical procedures was dependent upon dynamic storage allocation, an integer name for arrays, the provision of an ample head for each created array. To develop these features there was heavy reliance on flexibilities in FORTRAN IV, especially the equivalencing of different data types. The X3J3 FORTRAN standards committee is proposing a core FORTRAN to be combined with modules in different application areas. According to its minutes, it hopes to eliminate EQUIVALENCE and COMMON from core FORTRAN. I think that this would be a serious mistake if the core is meant to serve as a basis for a series of more specialized languages. The core should remain as flexible as possible, and EQUIVALENCE and COMMON promote flexibility in an important way. Those not desiring the flexibility can always avoid the use of these features.

REFERENCES

1. Sammet, Jean E. *Programming Languages.* Englewood Cliffs, New Jersey: Prentice-Hall, 1965.
2. Weizenbaum, J. *Symmetric List Processor.* Sunnyvale, California: Computer Department, General Electric, 1963.
3. Sakoda, James M. *DYSTAL Manual.* Providence, Rhode Island: Department of Sociology and Anthropology, Brown University, 1965.
4. Bobrow, J. G., editor. *Symbol Manipulation Languages and Techniques.* Amsterdam: North Holland, 1968.
5. Sakoda, James M. *DYSTAL Manual.* Providence, Rhode Island: Department of Sociology, Brown University, 1970.

1982 NATIONAL COMPUTER CONFERENCE COMMITTEES

PROGRAM COMMITTEE

Chairman
Howard L. Morgan
The Wharton School
Philadelphia, PA

Vice-Chairman
Eric K. Clemons
The Wharton School
Philadelphia, PA

Members
Gene P. Altshuler
Peat, Marwick, Mitchell & Co.
New York, NY

O. Peter Buneman
The Moore School
Philadelphia, PA

James E. Emery
The Wharton School
Philadelphia, PA

Dennis Frailey
Texas Instruments
Austin, TX

Robert Frankston
Software Arts, Inc.
Cambridge, MA

Randall Jensen
Hughes Aircraft Co.
Los Angeles, CA

Beverly K. Kahn
Boston University
Boston, MA

Alan N. Smith
Atlantic Richfield Co.
Los Angeles, CA

Amy D. Wohl
Advanced Office Concepts
Bala Cynwyd, PA

AFIPS Liaison
Sam Lippman
AFIPS
Arlington, VA

CONFERENCE STEERING COMMITTEE

Conference Chairman
Russell K. Brown
Brown and Associates, Ltd.
Houston, TX

Program Chairman
Howard L. Morgan
The Wharton School
Philadelphia, PA

Vice-Chairman—Program
Robert R. Stirling
IBM Corporation
White Plains, NY

Film Forum Chairman
Eddie Truncellito
Schlumberger Well Services
Houston, TX

Handicapped Facilities Chairperson
Ms. Kerry A. Baer
IBM Corporation
Houston, TX

Registration Chairman
Mr. Lynn Hobson
Houston Lighting & Power
Houston, TX

NCC Liaison Program
Harvey L. Garner
The Moore School
Philadelphia, PA

Vice-Chairman, Promotion
Robert J. Gemignani
Vallen Corporation
Houston, TX

Operations Chairman
Gene Giblin
Southwest Bancshares
Houston, TX

Transportation Chairman
Bob Griffin
Houston Transit Consultants
Houston, TX

Exhibits Chairman
Dave Nelson
IBM Corporation
Houston, TX

Executive Director of AFIPS
Paul Raisig
AFIPS
Arlington, VA

Plenary Coordinator
Susan L. Rosenbaum
AT&T
New Brunswick, NJ

NCC Liaison—Operations/Promotion
W. H. Sitter
Tenneco, Inc.
Houston, TX

Society Liaison
Carey H. Snyder
Texaco, Inc.
Houston, TX

Special Activities Chairperson
Linda Vermillion
Hydril Company
Houston, TX

Fiscal Officer
Jesse B. Tutor
Arthur Anderson & Company
Houston, TX

Conference Coordinator
Fred Boecker
Tenneco, Inc.
Houston, TX

Printing Chairman
J. W. Burchfield
Moore Paper Companies, Inc.
Houston, TX

Professional Development Seminar Chairman
Joseph S. Campisi
Aetna Life and Casualty
Hartford, CT

