Massachusetts Institute of Technology

Project MAC Progress Report VII July 1969 to July 1970


```
                 3
            SIN  (X)  - SIN (X)
(C1)        --------------------
                     2
                  COS  (X)

                            2            2
(C2)    SUBSTITUTE (1, SIN  (X) + COS  (X), C1)

(D2)                    - SIN (X)
-------------------------------------------------------------
           1/3     1/3 3           1/3    1/3
(C3)    (4     + 2    )  - 6 (4     + 2    ) - 6

(C4)                   SIMPLIFY

(D4)                      0
-------------------------------------------------------------
                                      2
(C5)            POWERSERIES (SIN  (X), X)

           INF    J
          ====  ====                      J  2 J + 2
          \     \                     (-1)  X
(D5)       >     >     ----------------------------------
          /     /      (2 K + 1)! (2 J - 2 K + 1)!
          ====  ====
          J = 0 K = 0
-------------------------------------------------------------
                          INF
                         /       M - 1
                         [    M X
(C6)            LIMIT    I   --------- DX
                M -> 0   ]     2
                         /    X  + 1
                          0

(C7)                  EVALUATE

(D7)                      1
```

The work reported here was carried out within Project MAC, an M.I.T. research laboratory. Support was provided by:

The Advanced Research Projects Agency of the Department of Defense, under Office of Naval Research Contracts Nonr-4102(01), -(02), and Defense Supply Service Contract DAHC15 69 C 0347;

The National Aeronautics and Space Administration, under Contracts NGR 22-009-393 and NAS 12-2093;

The National Science Foundation, under Contracts GJ-432 and GJ-1049;

The National Library of Medicine, under Contract PH-43-68-1249.

The support for some of this work came from the M.I.T. Departments and laboratories that participate in Project MAC and whose research programs are, in turn, sponsored by Government and private agencies.

Our cover illustrates four brief dialogues with MACSYMA, a computer system for algebraic manipulation under development at Project MAC since 1968. The lines labeled C1 through C7 are displays of lines typed in by a user, and translated to two-dimensional format on a typewriter-like device. The lines labeled D2 through D7 are computed responses to commands. The examples demonstrate some recent improvements to MACSYMA which include the ability to evaluate limits, improper integrals, and power series expansions.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE
Project MAC Progress Report VII July 1969 to July 1970

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Annual Progress

5. AUTHOR(S) *(Last name, first name, initial)*
Collection of reports from Project MAC participants
Prof. J. C. R. Licklider, Director

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
1 July 1970	156	(In Text)

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
Nonr-4102(01), -(02)	MAC Progress Report VII
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	--

10. AVAILABILITY/LIMITATION NOTICES
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT
The broad goal of Project MAC is experimental investigation of new ways in which on-line use of computers can aid people in their individual work, whether research, engineering design, management or education.

This is the seventh annual Progress Report summarizing the research carried out under the sponsorship of Project MAC. Details of this research may be found in the publications listed at the end of each section and in Appendix A.

14. KEY WORDS

On-Line Computers	Time-Sharing	Dynamic Modeling
Multiple-Access Computers	Information Systems	Programming Linguistics
Real-Time Computers	Artificial Intelligence	Computation Structures
Computer Networks	Machine-Aided Cognition	Theory of Automata
Interactive Management	Graphics	Hybrid Circuits

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

PROGRESS REPORT VII

JULY 1969 to JULY 1970

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

545 Technology Square Cambridge, Massachusetts 02139

PROJECT MAC

JULY 1969 to JULY 1970

D. S. Ackerman
W. Ackerman
A. C. Adams
S. M. Adams
G. K. Adler
H. Adler
E. Albanese
W. Allen
C. W. Andrews
G. S. Andrews
R. Ascott
H. Ashley
A. Bagchi
B. J. Bailin
M. E. Baker
R. Banks
D. Barma
A. Bardou
W. Bass
W. F. Bauer
R. C. Bean
R. Beatty
M. D. Beeler
P. Bennett
V. M. Berardinelli
R. M. Berman
W. T. Beyer
A. K. Bhushan
R. J. Bigelow
W. D. Bilofsky
T. W. Binford
G. R. S. Bingham
P. B. Bishop
E. H. Black
M. Bogue
J. W. Brackett
P. Brandler
P. G. W. Bras
M. F. Brescia
R. D. Bressler
D. Bricklin
H. R. Brodie
M. Bromberg
S. A. Brooks
M. S. Broos
A. L. Brown
C. Brown
E. Brown
K. M. Brown
H. P. Bruck
G. G. Bruere-Dawson
R. Bryan
D. E. Burmaster
M. C. Burnham
R. Bushkoff
T. F. Callahan
A. Campbell
R. H. Campbell
I. R. Campbell-Grant
D. Capps
O. D. Carey
K. J. Carley
T. Carlton
S. Carney
R. Carpenter
M. Carr
L. S. Cavallaro
C. Chang
J. R. Cecil
E. Charniak
J. C. Cheney
M. T. Cheney
L. M. Chui
D. D. Clark
J. Coffman
J. Cohan
M. A. Cohen
N. J. Colvin
M. J. Connell
L. J. Connelly
Prof. F. J. Corbató
R. Crowley
S. E. Cutler
R. C. Daley
D. T. Dalton
C. A. Dancey

B. K. Daniels
T. M. Dattilo
A. J. Davidoff
D. R. Dawson
P. E. deCoriolis
H. M. Deitel
Prof. J. B. Dennis
F. L. DeRemer
J. DeRosnay
Prof. M. L. Dertouzos
J. D. DeTreville
M. W. Dickens
J. Dionne
S. Dimond
Prof. J. J. Donovan
P. Dorin
C. P. Doyle
H. R. Drab, Jr.
M. S. Draper
F. Drenckham
D. Duggento
S. D. Dunten
R. S. Eanes
D. E. Eastlake
J. W. Eddleman
M. Edelberg
R. M. Elkin
J. R. Ellerman
Prof. A. Evans, Jr.
J. R. Evans
S. Fahlman
Prof. R. M. Fano
R. J. Fateman
R. J. Feiertag
S. Feldman
H. Fell
Prof. R. R. Fenichel
J. G. Fiascanaro
Prof. M. J. Fischer
R. Fish
T. Fitzpatrick
R. J. Fleischer
L. T. Flynn
J. L. Fowler
E. Fox
P. J. Fox
R. Frankston
W. A. Freeman
J. S. Freiberg
E. Freuder
J. D. Fulton
J. A. Friel
M. G. Furze
P. Gagner
E. A. Gardner
R. L. Gardner
C. C. Garman
S. Garner
S. L. Geffner
J. L. Gertz
J. P. Golden
R. E. Golden, III
R. S. Goldhor
I. Goldstein
R. C. Goldstein
Prof. G. A. Gorry, Jr.
R. W. Gosper
C. D. Graceffa
Prof. R. M. Graham
M. J. Grano
P. A. Green
R. S. Green
R. D. Greenblatt
J. M. Greene
I. G. Greif
J.-Y. Gresser
A. K. Griffith
J. M. Grochow
F. Guertin
R. H. Gumpertz
Prof. A. Guzman
M. Hack
M. K. Hadley
D. Hall
M. M. Hammer
A. Hanover
P. Hardin
R. J. Harman
J. H. Harris
Prof. M. Harrison
B. Harvey
J. F. Haverty

P. G. Hebalkar
D. A. Henderson, Jr.
W. H. Henneman
Prof. F. C. Hennie
A. Herskovits
C. E. Hewitt
D. C. Hill
R. F. Hill
M. G. Hinchey
P. Hirshon
R. Hoeft
J. T. Holloway
P. Holloway
B. K. P. Horn
E. C. Horvath
B. D. Hubbard
P. W. Hughett
W. F. Hui
W. G. Hutchinson, Jr.
K. M. Jacobs
L. H. Jamieson
J. L. Jaroslav
J. P. Jarvis, III
P. Jensen
J. W. Johnson
L. Johnson
R. Johnston
D. L. Jones
Prof. M. M. Jones
T. L. Jones
E. I. Kampits
R. K. Kanodia
E. I. Katz
R. M. Katz
D. Kaufman
C. A. Kessel
M. M. Kessler
D. J. Kfoury
C. Kidwell
P. A. King
E. S. Klang
P. R. Klein
J. C. Klensin
W. J. Klos, Jr.
T. F. Knight
Prof. Z. Kohavi
E. I. Kohn
D. Kontrimus
L. J. Krakauer
J. Kulick
M. Lambrinides
D. Lang
P. Leach
P. D. Lebling
R. J. Lecompte
P. M. Ledoux, Jr.
B. P. Lester
C. Leung
Prof. J. C. R. Licklider
J. C. Lind
R. Lindsay
L. Lipman
Prof. C. L. Liu
P. Loewe
S. Lothes
Prof. F. L. Luconi
C. Lynn
R. F. Mabee
J. Macko
S. E. Madnick
P. S. Malek
R. Mandl
M. J. Marcus
K. J. Martin
Prof. W. A. Martin
W. D. Mathews
T. McLaughlin
J. J. McGillivray
D. McMillan
R. P. McNamara
R. M. Metcalfe
C. R. Mehta
Prof. A. Meyer
E. W. Meyer, Jr.
P. L. Miller
R. G. Mills
J. M. Milner
Prof. M. L. Minsky
G. H. H. Mitchell
R. M. Moll
P. Monaco
S. Montgomery

E. T. Moore
R. C. Moore
B. A. Morneault
N. I. Morris
S. L. Morton
Prof. J. Moses
S. R. Murphy
D. Murthy
S. K. R. Murthy
E. Nangle
S. B. Nelson
Prof. D. N. Ness
J. R. Nestor
R. E. Neubauer
W. Y. Ng
S. E. Niles
R. Noftsker
G. E. Noseworthy
J. Nourse
C. Obler
B. Ong
R. Orban
R. C. Owens, Jr.
M. A. Padlipsky
M. A. Pagliarulo
P. A. Pangaro
L. G. Pantalone
Prof. S. A. Papert
Prof. M. S. Paterson
S. S. Patil
D. Peaselee
J. T. Pepe
R. Petrivalle
P. Pichoir
J. L. Piggins
J. E. Pinella
L. K. Platzman
W. Plummer
C. Ramchandani
K. Reagan
E. T. Reardon
D. P. Reed
C. L. Reeve
L. I. Reich
J. L. Reuss
H. Richards
R. Roach
R. B. Roberts
E. M. Roderick
G. T. Roe
J. S. Roe
B. Rosenbaum
L. J. Rotenberg
J. Rothnie
A. Rubin
J. B. Rubin
Prof. J. H. Saltzer
P. R. Samson
S. Saunders
D. C. Scanlon
R. R. Schell
A. Scherer
M. D. Schroeder
R. C. Schroeppel
J. Schwartz
K. Schwartz
Prof. M. S. Scott-Morton
J. I. Seiferas
A. Sekino
H. L. Selesnick
L. Seligman
L. I. Selwyn
T. H. Seymore
J. M. Shah
G. Sharp
H. J. Siegel
D. Silver
K. K. Simpson
L. B. S. Sloan
T. P. Skinner
N. J. Smith
S. W. Smoliar
J. Snell
C. Solomon
M. V. Solomita
J. W. Spall
M. Speciner
M. J. Spier
W. A. Spies
J. Stavrinos
M. K. Stephens
J. Stern

R. T. Stetson
J. R. Stinger
N. Stone
S. M. Stoney
J. M. Strayhorn
A. J. Strnad
G. J. Sussman
J. E. Sussman
J. E. Tamayo
C. D. Tavares
M. L. Terry
A. Testa
R. H. Thomas
M. R. Thompson
W. H. Thrasher
R. C. Thurber, Jr.
C. Tillman
H-M. D. Toong
L. E. Travis
E. Trautman
E. Tsiang
H. E. Tucker
D. H. Vanderbilt
T. H. VanVleck
J. L. Vecchione
K. D. Venezia
A. Vezza
B. J. Vilfan
C. A. Vogt
M. Von Sawyer
V. L. Voydock
J. W. Waclawski
R. W. Wade
C. T. Waldrop
W. C. Walker
G. Wallace
D. L. Waltz
M. Wand
P. S-H. Wang
J. E. Ward
P. Ward
S. A. Ward
M. B. Weaver
M. W. Webber
S. H. Webber
N. S. Weinstein
J. Weiss
Prof. J. Weizenbaum
T. A. Welch
D. M. Wells
J. C. Wentzell
J. L. White
T. Williams
L. Wilson
T. A. Winograd
P. H. Winston
C. H. Woebcke
E. M. Wolman
R. T. Wong
F. H. G. Wright, II
L. F. Yeager
C. Ying
J. C. Yochelson
F. L. Yost
K. Young
M. L. Young
B. J. Zak
S. N. Zilles

Guests

N. Adleman
T. G. Evans
Prof. E. Fredkin
Prof. G. Iazeolla
Prof. E. I. Organick
Prof. C. Stratchey

CONTENTS*

* The work of the Artificial Intelligence Group for 1969-1970 will be reported in Project MAC Progress Report VIII.

PROJECT MAC ADMINISTRATION

Prof. J. C. R. Licklider	Director
Prof. M. M. Jones	Assistant Director
D. E. Burmaster	Assistant Director for Student Activities and Business Manager
J. R. Ellerman	Assistant Business Manager (to February 1970)
P. Brandler	Assistant Business Manager
D. C. Scanlon	Assistant to the Director and Office Manager
R. J. Harman	Assistant to the Director
M. S. Draper	Administrative Assistant
L. J. Connelly	Property Officer (to May 1970)
M. K. Hadley	Librarian

S. A. Brooks
L. S. Cavallaro
M. J. Connell
S. Dimond
D. Duggento
R. E. Golden, III
C. D. Graceffa
J. M. Greene
D. Kontrimus
E. T. Moore
L. G. Pantalone
E. M. Roderick
K. K. Simpson
J. Stavrinos
M. K. Stephens
R. T. Stetson
E. M. Wolman
M. L. Young

INTRODUCTION

In Project MAC ("Men and Computers"), about 270 persons are engaged in digital computer research and development; they include faculty members -- mainly of the Departments of Electrical Engineering and Mathematics and of the Sloan School of Management -- staff members, and students.

The over-all program of Project MAC comprises the programs of 11 interacting and overlapping groups. The work of five of these will be summarized here in order to describe the Project MAC effort in 1969-1970.

Artificial Intelligence

The last year has seen significant advances in analysis of visual scenes and visually controlled manipulation of objects by computer, in machine understanding of natural language and narrative, and in a broad effort to incorporate knowledge and intelligence into programs. In these areas, which we group under the rubric "Artificial Intelligence", Professors Robert R. Fenichel, Michael J. Fischer, Marvin L. Minsky, Seymour A. Papert, Michael S. Patterson, Joseph Weizenbaum, and Patrick H. Winston and Visiting Professor Edward Fredkin have conducted research with approximately 60 staff members and students.

Into a new programming language and system, PLANNER, Carl E. Hewitt has incorporated an array of features that promise to be as basic to heuristic programming as have been the "DO Loops" of FORTRAN and the "FOR Statements" of ALGOL to numerical programming. In PLANNER one can write, for example, "Whenever X happens, do Y", where X is a general description of an event and Y is almost any action at all. For example, one can tell PLANNER to choose a simpler goal whenever three efforts to reach the old goal fail -- and PLANNER will set up a process ("demon") that keeps an eye open for trios of failures and, whenever it sees one, initiates the reselection process.

Terry A. Winograd completed a system of programs that translates a wide range of statements from English into the PLANNER language. Winograd's system is based on a heuristic grammar that uses contextual information; his system handles the semantic and syntactic parts of the analysis concurrently. An important feature, which gives the system more flexibility than is afforded either by "semantic networks" or by lists of grammatical rules, is the representation of the grammar as a set of programs. The definition of a word is also a program -- as, indeed, is each component of the system's "knowledge of the world". All such programs are available to the deductive part of the system.

The interests of the Artificial Intelligence Group embrace human as well as machine intelligence. The last year pressed home the essential

pertinence to human teaching and learning of basic concepts developed or clarified through research on artificial intelligence, and Professors Minsky and Papert and some of their colleagues determined to exploit the breakthrough into the realm of human cognition. In April 1970, Professor Papert gave a Saturday lecture on this subject to a capacity audience at M.I.T., and he and Professor Minsky participated in a discussion with visiting leaders in the field. In June, the National Science Foundation provided initial funds for research in teaching and learning.

Computer-Based Mathematics Laboratory

Continuing the development of "Mathlab", a system of computer programs designed to provide sophisticated assistance to people working on mathematical problems that involve complex symbolic expressions, Professors William A. Martin and Joel Moses implemented a new algebraic manipulation system. With this new system, Mathlab is able to give strong assistance in work with summations, integrals, derivatives, exponentials, logarithms and factorials. If, for example, at point C14 in a certain calculation the user types to Mathlab

FACTOR (X**6 - 1)

and then presses the @ key to tell Mathlab to go, Mathlab at once displays

(D14) $(X + 1)(X - 1)(X^2 + X + 1)(X^2 - X + 1)$

If at another point an expression stands as

(D20) $$\frac{X^2+X-6}{X^3+6X^2+9X}$$

the user can have it "rationally simplified" by typing

RATSIMP (%)@

where % means "it" or "the preceding expression". Mathlab then responds with

(D21) $$\frac{X-2}{X^2+3X}$$

When given $(e^{2x} + 2e^x + 1) - 2\log(e^x + 1)$, Mathlab simplifies it to zero.

Computation Structures

Human mathematicians are, of course, superior to Mathlab in intuition, in deciding what manipulations to try in order to reach a goal. However, a suitably programmed computer can handle, much more rapidly and accurately than any human mathematician, algebraic manipulations involving dozens or hundreds of terms. Thus, the human and computer capabilities complement each other. Even though the development of

Mathlab is far from complete, it proved itself, this past year, to be a very helpful assistant in serious mathematical work. During the coming year, its capabilities will be further increased.

Computation Structures

In research on "computation structures", a group of 13 staff members and students led by Professor Jack B. Dennis worked toward a formal integration of hardware and software concepts, especially of concepts pertaining to highly parallel, asynchronous computer systems. Their work dealt with design, architecture, specification and modeling of digital systems, with representation of concurrent processes, and with security, privacy, and controlled sharing of procedures and data.

One of the tools that most facilitates thinking about complex concurrent processes is a diagram called the Petri net. Suhas S. Patil generalized Petri nets, as modified by Holt, to handle coordination of asynchronous events and has showed that Petri nets can be systematically converted into asynchronous modular structures. In turn, Professor Dennis showed that Patil's generalized nets are suitable for representing the control of very large computers. Asynchronous design of a machine, similar in many ways to the synchronous CDC 6600 but simpler in detail, required only nine types of control module. For such a machine, asynchronous design has important advantages in conceptual simplicity and perhaps also in speed.

Attempting to understand a large and complex digital system, a person examines it one part at a time and then, actually or conceptually, puts the parts and their behaviors together. Suppose that each part turns out to be determinate in the sense that all runs of any program (that will run in it) yield the same result. Is the over-all system necessarily determinate? This last year, Patil showed that it is, given an appropriate input-output discipline, which he defined. He showed that a class of Petri nets called "marked graphs" has the determinacy-preserving property.

Prakash Hebalkar carried out a study of restrictions of concurrent activities that are imposed by limitation of resources -- an ubiquitous problem (encountered in transportation, manufacturing, maintenance, etc.) that is of great interest in the field of computation. With the aid of a very useful representation called "demand graphs", he developed a fundamental understanding of the phenomenon of deadlock, in which would-be concurrent processes block one another by hoarding resources, and of the safeness algorithm used in efforts to anticipate and avoid deadlock.

Other research carried out by the Computation Structures Group includes an analysis of hierarchical associative memories, the development of schemata ("computational schemata") for modeling the structure

of computer programs, and the beginning of the definition of a very basic and general programming language, intermediate between such a language as ALGOL and the "language" of the code that is directly executable by computer hardware. In June at Woods Hole, Massachusetts, the Computation Structures Group held a conference, attended by 27 research workers from more than a dozen laboratories, on "Concurrent Systems and Parallel Computation".

Computer System Research

Under the leadership of Professor Fernando J. Corbató, Professor Jerome H. Saltzer and Robert C. Daley and in close cooperation with a group in the General Electric Company headed by Charles T. Clingen, the Computer System Research Group of Project MAC brought the Multiplexed Information and Computing Service (Multics) System -- the advanced and comprehensive time-sharing system on which Project MAC has focused a large part of its total effort since 1965 -- into successful operation. On 1 October 1969, Project MAC transferred operational control of the Multics System to the M.I.T. Information Processing Center under an arrangement that leaves Project MAC in charge of continued development of the operating system and of research on computer-utility and computer-network aspects of Multics.