Protocol Chairman
Albert K. Hawkes
Sargent & Lundy Engineers
Chicago, IL

National Promotion Chairman
Alex Hoffman
Consultant
Fort Worth, TX

Pioneer Day Chairman
J. A. N. Lee
Virginia Polytechnic Institute & State University
Blacksburg, VA

AFIPS Liaison
Sam Lippman
AFIPS
Arlington, VA

Implementation Plan Chairman
Bill Carlisle
Southwestern Bell Telephone Company
Houston, TX

Vice-Chairman, Operations
Bob Coker
Houston, TX

Local Promotion Chairman
Walter Ulrich
Walter E. Ulrich Consulting
Houston, TX

FILM FORUM COMMITTEE

Chairman
Eddie Truncellito
Schlumberger Well Services
Houston, TX

Members
Cheryl Culifer
DELTAK, Inc.
Houston, TX

Sal Menez
Schlumberger Well Services
Houston, TX

Stephanie Sample
DELTAK, Inc.
Houston, TX

Joe Van Hook
OXY Systems
Houston, TX

OPERATIONS COMMITTEE

Chairman
Gene Giblin
Southwest Bancshares
Houston, TX

Members
Sigman Byrd
Texas Commerce Bank
Houston, TX

Tom Houston
First City East
Houston, TX

Michael Kwiatkowski
Dresser Industries
Houston, TX

Bob Michael
First City East
Houston, TX

Craig Sherrill
Allied Bank of Texas
Houston, TX

Jerry Swan
First City East
Houston, TX

Tom Taylor
First City East
Houston, TX

Keitha Tullos
First City East
Houston, TX

Bob Voelker
First International Bank
Houston, TX

PIONEER DAY COMMITTEE

Chairman
J. A. N. Lee
Virginia Polytechnic Institute and State University
Blacksburg, VA

Members
William Aspray
Williams College
Williamstown, MA

Walter Brainerd
University of New Mexico
Los Alamos, NM

Scott Guthrie
Schlumberger Well Services
Houston, TX

Daniel Leeson
IBM Corporation
San Jose, CA

Jack Palmer
IBM Technical History Project
Yorktown Heights, NY

Steven J. Shepherd
Tenneco Oil Co.
Houston, TX

Henry Tropp
Humboldt State University
Arcata, CA

Jerrold L. Wagener
Amoco Production Research
Tulsa, OK

Thomas C. Wesselkamper
Hunter College
New York, NY

Richard L. Wexelblat
Sperry-Univac
Blue Bell, PA

PROFESSIONAL DEVELOPMENT SEMINAR COMMITTEE

Chairman
Joseph S. Campisi
Aetna Life & Casualty
Hartford, CT

Members
Richard K. Edwards
Aetna Life & Casualty
Hartford, CT

George R. Eggert
DCASR, Department of Defense
Chicago, IL

Robert J. Garabedian
Aetna Life & Casualty
Hartford, CT

Lowry McKee
Link Division, Singer
Houston, TX

Philip Palermo
Connecticut General Insurance Company
Hartford, CT

Jean M. Smith
Aetna Life & Casualty
Hartford, CT

PROMOTIONS COMMITTEE

Promotions Committee Vice-Chairman
Robert J. Gemignani
Vallen Corporation
Houston, TX

National Promotion Chairman
Alex Hoffman
Consultant
Fort Worth, TX

Local Promotion Chairman
Walter Ulrich
Walter E. Ulrich Consulting
Houston, TX

Members—Promotions Committee
Bob Griffin
Houston Transit Consultants
Houston, TX

Albert K. Hawkes
Sargent & Lundy Engineers
Chicago, IL

Carey H. Snyder
Texaco Inc.
Houston, TX

Susan Tourtellot
Houston Convention Bureau
Houston, TX

Members—National Promotions Committee
John di Targiana
Gillette Co.
Boston, MA

James V. M. Hale
Coca-Cola USA
Atlanta, GA

John Hamblen
National Bureau of Standards
Washington, DC

Phillip R. Jones
General Dynamics Corporation
Clayton, MO

Kyu Y. Lee
Seattle University
Seattle, WA

Beverly McMurrey
Consultant
Houston, TX

E. Z. Million
Million Associates
Norman, OK

Bill Rieken
Consultant
San Mateo, CA

Members—Local Promotions Committee
Bob Brejcha
Houston Natural Gas Corporation
Houston, TX