Although Multics is a much more complex and sophisticated system than its predecessor, the Compatible Time Sharing System, which was the first large general-purpose multi-access computer system, Multics was able in Fall 1969 to support as many users as CTSS; and it has been increasing steadily in number of simultaneous users and in ratio of performance to cost ever since it reached its initial operating capability. The number of registered users of Multics has increased quite linearly from 26 projects and 190 individuals in October to 72 projects and 408 individuals in June, and it now seems quite probable that Multics will meet the initial design expectations, which seemed radical when they were published in 1965, in respect of performance and use.

In retrospect, it appears that one of the best decisions of the Multics project was to program the operating system in a high-level programming language. That decision represented a break with the tradition of system programming in "assembler language". Using a high-level language made it possible to revise the program repeatedly, some parts as many as seven times, and to make progress despite "usually high" turnover in the staff. These two factors far outweighed the advantage (perhaps a factor of two, over-all) that could have been achieved through the more efficient coding possible in assembler language -- and, in any event, that advantage remains open, to be exploited, if it should seem worthwhile, when no further fundamental revisions of the operating system are envisaged.

Over the long development period, there were times when it seemed that the main objective of the Multics project was simply to complete Multics. During the last year, however, it was possible to devote time and energy to the earlier-conceived and more-basic purpose: to understand how to systematize and optimize the myriad factors and forces that interact with one another in a comprehensive multi-access information and computing system. Marked progress was made toward that goal. It was possible to make sense out of about a dozen technical puzzles. Each gain in understanding reflected itself at once in improved system performance and, at the same time, added a significant element to the body of knowledge of computer system design.

Toward the end of the year, some of the interests and energies of the Computer System Research Group turned to problems of graphical display and to Multics as a node in a multi-computer network. Those topics will figure strongly in research during the coming year.

Programming Linguistics

Professors Robert M. Graham, Arthur Evans, Jr., and John J. Donovan, Visiting Professor Michael A. Harrison, and a group of 38 staff members and students conducted research in the linguistics of computer programs. Much of this research is aimed at understanding programming languages in terms of formalisms similar to those of logic and mathematics. Because computer programming languages are simpler, have more definite purposes, are more likely to be deliberately designed, and are more susceptible to measurement and analysis than natural languages, there is some chance of understanding them formally, in due course, and dealing with them as quasi-mathematical objects rather than (as is now approximately the case) as cooking recipes or instructions for assembling hi-fi kits. The practical advantages to be gained through formal mastery of the language of computers are very great. If it were possible, for example, to state precisely what a computer program is intended to do and then formally -- through a definite sequence of operations similar to those used in proving theorems -- to show that it does or does not do it, then one of the main sources of trouble in the use of computers could be eliminated. As matters stand now, about all one can do to test a program is to check in a few specific (and usually oversimple) cases that, step-by-step, it performs the operations its programmer specified and, at the end, yields output considered correct on the basis of external criteria. That procedure is so obviously unsatisfactory as to provide strong motivation for more formal "theorem-proving" approaches, almost no matter how difficult they appear to be. At the same time, it is evident that work in formal programming linguistics is intellectually attractive and self-motivating.

Professor Donovan and his associates developed a mathematico-linguistic formalism called "Canonic Systems" within which one can specify the

syntaxes of computer languages and the rules for translating from one computer language (e.g., a compiler language) to another (e.g., an assembler language). They were successful, though as yet only in a simplified case, in preparing programs capable, given the syntaxes and the rules, of carrying out the translation automatically; and they made progress toward specifying the complexity of the translation process for various language pairs in terms of the number of steps theoretically required.

Professor Evans and his associates studied methods of formalization that appear promising from the points of view of language description and language extension. They found several ways to improve the definition of programming languages, which currently are described in manuals full of rather jargonistic natural language plus syntactic "rewrite rules". They also found several ways to let the user of an "extensible" programming language specify extensions that, for some special purpose, he would like to make to its general-purpose base language. As tools in the study of formalization, the group used the languages PAL and BCPL. It brought the formalization and documentation of PAL, which was designed especially for pedagogical purposes, near to completion, and it improved the performance and expanded the library of BCPL in Multics, produced a computer-based version of the BCPL Reference Manual, and "exported" tapes of BCPL to 16 System 360 installations.

Other Research Programs

The five programs touched upon in the foregoing paragraphs subsume about two-thirds of the research program of Project MAC. It will have to suffice merely to mention the rest in this summary.

Professors Frederick C. Hennie, C. L. Liu, and Albert R. Meyer and nine associates continued research in the theory of automata, advancing the understanding of the complexity of computations and the structure of automata. They proved two new theorems about complexity, clarified the concept of randomness as applied to particular sequences, extended findings of Minsky and Papert to additional varieties of perceptron, and obtained new results in graph theory, algebraic coding theory, integer programming, and extensible languages.

Professors Malcolm M. Jones, G. Anthony Gorry, and Michael S. Scott-Morton conducted research in management application of computers. With Professors Daniel Roos and James D. Bruce, Dr. Myer M. Kessler, and a distributed group of about 20 staff members and students, Professor Jones conducted a program of studies on interactive problem-solving and decision-making and continued the development of the simulation system SIMPLE; and he and Robert Goldstein carried on the development of the Advanced Information Management System, MacAIMS.

Professor Michael L. Dertouzos and several associates in the Electronic Systems Laboratory, functioning as a research group of Project MAC, conducted studies of an essentially new kind of computer, a computer made of components that are in one respect digital and another analog. It seems possible that such a computer can solve certain classes of problems more rapidly than ordinary digital computers and more accurately than ordinary analog computers.

Professor Robert M. Fano has long been concerned about the possible and actual impacts upon society and, especially, with the question of how to make computers serve individuals (as distinguished from organizations). Since he retired from the Directorship of Project MAC two years ago, he and several students have studied impact-related issues intensively. Professor Fano's article, "Computers in Human Society -- for Good or Ill?" in the Technology Review of March 1970, summarizes some of their thinking.

The Dynamic Modeling Group, formed at the beginning of the year to develop techniques and an interactive computer system to facilitate the formulation and testing of ideas in terms of computer-program models, acquired as a foundation for its system a Digital Equipment Corporation PDP-6/10 computer and the very sophisticated and responsive time-sharing software developed since 1965 by members of the Artificial Intelligence Group. By the end of the year, the most essential subsystems of the dynamic modeling system were operating, and a major part of the effort was shifting from "basic system programming" to the development of the programs with which users of the system will directly interact.

In the areas of Computer Networks and Computer Graphics, the past year's efforts were mainly groundwork. The Interface Message Processor that will connect Multics and one or both of the PDP-6/10 computer systems to a coast-to-coast network of research computers was installed, and an advanced display subsystem was incorporated into the dynamic modeling computer system. At the end of the year, the network and graphics programs were shifting into high gear.

Student Participation

During the past year, the number of undergraduate student members of Project MAC increased from approximately 25 to 76. This increase was due partly to a deliberate effort, championed by David Burmaster, Assistant Director for Student Activities, and partly to the successful initiation of M.I.T.'s Undergraduate Research Opportunities Program, under the direction of Dr. Margaret MacVicar. The number of graduate student members of Project MAC increased, during the year, from 25 to 40.

INTRODUCTION

Administration

In September, Miss R. Joyce Harman joined Project MAC as Assistant to the Director. During the year, Miss Harman greatly improved the operation of the Document Room and Publications Office.

Financial Support

During the past year, the core program of Project MAC and the Artificial Intelligence Group were supported, as heretofore, by the Information Processing Techniques Directorate of the Advanced Research Projects Agency (ARPA). Individual projects were funded by several other agencies: research in visual perception and in extensible languages, National Aeronautics and Space Administration; interactive problem-solving and decision-making, Office of Naval Research; library-information networks, Lister Hill National Center for Biomedical Communication of the National Library of Medicine; dynamic modeling, Behavioral Sciences Directorate of ARPA; programming generality, National Science Foundation.

COMPUTATION STRUCTURES

Prof. J. B. Dennis

I. R. Campbell-Grant
R. Carpenter
H. M. Deitel
Prof. R. M. Fano
P. J. Fox
J. L. Gertz
I. G. Greif
M. Hack
P. G. Hebalkar
Prof. F. L. Luconi
M. J. Marcus
B. Morneault
S. S. Patil
L. J. Rotenberg
L. Seligman
D. H. Vanderbilt
W. C. Walker

COMPUTATION STRUCTURES

I. INTRODUCTION

Research in the Computation Structures Group has the objective of advancing knowledge and understanding of computer system organization through abstraction and analysis. Our activities have led us to some interesting ideas regarding appropriate directions for the evolution of general-purpose computer hardware. Much of our current activity explores the implications of these ideas concerning computer system organization. Areas under study include: the theory and practice of asynchronous systems; concurrency in computation -- its influence on computer structure and on the representation of algorithms; the concept of "programming generality" -- the property of a computer system that would permit unrestricted combination of independently written programs; the controlled access to programs and data bases; and an approach to formal semantics for programs based on an abstract model for information structures.

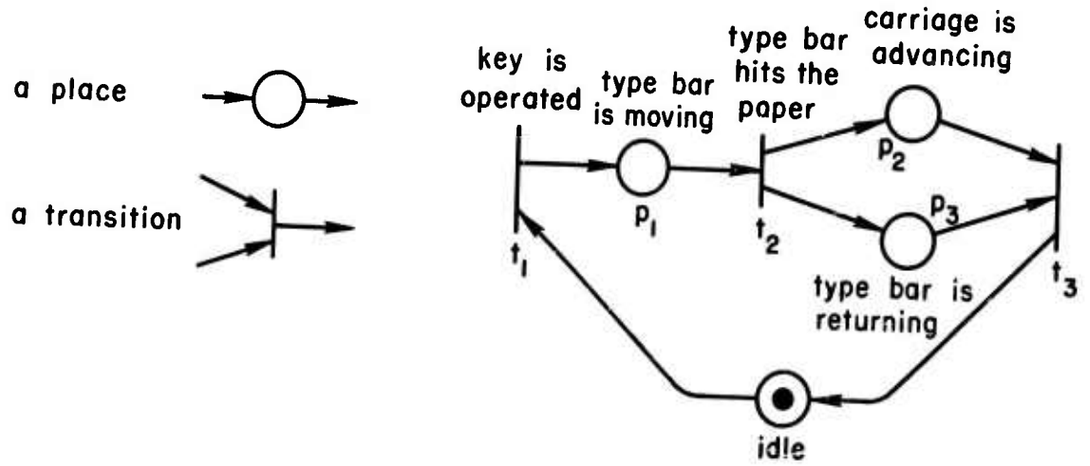
The past year has seen major advances in our understanding of modular asynchronous systems and the intimate relation of modular control structures to the Petri nets studied by Anatol Holt. We have found our knowledge of asynchronous systems sufficient to yield elegant and readily understood implementations of the control mechanisms of complex central processors. We have analyzed aspects of the concept of a hierarchical associative memory. Our understanding of the properties of uninterpreted schemes of programs has been improved through study of graphs that explicitly show data dependence. Finally, we have studied formal models of two aspects of advanced operating systems -- the controlled sharing of information, and the avoidance of deadlocks arising from resource sharing.