Linda Caruso
Management Systems
Houston, TX

Oscar Dugey
American National Insurance
Galveston, TX

Barbara Green
Office of the City Comptroller
Houston, TX

Connie Harris
Corporate Associates
Houston, TX

Sholeh Huber
City of Houston Health Department
Houston, TX

Mark Kellermeyer
The Cameron Group
Houston, TX

Ernie Logan
Vallen Corporation
Houston, TX

PLENARY COMMITTEE

Chairperson
Susan Rosenbaum
AT&T
Piscataway, NJ

Members
Mary Charles Blakebrough
IBM
Poughkeepsie, NY

Ocie M. Gamble
Sun Gas Co.
Dallas, TX

William A. Ritchie
AT&T
Piscataway, NJ

PROTOCOL COMMITTEE

Chairman
Albert K. Hawkes
Sargent & Lundy Engineers
Chicago, IL

Members
Glenn B. Burkhardt
Texas Instruments
Dallas, TX

Hans Puehse
Fireman's Fund Insurance Companies
San Rafael, CA

Stephen S. Yau
Northwestern University Technological Institute
Evanston, IL

SPECIAL ACTIVITIES COMMITTEE

Chairman
Linda U. Vermillion
Hydril Company
Houston, TX

Members
Claudia Bryan
Fluor Ocean Services, Inc.
Houston, TX

Lucille Franks
Lucille Franks & Associates
Houston, TX

Marsha M. Kaan
IBM Corporation
Houston, TX

Jo T. Kennedy
Fayez, Sarofim & Company
Houston, TX

Linda Swift
Houston Lighting & Power Co.
Houston, TX

Sara Walsh
University of Houston/DC
Houston, TX

NCC '82 SESSION LEADERS

Jeanne Adams
Chair, ANSI X3J3
Boulder, CO

Jeffrey S. Augenstein
University of Miami Medical School
Miami, FL

John Backus
IBM Corporation
San Francisco, CA

Roger E. Billings
Billings Computers
Independence, MO

Naomi Lee Bloom
American Management Systems, Inc.
New York, NY

Barry W. Boehm
TRW Systems, Inc.
Redondo Beach, CA

Grady Booch
Department of Computer Science
USAF Academy, CO

Alex Borgida
Rutgers University
New Brunswick, NJ

John W. Brackett
Softech Microsystems
San Diego, CA

Dave Brandin
SRI International
Menlo Park, CA

A. Winsor Brown
Volition Systems
Delmar, CA

J. C. Browne
University of Texas at Austin
Austin, TX

K. M. Chandy
University of Texas at Austin
Austin, TX

Ned Chapin
InfoSci Inc.
Menlo Park, CA

Scott Davidson
Western Electric Company
Princeton, NJ

Carl Davis
Ballistic Missile Defense Advanced
Technology Center
Huntsville, AL

Michael S. Deutsch
Hughes Aircraft Company
Los Angeles, CA

Henry Dreifus
The Wharton School
Philadelphia, PA

Martha Evens
Illinois Institute of Technology
Chicago, IL

Robert Fenchel
Xerox Corporation
El Segundo, CA

Robert E. Filman
Hewlett Packard
Palo Alto, CA

Dennis Frailey
Texas Instruments
Austin, TX

Robert C. Gammill
North Dakota University
Fargo, ND

C. F. Gibson
Index Systems Inc.
Cambridge, MA

Sakunthala Gnanamgari
Siemens Corporation
Cherry Hill, NJ

Paul Gray
Southern Methodist University
Dallas, TX

Jerrold M. Grochow
American Management Systems, Inc.
Arlington, VA

Paul Heckel
Interactive Systems Consultants
Los Altos, CA

Alex Hoffman
Consultant
Fort Worth, TX

Lance Hoffman
George Washington University
Washington, DC

Mark A. Holthouse
The Analytic Sciences Corporation
Reading, MA

Portia Isaacson
Future Computing, Inc.
Richardson, TX

Tom H. Johnson
Nolan, Norton and Co.
Lexington, MA

Michael A. Kahn
Honeywell Information Systems, Inc.
Billerica, MA

Steven Kartashev
DCA, Inc.
Lincoln, NB