II. MODULAR ASYNCHRONOUS SYSTEMS

By "system" we mean an arrangement of parts that interact with one another by means of discrete signals. The essence of systems is activity: The parts of a system act at instants in consequence of earlier actions by other parts of the system. Most systems have many parts that act without immediate intercommunication. Such independent parts that may act simultaneously are said to have concurrent activity. Man-machine interaction involves concurrent activity of the man and the computer; a digital system operates through the concurrent activity of its individual circuits. The importance of concurrency goes far beyond the use of parallel actions to attain greater speed. A large system is usually constructed through interconnection of simpler systems which often operate without central control. The component systems must interact to make their presence felt and this interaction is inherently a concurrent activity. We shall review some aspects of our current

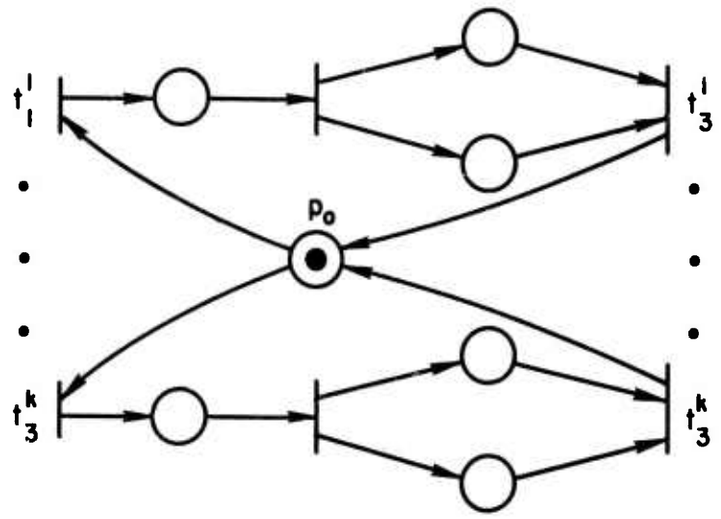
work on the representation of concurrent activity and the implementation of systems in the form of asynchronous modular hardware structures.

Consider what happens when a typewriter key is pressed. The type bar is initially idle. When the key is operated, the type bar starts moving toward the carriage; when it hits the paper, it starts to retreat and at the same time the carriage starts to advance. The key can be operated again only after the mechanism has returned to the idle condition, that is, the motion of the carriage has stopped and the type bar has returned to rest. This activity may be represented by a diagram called a Petri net:



The circles are called places and the solid bars are called transitions. The places are associated with conditions and the transitions with events. The condition associated with a place is said to hold when there is a token (sometimes called a marker or stone) at the place. A transition is enabled if all its input places have tokens. An enabled transition may fire, taking one token from each of its input places and putting a token on each of its output places. In terms of events and conditions, one can thus say an event occurs only when all conditions required for its occurrence hold, and the occurrence of an event causes the old conditions to cease and new conditions to hold. The activity continues as transitions fire, changing the distribution of tokens in the net and enabling other transitions.

In Petri nets, a place may be an input place of more than one transition. The situation where two transitions are active, but have one input place in common, is called a conflict because the transitions are in conflict over the token at the shared place. Only one of the transitions may fire even though both are active. Such a situation arises in a typewriter which does not permit more than one key to be operated at the same time. The Petri net below illustrates this situation.



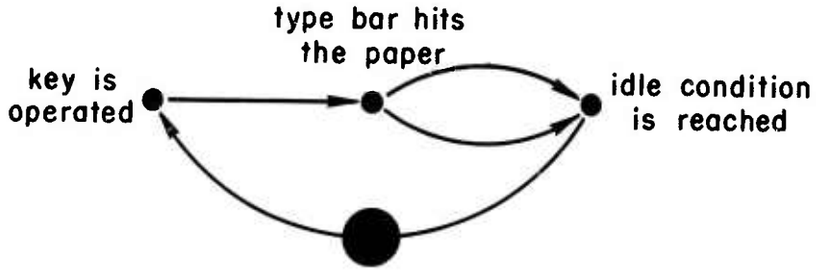
In this figure, transitions $t_1, \ldots, t_k$ are in conflict over the place p_o, and the conflict at this place prevents two or more keys from being operated concurrently.

Petri nets are a scheme for representing concurrent systems adopted by Anatol Holt of Applied Data Research [1] from the nets originally proposed by Carl Adam Petri of the University of Bonn [2]. In the Computation Structures Group, Suhas Patil has developed a generalization of Petri nets that simplifies the representation of interactions associated with resource sharing [3]; Jack Dennis has investigated the use of Petri nets to represent the control structures of a highly parallel computer processing unit [4]; and we have studied the implementation of nets in the form of asynchronous modular structures. A few aspects of these investigations are discussed briefly in the following paragraphs.

Marked graphs constitute a subclass of Petri nets in which each place is an input place of exactly one transition and an output place of exactly one transition. The net describing the operation of one key of a typewriter is a marked graph. Marked graphs have many important properties, and there is a direct correspondence between marked graphs and elementary control structures for digital systems built by the interconnection of a set of primitive asynchronous control modules to be introduced shortly. This correspondence is useful in two ways: A computer control unit specified as a marked graph can be translated into an asynchronous control structure by a clerical procedure; and a control structure may be converted into a marked graph to facilitate analysis.

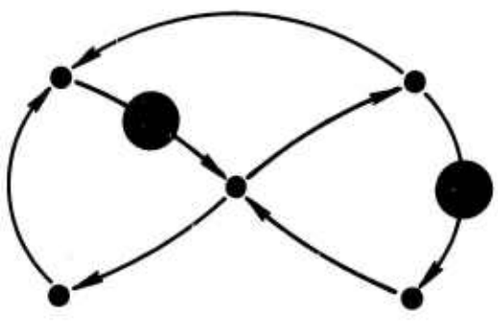
Since a place in a marked graph has only one incident arc and only one emergent arc, the circles representing the places are usually omitted -- an arc from one transition to another is understood to have

a place on it. Further conciseness is obtained by drawing the transitions as solid dots. In this simplified form, the marked graph describing the operation of a typewriter becomes:



In the new notation, the presence of tokens is indicated by placing markers on the arcs -- hence the name "marked graphs".

An important question about a marked graph is whether its activity continues forever or comes to a halt. The property of representing activity that goes on indefinitely is called liveness. A net is said to be live for some initial marking if, after any arbitrary activity has passed, a continuation of activity is possible that will fire any chosen transition. In other words, in a live net no transition is ever crossed off the list of transitions that may be called upon to fire. In general, it is difficult to determine whether an arbitrary Petri net is live. Yet marked graphs have the nice property that a marked graph is live if and only if cutting the marked edges of the graph leaves an acyclic graph. The marked graph shown below is live.

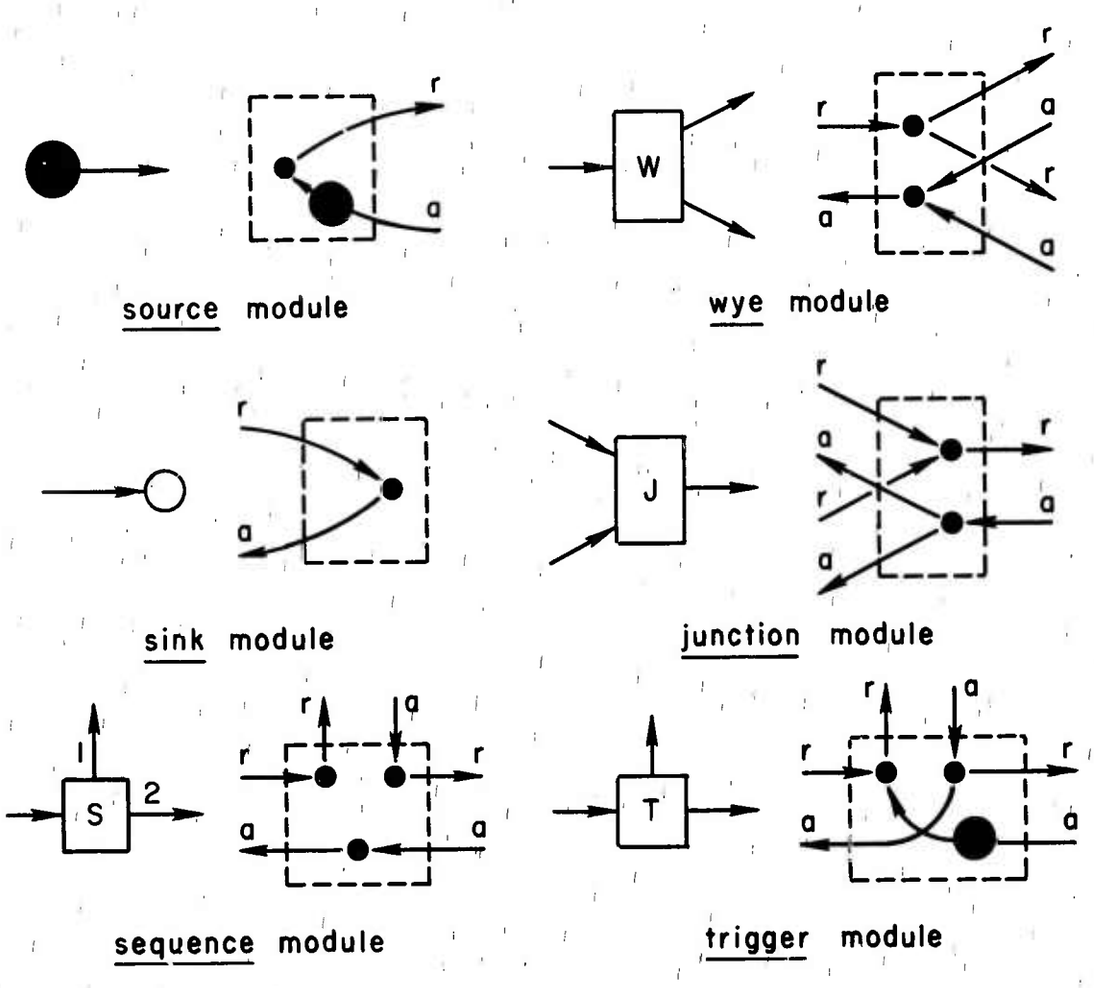


a live marked graph

The reader can check that, if any of the markers are removed, the activity of the graph will come to a halt. This property of marked graphs is very useful in determining whether an elementary control structure is free of hang-ups.

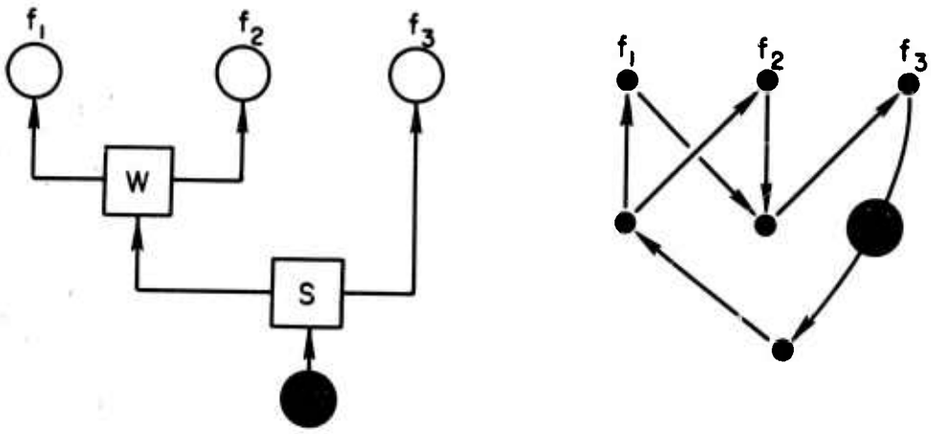
An elementary control structure is a digital system consisting of models of six types interconnected by directed links. Each link is able to transmit ready signals in the forward direction and acknowledge signals in the reverse direction. By associating two arcs with each

link of a module, the behavior of each module type may be specified by a marked graph fragment as follows:



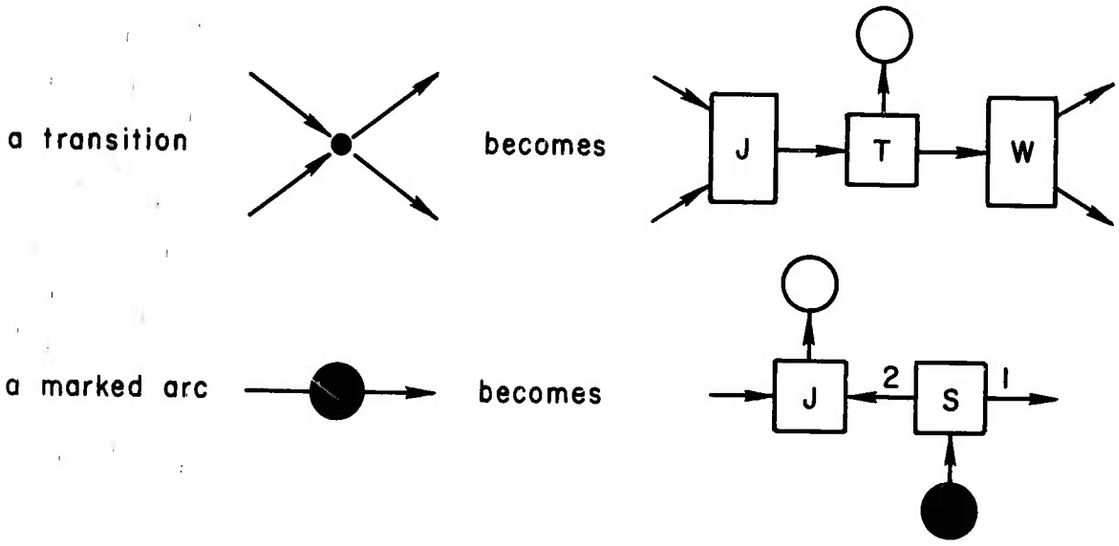
The arrival of a token on an arc in the marked graph corresponds to the transmission of a ready or acknowledge signal between two modules. A wye module, for example, sends a ready signal over the two emergent links when a ready signal is received on the incident link. Then, when acknowledge signals have been returned, an acknowledge signal is returned over the incident link.

Thus a wye module controls the concurrent execution of two independent operations. The sequence module controls the sequential execution of two operations. The junction module permits an action to take place only when the conjunction of two conditions becomes true. The control structure shown on the next page causes concurrent execution of activities f_1 and f_2, and causes activity f_3 to occur only when f_1 and f_2 have completed. The operators f_1, f_2, and f_3 are represented by sink modules, and a source module is included so that the control structure will have unceasing activity. The corresponding marked graph was found by substituting for control modules the marked graph fragments



given above, and simplifying the resulting graph by omitting certain redundant nodes. Since the marked graph is live, we can conclude that the control structure from which it was derived will not hang up.

It is also straightforward to obtain an elementary control structure that implements an arbitrary marked graph by making the following substitutions:



The resulting control structure is guaranteed to be hang-up free if the given marked graph is live.

Work is continuing on the problem of obtaining control structures for more general subclasses of Petri nets. We know, from the work of Suhas Patil [3], a systematic way of implementing any Petri net by an interconnection of asynchronous modules. However, this scheme seems unnecessarily complex, and we are studying what sets of simple primitive modules are sufficient to implement several intermediate classes of nets.

III. DESCRIPTION OF A HIGH PERFORMANCE PROCESSOR

We have looked into the suitability of Petri nets and asynchronous control structures for representing and implementing the control mechanisms of a high-performance processor. For this exercise, we chose a machine similar in principle to the Control Data 6600 but simpler in detail. The machine has several functional units that can perform different operations concurrently. The processor is so organized that instructions may be executed in a sequence different from their order of appearance in the instruction stream. A mechanism known as the scoreboard controls access of the functional units to values held in data registers so that each unit operates only when its operands are available.

Synchronous logic design techniques were used for the 6600. Thus it appeared to be an interesting challenge to see whether the control mechanisms of such a machine could be conveniently implemented by using the asynchronous modular techniques developed by the Computation Structures Group.

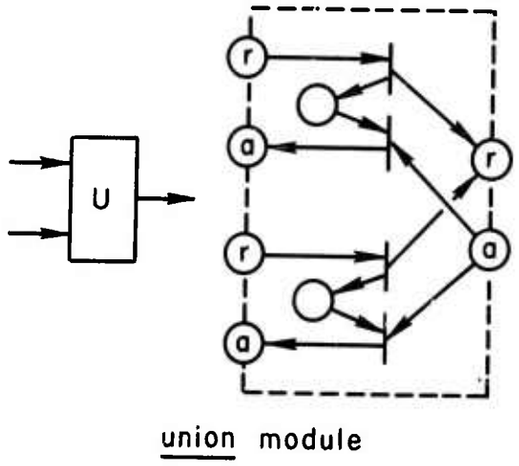
We divided the control problem into these parts: the instruction queue, the instruction allocator, the scoreboard, and control circuits for the functional units. Each was represented by a Petri net, and a control structure was devised to have exactly the behavior represented by each Petri net. It turned out that nine types of control modules were sufficient to give reasonable implementations of all six control structures:

source	wye	decision
sink	junction	union
sequence	trigger	arbiter

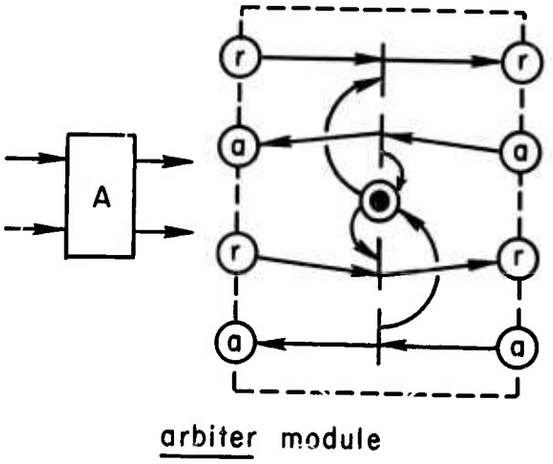
The first six of these modules were specified earlier in terms of marked graphs. The three remaining modules are defined by the fragments of Petri nets shown on the following page.

The union module permits control of an activity from either of two points within a control structure. The arbiter interlocks two activities so that only one of them may be in progress at a time. The decider module makes it possible for the control structure to effect different activities, depending on information residing outside the control structure -- for instance, the operation code of an instruction.