Svetlana Kartashev
University of Nebraska
Lincoln, NB

Peter G. W. Keen
MIT/Sloan School of Management
Cambridge, MA

Tom Kehler
Texas Instruments
Dallas, TX

Steve E. Kolodney
Search Group Inc.
Sacramento, CA

Ken Kristie
Motorola, Inc.
Austin, TX

Dale Kutnick
The Yankee Group
Cambridge, MA

Richard C. Layer
3M
St. Paul, MN

Samuel J. Lomonaco
Institute of Defense Analysis
Alexandria, VA

Rita Gail MacAuslan
Honeywell Information Systems, Inc.
Billerica, MA

Vance Mall
Ada Joint Program Office
Arlington, VA

Fred Maryanski
Digital Equipment Corporation
Hudson, MA

Richard Mason
USC
Los Angeles, CA

Charlie McClear
Motorola, Inc.
Austin, TX

Dennis McLeod
University of Southern California
Los Angeles, CA

John McQuillan
BBN Information Management Corporation
Cambridge, MA

John F. Meyer
University of Michigan
Ann Arbor, MI

Jim Millar
Texas Instruments Inc.
Houston, TX

Don Minami
DMA Systems
Santa Barbara, CA

Howard L. Morgan
The Wharton School
Philadelphia, PA

Amihai Motro
University of Southern California
Los Angeles, CA

Christian Mueller-Schloer
Siemens Corporation
Cherry Hill, NJ

A. Napier
University of Houston
Houston, TX

Irene Nesbit
Nesbit Consulting
Princeton, NJ

Susan Nycum
Gaston, Snow and Ely Bartlett
Palo Alto, CA

J. Michael Nye
Marketing Consultants International, Inc.
Hagerstown, MD

Bob Patterson
Microprocessor Operation
Intel Corporation
Santa Clara, CA

Dave Penniman
OCLC Inc.
Dublin, OH

Jock A. Rader
Hughes Aircraft Co.
El Segundo, CA

Elizabeth D. Rather
Forth, Inc.
Hermosa Beach, CA

David C. Rine
Western Illinois University
Macomb, IL

Anne E. Robinson
SRI International
Menlo Park, CA

Patricia Seybold
The Seybold Report on Office Systems
Boston, MA

R. Shatzer
Sytek, Inc.
Sunnyvale, CA

Allen Smith
Atlantic Richfield Co.
Los Angeles, CA

Nancy Stern
Hofstra University
Hempstead, NY

Jim Swager
Honeywell Information Systems, FSD
McLean, VA

Larry Tesler
Apple Computers
Cupertino, CA

Glenn N. Thomas
Kent State University
Kent, OH

Fred Thorlin
Atari, Inc.
Sunnyvale, CA

Rein Turn
California State University
Northridge, CA

Walter Ulrich
Walter E. Ulrich Consulting
Houston, TX

Joseph E. Urban
University of Southwestern Louisiana
Lafayette, LA

David Vaskevitch
Standard Software Ltd.
Toronto, Ontario, Canada

Lynn Webber
Peat, Marwick and Mitchell
New York, NY

Evelyn S. Wilk
Arthur Anderson and Co.
Chicago, IL

James Winchester
Hughes Aircraft Company
Fullerton, CA

Amy Wohl
Advanced Office Concepts
Bala Cynwyd, PA

S. Bing Yao
University of Maryland
College Park, MD

Daniel C. Zatyko
Zatyko Associates
Santa Ana, CA

NCC '82 REFEREES

Altshuler, Gene
Ariav, Gad
Astrahan, Morton

Bartol, Ray
Beller, Aaron
Buneman, Peter

Chang, Mike
Chapin, Ned
Clemons, Eric
Couger, Dan

Dreifus, Henry

Emery, James
Evens, Martha

Frailey, Dennis
Frankston, Bob
Franta, William

Gerritsen, Rob
Ginsberg, Ralph
Gnanamgari, Sakunthala
Greenfield, Arnold

Hanks, Steve
Hsiao, David
Hoffman, Lance

Jaramillo, P.
Jensen, Randall

Kahn, Beverly
Kartashev, Steven I.
Kartashev, Svetlana P.
Kuck, D.

Levin, Dan

Miller, Kip
Morgan, Howard

Prywes, Noah

Root, David

Shneiderman, Ben
Smith, Alan
Smith, D.