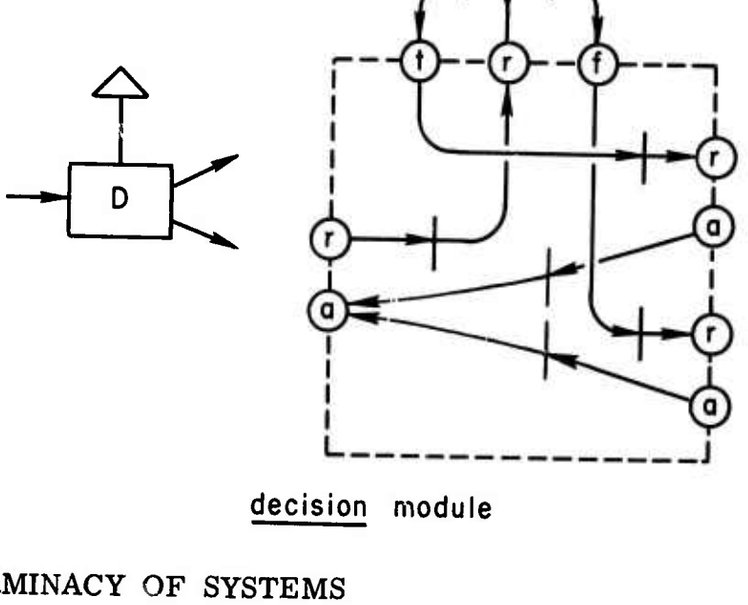
The design of the scoreboard turned out to be particularly elegant, and it seems clearly preferable to a synchronous design in regard to complexity and speed. Details are given in a recent paper by Jack Dennis [4].



union module



arbiter module



decision module

IV. DETERMINACY OF SYSTEMS

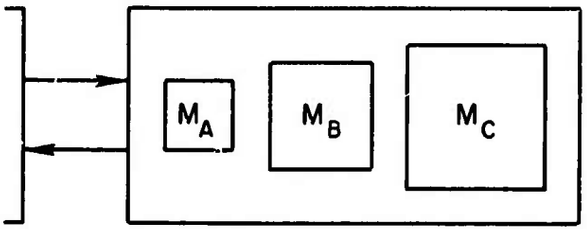
To keep the design complexity of a large system within manageable limits, the system is generally conceived as a combination of simpler systems. Unfortunately, even if the subsystems are known to work correctly, one cannot conclude that the interconnection of the subsystems to form the complete system will operate as intended. Therefore, it is important to obtain a better understanding of the problems which arise when systems are interconnected. In this direction we have achieved some important results concerning interconnections of determinate systems -- systems whose input-output relations are functions. A computer system which gives the same results for two runs of a given program for given data is a determinate system, a system that does not is not determinate. In constructing a large system from simpler determinate systems one would like to know how to ensure that the interconnection will result in a determinate system. Suhas Patil [5] has shown that, if the intercommunication discipline is chosen properly, any interconnection of a number of determinate systems may be

guaranteed to be determinate. This work provides a theoretical basis for elementary control structures: The elementary control structures form a class which is closed under interconnection. Moreover, since each of the elementary control modules discussed earlier is determinate, each member of the class of elementary control structures is guaranteed to be a determinate system. Correspondingly, the marked graphs form a class of determinate systems.

This work on the interconnection of systems may have significant application to networks of computers in which one would like to ensure correctness of a computation even though parts of it are carried out at different installations.

V. HIERARCHICAL ASSOCIATIVE MEMORY

The use of location-independent addressing is essential in a computer system that offers programming generality. In contemporary computer systems, where the memory consists of several physical storage media (solid-state, magnetic-core, drum, etc.), combinations of software and hardware mechanisms (paging, for example) have generally been used to realize location-independent addressing. Nevertheless, it is recognized that these implementations suffer from gross inefficiencies in the form of wasted processor time and poorly utilized memory space. In 1968, we outlined a radical concept of computer organization, and proposed the concept of a hierarchical associative memory [6].



processor hierarchical associative memory

In such a memory system each level is arranged as an associative memory with value fields of n bits and key fields of p bits; M_A is small and fast, M_C is slow by comparison but large. Reference to an item is made by presenting its name to the memory system. A match is first sought in M_A; if successful, the required item has been located and is read out or altered. If the search in M_A is unsuccessful, the key is used for a search of M_B, and then a search of M_C. When an item is found, it is moved to the highest level M_A, possibly together with other items known likely to be required in conjunction with it. In each level, we suggested that the age of items since their last instance of use be used to determine which items should be moved down in the hierarchy to maintain a suitable number of vacant locations for newly referenced items.

As in conventional memory systems, an organization is desired that permits a large throughput (average number of references completed per unit time). In contemporary high-performance systems, high throughput is achieved by building the memory in several modules each of which can be performing memory accesses concurrently with the others. In a location-addressed memory, this scheme works well because each name (address) always designates the same location in the same module, and action by more than one module is never required to complete a reference.

The construction of a modular associative memory poses some new problems. Since an item may occupy different locations in the memory at different times, one does not know in general which module will contain an item when access to it is required. Unless some provision is made for organized assignment of items to modules, an access request to a modular associative memory must be presented to each of the modules either in sequence or concurrently. If this is done sequentially, an average of half the modules will have to be interrogated before the item is found. If the item is not present in this level of the memory hierarchy, all modules must be interrogated before this fact is known. If all modules are interrogated concurrently, each one will be activated whether or not the item is present in the level, but the average time required to complete an access may be less. In either scheme, the speed advantage of using a modular memory is lost.

Jeffrey Gertz has investigated two alternate schemes for avoiding the necessity of searching all modules [7]. Both schemes assign each item to a specific module according to some readily tested property of the item:

(1) By ownership -- all items belonging to the same computation are assigned to the same module.

(2) By transformation -- a transformation of the key (a hash code) determines the module to which an item is assigned.

If items are assigned to memory modules by ownership, a search of more than one module is required only when reference is made to the information owned by another computation. If the key includes unique identification of an item's owner, only one module need be searched. If the key does not indicate ownership, the module containing owned information can be interrogated first -- on the assumption that items referenced are more likely to be owned items than shared items. The use of this scheme implies there are many more active computations than modules because it is unreasonable to expect one module to exactly fit the memory requirement of any one computation.

The assignment of items to modules according to a hash code of their keys is attractive where one expects most information to be shared among active computations. Only one interrogation is required to locate an item or to find that it must be retrieved from a lower level.

However, if an item may be referenced by two distinct keys, either the item would have to be duplicated in two modules, or all modules would have to be interrogated to effect reference by one of the two keys.

VI. BASE LANGUAGE RESEARCH

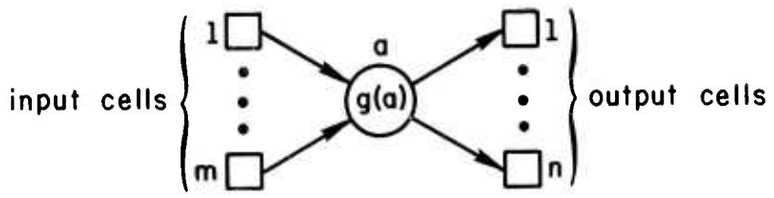
During the past year, work has begun toward the definition of a base program language. We think of the base language as a representation scheme for programs intermediate between source programming languages such as Algol and Snobol, and a machine-level representation. In its design, we hope to achieve three goals: to create a general-purpose language that is entirely consistent with the requirements of programming generality; to find a representation that expresses all possibilities for concurrent execution of parts of algorithms; and to obtain a language that can be used as a functional specification for an advanced highly parallel computer design.

We have made major gains in our understanding of the properties of certain mathematical models of the structure of programs; we call these models computation schemata. Our theoretical work with computation schemata has so far been restricted to computations that operate on simple variables -- variables whose structure as a collection of simpler entities is not relevant to the scheme (the flowchart) of the computation. Yet it is important to thoroughly understand this subject as a basis for building a more general theory for programs that operate on structured data.

VII. COMPUTATION SCHEMATA

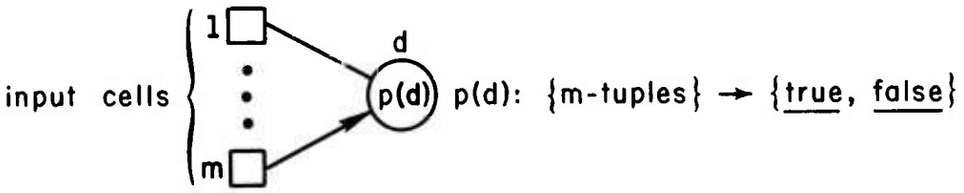
Our work on computation schemata has evolved from the thesis research done by Van Horn [8], Rodriguez [9], Luconi [10], and Slutz [11] at Project MAC, and has been considerably influenced by the original studies of Yanov [12] and, more recently, the work of Karp and Miller [13], and the work of Paterson [14]. Two questions have been of greatest interest to us: What sort of constraints must be met in the representation of parallel computations so that unique results of computations may be guaranteed? Under what conditions is it possible to determine whether two representations (schemata) describe identical classes of computations? For the class of schemata we have considered, we now have satisfactory answers to the first question, and have gained a better understanding of the second.

A computation schema represents the manner in which functional elements and decision elements are interconnected, and their action sequenced, to define an algorithm. The functional elements of a schema are called operators: Each operator a evaluates some unspecified function of an m-tuple of input variables and assigns values to an n-tuple of output variables.



g(a): {m-tuples} → {n-tuples}

The unspecified function associated with an operator a is denoted by g(a). The decision elements of a schema are called deciders: Each decider d tests some unspecified predicate p(d) for an m-tuple of input variables.

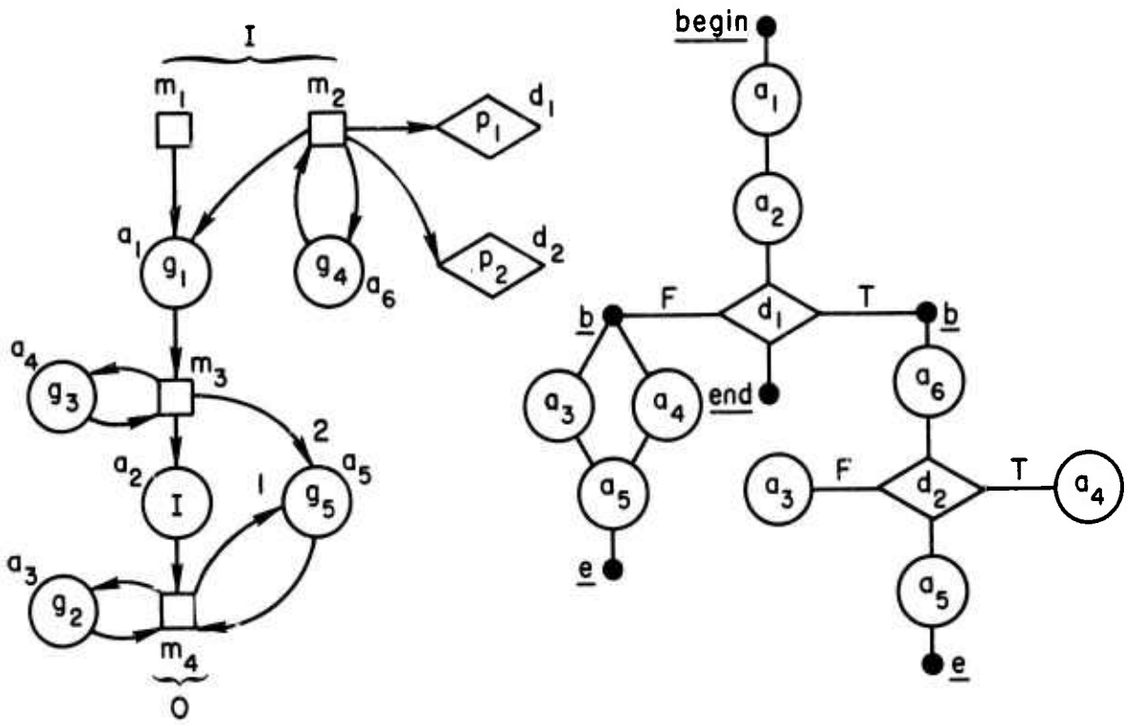


A computation schema has two parts -- a data flow graph and a control. The data flow graph defines the interconnections through which results of each operator application are passed on as arguments for further transformations and tests. The variables of a schema are represented in the data flow graph by boxes called cells. There is also a circle for each operator and a diamond for each decider. Directed arcs join the operators to their output cells and represent the connections to each operator and decider from its input cells.