Webber, Bonnie
Wohl, Amy

Yianalos, Peter

NCC '82 SPEAKERS AND PANELISTS

Alford, Mack
Allen, Cheryl C.
Allen, Dan
Annaratone, Marco
Aronofsky, Julius
Augenstein, J.

Backus, John
Bandy, Jim
Bair, James
Balzer, Bob
Barr, Avron
Bartlett, Joel
Belady, Les
Bemmu, Robert
Blank, George
Block, Dennis
Bode, Mishe
Boehm, Barry
Bowles, Kenneth
Brandin, D.
Bridge, Ed
Brinklin, Dan
Bronsema, Gloria
Brosgol, Benjamin M.
Buchanan, Bruce
Burr, Bill

Cheatham, Thomas E.
Cheng, Ray
Clippinger, Richard
Colburn, Don
Condon, Maureen
Couger, Daniel J.
Currie, Edward

Davis, Carl
Davidson, Charles
Davidson, John
Denning, Peter
Deutsch, Don
Diesem, John
Doelling, Arthur
Dowlin, Ken
Driscoll, James

Eckert, J. Presper
Estridge, William O.
Everest, Gordon

Fain, Robert L.
Farber, Dave
Fisher, Gerald
Fisher, Paul
Finin, Tim
Fox, Mark
Freed, Roy

Gaggle, Michael
Galitz, Wilbert O.
Gambino, Thomas
Gibbons, Fred M.
Girishparikh, Mr.
Goldberg, Richard
Goldstein, David K.
Goldstine, Herman H.
Gottlieb, Alan
Grabendike, K.
Gravina, Art
Greenfield, Martin
Grey, Paul
Greynolds, Elbert B.
Grochow, Jerrold M.

Hardgrave, Terry
Harper, Thomas
Harris, Kim
Harris, Larry
Harris, Richard D.
Heckel, Paul
Heimbigner, Dennis
Heising, William
Henry, Glen
Hughes, Robert
Huston, Bill

Ignizio, James P.

Januelapis, Victor
Jaworski, Joe
Johnson, Dave
Juliussen, Egil

Kaczowka, Peter
Kameny, Iris
Kane, Gerald R.
Kay, Marin
Keplinger, Mike
Kim, K.
King, John
Koskinson, Joyce
Krieger, Mark

Laffitte, David S.
Landau, Herb
Lee, Kyu Y.
Lin, Peter
Lomuto, Nico
Lowenthal, Eugene

Maples, Michael J.
Marcellino, James J.
Maresca, Gerry
Markkula, Mike
Maryanski, Fred
Mathis, Robert
Mayfield, Anne M.
McCracken, Daniel
McDonald, Walter R.
McKenney, James J.
McLeod, Dennis
McPherson, John
Meserve, Bill
Millar, Jim
Miller, Ed
Miller, Mark
Millett, Mark A.
Mills, Dick
Mills, Nancy R.
Morgan, David
Morse, John E.
Moss, Sam
Motro, Amihai
Munson, John A.
Murphy, Catherine M.
Myer, Theodore H.

Nageshwar, Srini
Nelson, Dave
Novotny, Eric
Nutt, Roy

O'Connor, Rob
Overgaard, Mark

Palmer, David F.
Parker, D.
Paul, Charles
Payne, John
Pearson, Allen L.
Peatrowsky, Ed
Peddecord, Tom
Perkins, Thomas E.
Perry, Rich
Peterson, Robert W.
Pogran, Zen
Phillips, Betty A.
Price, Lynne
Purtell, John

Quantz, Paul

Ramamoorthy, C. V.
Rattner, Justin
Ray, Clifton V.
Reggia, James
Reiser, Dick
Rolander, Tom
Rosenblatt, Bruce
Rosen, Benjamin M.
Ryan, Hugh

Sacerdotim, Earl
Sakoda, James
Sami, Maria Giovanna
Sanchez, James
Schklain, Nicholas
Scureman, M.
Shaw, Ward
Simonyi, Charles
Singh, Jitendra
Slater, Dan
Smith, Dave
Smith, John
Smith, Raoul N.
Smith, Robert
Smith, Steve
Soloway, Elliot
Spradlin, E.
Stallard, Jim
Stern, Sal
Stuewald, David C.
Sutherland, Duncan
Swanson, Burton E.

Tennant, Harry
Tesler, Larry
Thawley, Tom
Thorndyke, Perry
Tobes, Roselte
Turner, Byron

Urban, Joseph E.

Vick, Charles R.

Wagman, David S.
Wagner-Korne, Anne
Walter, Chris
Ware, W. H.
Ware, Willis
Warner, Silas
Weems, Joe
Wegner, Peter
Weingarten, Fred
Weinreb, Daniel
Weissman, Larry
Wensley, John H.
Wilk, Chuck
Wiekes, Maurice
Williams, Robert D.
Wilson, Diane
Wong, Harry
Woteki, Tom H.

Yates, Jean
Yeh, Raymond T.
Yelowitz, Larry

Zeldin, Saybean
Ziehe, Theodore W.
Zloof, Moshe

AUTHOR INDEX

Agrawal, Dharma P., 135, 239
Alexander, William, 257
Allen, F. E., 805
Amamiya, Makoto, 143
Amsler, Robert A., 657
Annaratone, M., 117

Batcher, Kenneth E., 185
Beech, David, 493
Bemer, R. W., 811
Berg, Helmut K., 3
Berra, P. Bruce, 125
Berry, R., 251
Bhuyan, Laxmi N., 135
Blackman, Maurice, 785, 793
Bloom, Naomi Lee, 539
Bowles, Kenneth L., 327
Brice, Richard, 257
Browne, J. C., 217

Callender, E. David, 381
Cardenas, Alfonso F., 341
Center, John W., 399
Chandy, K. M., 251
Cheng, Ray, 775
Choudhari, Ramesh, 501
Corbin, Jerry L., 81
Cox, David A., 555

Davidson, Edward, 639
Davis, Carl, 167
Deutsch, Michael S., 301
DeWitt, David J., 207
Dumse, Randy M., 73

Elwell, James F., 309
Estrin, Gerald, 369

Filman, Robert E., 671
Frank, G. A., 225
Franta, William R., 589
Friedland, Dina, 207
Friedman, Daniel P., 671
Fujino, Seiji, 767

Gammill, Robert, 759
Goodman, Aaron M., 359
Goyal, Ambuj, 153
Grafton, William P., 341
Greenawalt, E. M., 225
Greenfield, Martin N., 817
Grochow, Jerrold M., 389

Hardgrave, W. Terry, 571
Harslem, Eric, 515
Hartsough, Christopher, 381
Hasegawa, Ryuzo, 143
Hayes, Philip J., 469
Hicks, Anthony, 697
Hiromoto, Robert, 233
Hoffman, Lance J., 461
Honda, Masanori, 767
Hsu, Khai Li, 727
Hull, Jonathan J., 501
Huston, Bill, 19, 85
Hwang, C. Jinshong, 735

Ignizio, James P., 193
Irby, Charles, 515

Jackson, James E., 549

Kamibayashi, Noriyuki, 605
Kartashev, Steven I., 103, 167
Kartashev, Svetlana P., 103, 167
Keller, Tom W., 649
Kimball, Ralph, 515
Kohn, Leslie, 199
Koll, Matthew B., 571
Kulkarni, A. V., 225
Kurose, James F., 273

Lakshmi, M. Seetha, 649
Le Mer, Eric, 263
Levin, K. Dan, 691
Lin, Ching-Fang, 727
Lipovski, G. Jack, 153
Liu, J. W. S., 775
Liuzzi, Raymond, 125
Luo, Dawei, 617
Lybrook, C. W. 415

MacNair, Edward A.,273
Malek, Miroslaw, 153
Mark, William, 475
Maryanski, Fred, 429
Mateosian, Richard, 53
McKelvey, Terrence R., 239
McMahon, Edith M., 319
Mikami, Hirohide, 143
Minami, Don M., 11
Misra, J., 251
Mooney, James D., 529
Morris, Richard V., 381
Mueller-Schloer, Christian, 487
Murphy, Catherine M., 193

Nakamura, Osamu, 143
Nance, Richard E., 293
Neugent, William, 441
Neuse, D., 251

Okawa, Yoshikuni, 713

Palmer, David F., 193
Pathak, Janak, 53
Peatrowsky, Ed, 67
Peterson, James L., 665
Plamondon, Réjean 749
Potochnik, John R., 595