The cells of the schema are identified by the letters m_1, m_2, ... Certain cells are designated as input or output cells. Values are assigned to the input cells before a computation begins; upon completion, the result is the set of values present in the output cells. Several operators, a and b, say, may have the same associated function letter: g(a) = g(b). In this way, a schema may require that two operators, a and b, always implement the same transformation, although the particular transformation is unspecified. Similarly, each decider designates a predicate letter p(d). The function letters and predicate letters of a schema make up two finite sets G and P.

The control of a computation schema is a specification of the order in which the operators and deciders of the data flow graph are permitted to act. In particular, the control indicates how further progress of computations is affected by the actions of the deciders. For the examples of computation schemata given below, we shall represent the control by precedence graphs. An example of a computation schema is the following.

S_1: data flow graph precedence graph



Each diamond node in the precedence graph connects to two subordinate precedence graphs that specify alternative computations according to whether the designated decider has a true or false outcome. Operator a_2 in the data flow graph is an identity operator; the associated function $g(a_2)$ is always the identity function.

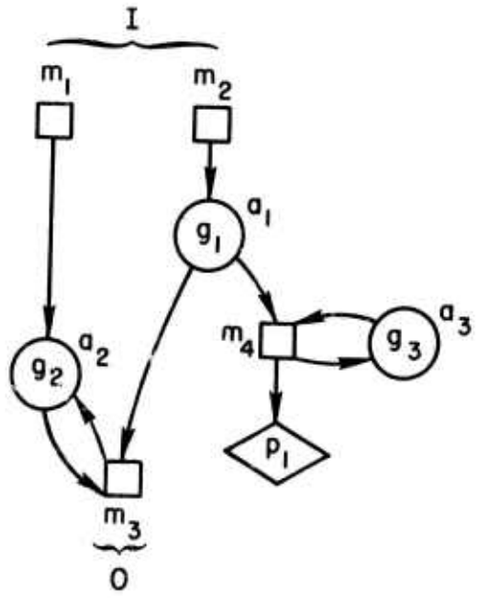
For schema S_1, the precedence graph allows just four distinct sequences of action by the operators and deciders of the schema. These sequences comprise the control set C of the schema

$$C_1: \left\{ \begin{array}{l} (a_1\ a_2\ f_1\ a_3\ a_4\ a_5) \\ (a_1\ a_2\ f_1\ a_4\ a_3\ a_5) \\ (a_1\ a_2\ t_1\ a_6\ f_2\ a_3\ a_5) \\ (a_1\ a_2\ t_1\ a_6\ t_2\ a_4\ a_5) \end{array} \right\}$$

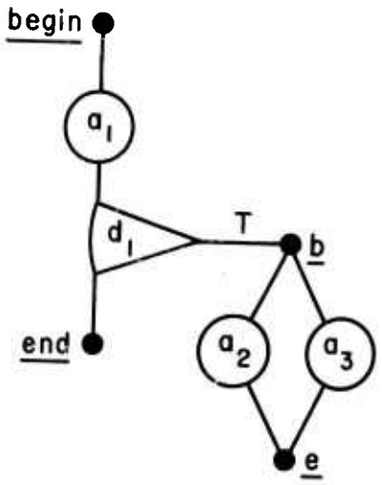
In these sequences, a_1 stands for an action by operator a_1; f_1 stands for an action by decider d_1 for which the outcome is false; and t_1 stands for an action by decider d_1 for which the outcome is true. Since no iteration is present, the control set C_1 is finite.

Iteration is represented in a precedence graph by a pie-shaped node connected to a single subordinate precedence graph.

S_2: data flow graph



precedence graph



The computation specified by the subgraph is repeated until the decider acts with a <u>false</u> outcome. The control set for schema S_2 is C_2.

$$C_2: \left\{ \begin{array}{l} (a_1\ f_1) \\ (a_1\ t_1\ a_2\ a_3\ f_1) \\ (a_1\ t_1\ a_3\ a_2\ f_1) \\ (a_1\ t_1\ a_2\ a_3\ t_1\ a_2\ a_3\ f_1) \\ \quad \cdot \qquad \cdot \qquad \cdot \\ \quad \cdot \qquad \cdot \qquad \cdot \\ \quad \cdot \qquad \cdot \qquad \cdot \end{array} \right\}$$

To convert a computation schema into a specification of a particular algorithm, it is necessary to specify the functions and predicates designated by the function letters in G and the predicate letters in P. Of course, the specified functions and predicates must have domains and ranges consistent with the topology of the data flow graph, and must be in agreement whenever the value of a function may be the argument of a function or predicate. Such a specification of functions and predicates is called (after Yanov) an <u>interpretation</u> of a schema.

Two properties of schemata are of particular interest to us. A schema S is <u>determinate</u> if, for any interpretation of the function and predicate letters, S determines a functional relation of output tuples to input tuples. In order to say whether two schemata S_1 and S_2 describe the

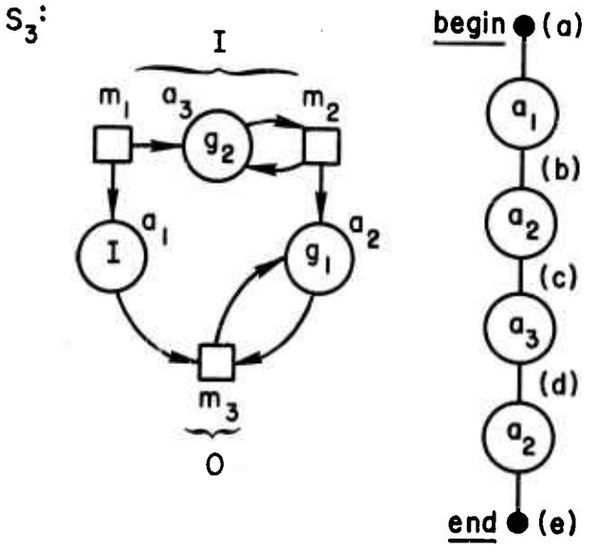
same computations, we must be able to relate the interpretations of the function and predicate letters in S_1 and S_2. For this purpose, let

$$G = G_1 \cup G_2 \qquad P = P_1 \cup P_2 \quad .$$

Then S_1 and S_2 are <u>equivalent</u> schemata if, for any interpretation of the functions and predicate letters in G and P, S_1 and S_2 determine precisely the same relation of output tuples to input tuples.

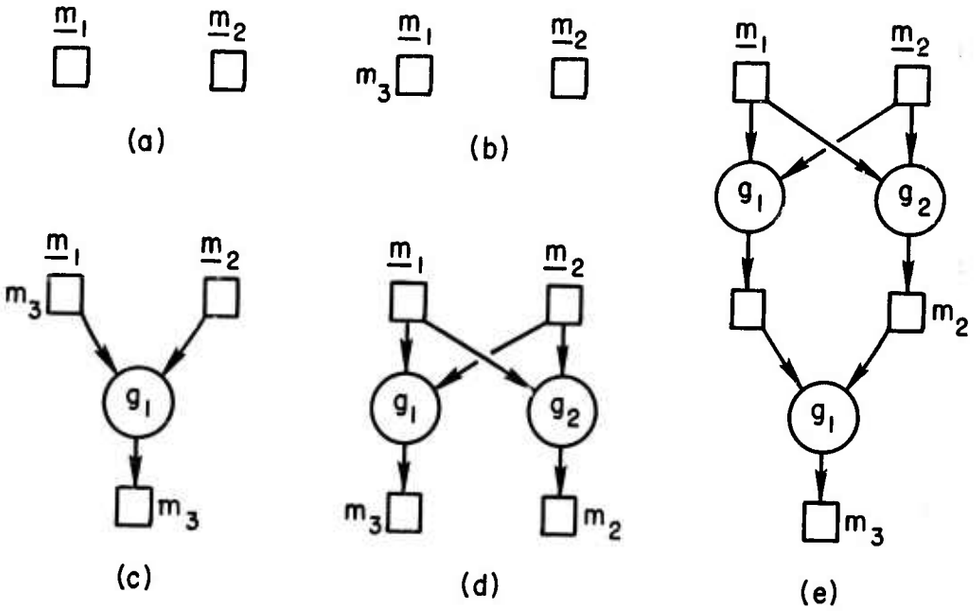
To develop insight into the questions of determinism and equivalence, we have devised the notion of <u>data-dependence graph</u> or <u>dadep graph</u> for short. A dadep graph of a schema sets forth separately each action by an operator or decider. For a particular control sequence of a schema, the final value placed in each output cell will be the result of some cascaded composition of functions. A dadep graph is just a graph representation of the cascade composition of operators associated with each output cell.

Let us construct the dadep graph for schema S_3 from its unique control sequence $\alpha = (a_1\ a_2\ a_3\ a_4)$.