Rahimi, Said K., 589
Rao, Prakash, 3
Raymond, Janis G., 281
Rich, Elaine A., 481
Robillard, Pierre N., 749
Rosenberg, Saul, 287
Roth, Richard L., 351
Ryan, Hugh, 785, 793
Ryan, John R., 393

Sakoda, James M., 825
Salazar, Sandra B., 571
Sami, M. G., 117
Santhanam, Viswanathan, 595
Sauer, Charles H., 273
Seo, Kazuo, 605
Shapiro, Michael, 95
Shneiderman, Ben, 579
Shriver, Bruce D., 3
Smith, C. U., 217
Smith, David Canfield, 515
Srihari, Sargur N., 501
Standish, Thomas A., 333
Stockton, John F., 29

Taute, Barbara J., 409
Thomas, Glenn, 579
Thorp, Lynn, 759
Tong, Fu, 627
Turn, Rein, 449

Vaskevitch, David, 509
Vinberg, Anders, 719
Vincent, David R., 37

Wagner, Neal R., 487
Wah, Benjamin W., 697
Waldrop, James H., 363
Warner, Walter P., 293
Winchester, James W., 369

Xia, Daozhong, 617

Yada, Koji, 767
Yamamoto, Yuzo, 381
Yao, S. Bing, 617, 627

Zhou, Chaochen, 679
Zingale, Tony, 59
Zvegintzov, Nicholas, 561

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES, INC. (AFIPS)

OFFICERS

President

J. Ralph Leatherman
Hughes Tool Company
Houston, TX

Vice President

Sylvia Charp
The School District of Philadelphia
Philadelphia, PA

Treasurer

Walter A. Johnson
Consolidated Papers, Inc.
Wisconsin Rapids, WI

Secretary

Arthur C. Lumb
Procter & Gamble Company
Cincinnati, OH

Executive Director

Paul J. Raisig
AFIPS
Arlington, VA

BOARD OF DIRECTORS

AFIPS Immediate Past President

Albert S. Hoagland
IBM Corporation
San Jose, CA

American Society for Information Science (ASIS)

James N. Cretsos
Merrell Dow Pharmaceuticals, Inc.
Cincinnati, OH

American Statistical Association (ASA)

George Minich
World Bank
Washington, DC

Association for Computational Linguistics (ACL)

Donald E. Walker
SRI International
Menlo Park, CA

Association for Computing Machinery (ACM)

Peter J. Denning
Purdue University
West Lafayette, IN

Aaron Finerman
University of Michigan
Ann Arbor, MI

Raymond E. Miller
Georgia Institute of Technology
Atlanta, GA

Association for Educational Data Systems (AEDS)

John Hamblen
National Bureau of Standards
Washington, DC

Data Processing Management Association (DPMA)

P. Roger Fenwick
New York Telephone
New York, NY

Robert A. Finke
Cummins Engine Company
Columbus, IN

Robert J. Marrigan
Mail Communications, Inc.
Everett, MA

IEEE—Computer Society

Rolland B. Arndt
Sperry Univac
St. Paul, MN

Oscar N. Garcia
University of South Florida
Tampa, FL

Steven S. Yau
Northwestern University
Evanston, IL

Instrument Society of America (ISA)

Chun H. Cho
Fisher Controls Company
West Marshalltown, IA

Society for Computer Simulation (SCS)

Per Holst
The Foxboro Company
Foxboro, MA

Society for Industrial and Applied Mathematics (SIAM)

Donald K. Thomsen
SIAM Institute for Mathematics & Society
New Canaan, CT

Society for Information Display (SID)

Carlo Crocetti
Rome Air Development Center/XP
Griffis Air Force Base, NY

NATIONAL COMPUTER CONFERENCE BOARD MEMBERS

Chairman and ACM Representative

Seymour Wolfson
Wayne State University
Detroit, MI

Vice Chairman and SCS Representative

Carl Malstrom
North Carolina State University
Raleigh, NC

Small Societies Representative

George Minich
World Bank
Washington, DC

Treasurer and AFIPS Representative

Walter Johnson
Consolidated Papers, Inc.
Wisconsin Rapids, WI

AFIPS Representative

J. Ralph Leatherman
Hughes Tool Company
Houston, TX

AFIPS Representative

Sylvia Charp
The School District of Philadelphia
Philadelphia, PA

Secretary and DPMA Representative

George Eggert
Chicago DCASR
Chicago, IL

IEEE-CS Representative

Stanley Winkler
IBM Corporation
Armonk, NY

ACM President—Ex Officio

Peter J. Denning
Purdue University
West Lafayette, IN

IEEE-CS President—Ex Officio

Oscar N. Garcia
University of South Florida
Tampa, FL

DPMA President—Ex Officio

Donald E. Price
Siena College
Rocklin, CA

SCS President—Ex Officio

Stewart I. Schlesinger
The Aerospace Corporation
Los Angeles, CA

NCCC Chairman—Ex Officio

Irwin J. Sitkin
Aetna Life & Casualty
Hartford, CT

IAP Chairman—Ex Officio

Dallas Talley
Qantel Corporation
Hayward, CA

NATIONAL COMPUTER CONFERENCE COMMITTEE OF THE NCC BOARD

Chairman

Irwin J. Sitkin
Aetna Life & Casualty
Hartford, CT

Secretary

Floyd Harris
Life of Georgia
Atlanta, GA

Members

Morton M. Astrahan
IBM Research Laboratory
San Jose, CA

Harvey L. Garner
Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, PA

Albert K. Hawkes
Sargent & Lundy Engineers
Chicago, IL

Jerry Koory
Rand Corporation
Santa Monica, CA

William Sitter
Tenneco, Inc.
Houston, TX

Arnold P. Smith
IBM Corporation
White Plains, NY

Robert Spieker
AT&T Company
New Brunswick, NJ

NCC '82 Chairman

Russell K. Brown
Brown and Associates, Ltd.
Houston, TX

NCC '83 Chairman

Donald Medley
California State Polytechnic University
Pomona, CA

OAC '83 Chairman

James F. Towsen
Harrisburg, PA

NATIONAL COMPUTER CONFERENCE BOARD INDUSTRY ADVISORY PANEL

Chairman
Dallas Talley
Qantel Corporation
Hayward, CA

Members
David Bowman
Roanoke, TX

Jack Davis
Harris Corporation
Melbourne, FL

Frederick M. Hoar
Fairchild Camera and Instrument Company
Mountain View, CA

S. A. Lanzarotta
Xerox Corporation
El Segundo, CA

William Lonergan
Xerox Development Corporation
Beverly Hills, CA

Richard Mau
Sperry Rand Corporation
New York, NY

Jack McMahon
IBM Corporation
Armonk, NY

Jim Morris
DATAMATION
New York, NY

Herbert Richman
Data General Corporation
Westboro, MA

Gordon Smith
Memorex Corporation
Santa Clara, CA

AFIPS HEADQUARTERS STAFF

OFFICE OF EXECUTIVE DIRECTOR

Executive Director
Paul J. Raisig

Executive Secretary
Joan Tackett

Public Information Secretary
Marion Baskin

FINANCE AND ADMINISTRATION

Director, Finance and Administration
Janis Miller

Accountant
Saryratha Thach

Bookkeeper
Carrol Reid

Administrative Manager
Mary A. Dix

Administrative Coordinator
Ken Fields

Analyst
Ramsey Harris

Secretary/Receptionist
Terry DiMurro

AFIPS PRESS

AFIPS Press Director
Christopher N. Hoelzel

Fulfillment Administrator
Olive Shilland

Secretary
Sharon Lee Conway

NCC Copy/Production Editor
Elizabeth G. Emanuel

CONFERENCE DEPARTMENT

Director of Conferences
James H. Kroell

Administrative Assistant
Sue Robinson

Manager, Conference Operations
Sam Lippman

Conference Coordinator
Margaret Dyer

Conference Secretary
Wendy Chin

Manager, Exhibit Operations
Larry Jennings

Exhibit Operations Secretary
Jill Newman

Exhibit Sales Manager,
Luellen Hoffman

Exhibit Sales Secretary
Dennis Smoot

Marketing Manager
Betty Lou Cooke

Marketing Coordinator
Debbie Kalbfleisch

Marketing Secretary
Lori Keller

COMMUNICATIONS DEPARTMENT

Director of Communications
John Gilbert

Research Associate
Ellen Law

Secretary
Patty Mayo