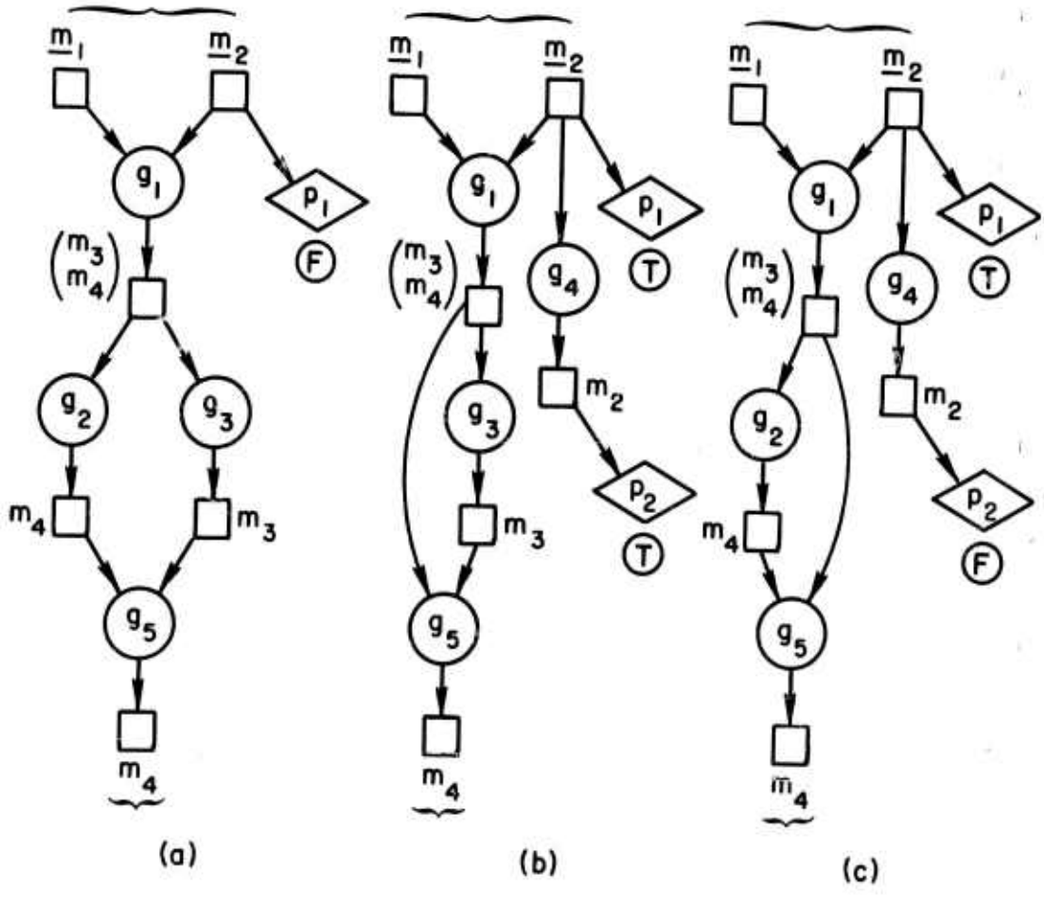


The construction is shown on the next page. We start by setting down a copy of each input cell of the schema. (The letters denoting these cells are underlined.) Then we add a copy of an operator and its output cells for each succeeding element of the control sequence. Each cell added to the dadep graph is labeled as in the data flow graph, and this label is erased from the cell copy previously bearing it. In the case of an identity operator, a second label is given to the most-recent copy of its input cell, and no copy of the operator is made.

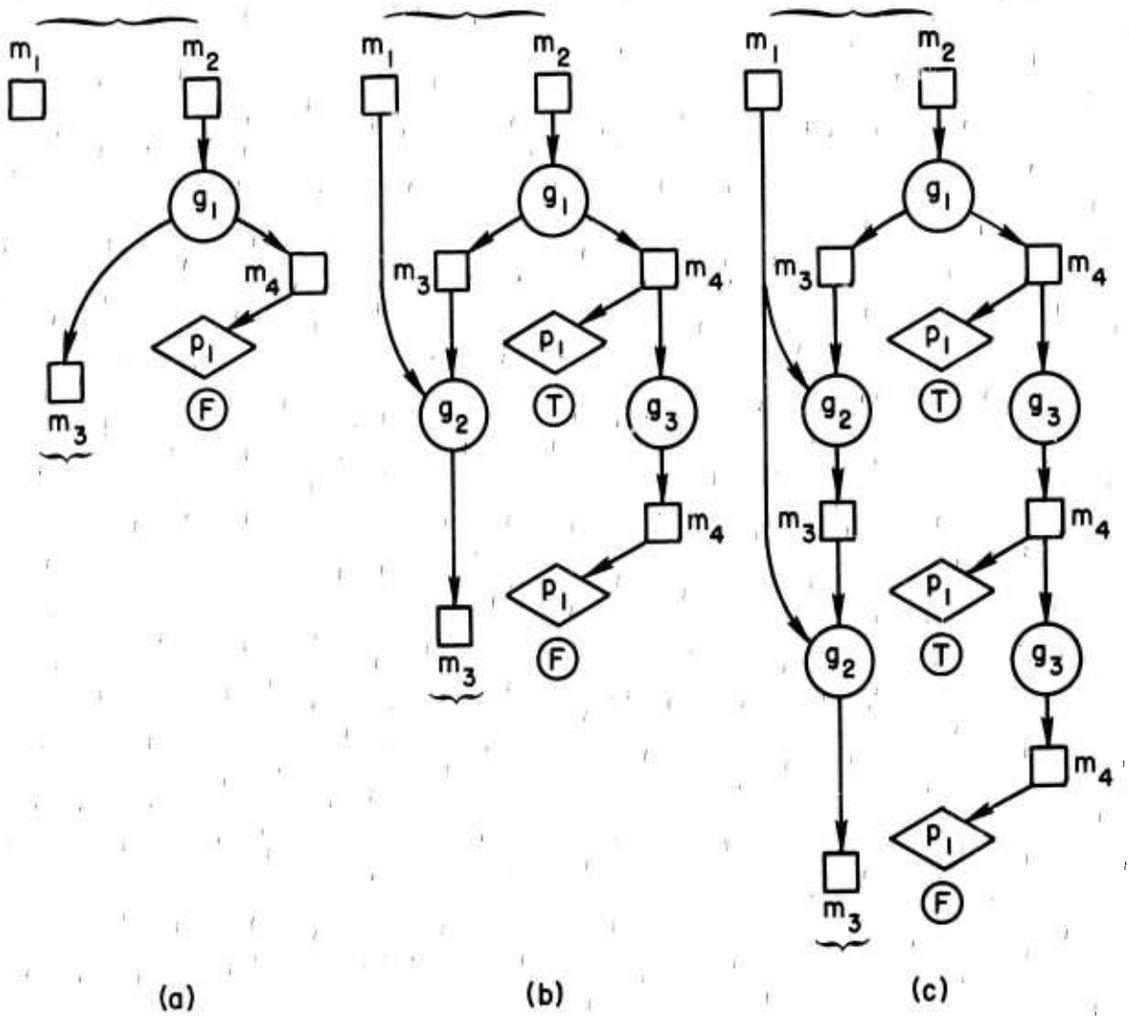
For schemata that include deciders, there will be a cascade composition of functions associated with each action of a decider as well as each



output cell. A determinate schema with k deciders could have 2^k distinct dadep graphs -- one for each combination of decisions that might occur in the course of some computation. For the schema S_1, there are just three dadep graphs because a decision of <u>false</u> by d_1 results in the absence of any action by d_2.



In general, a schema that represents an iteration defines an infinite set of dadep graphs. In the case of S_2, the three simplest dadep graphs are:

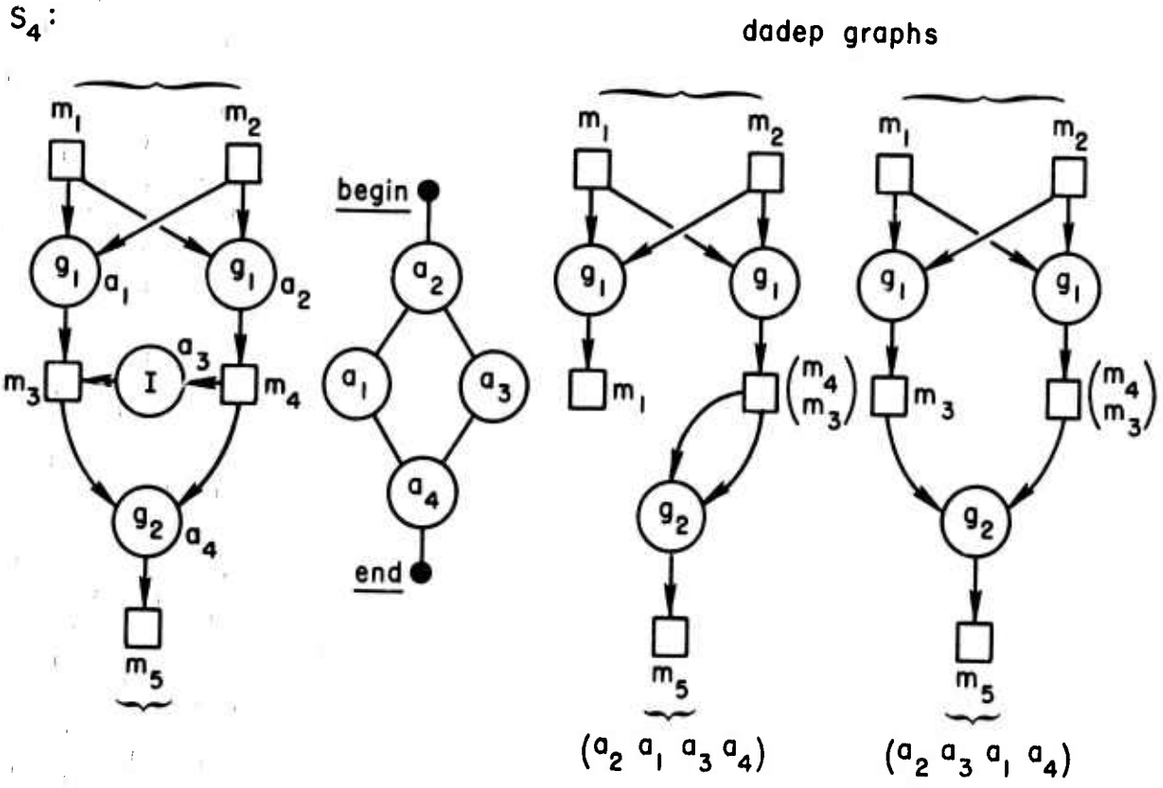


Certain properties are important in the study of schemata: A schema is persistent if the occurrence of one of two actions that could proceed concurrently does not inhibit or block the other action. Furthermore, a schema is commutative if the order in which two concurrent actions occur has no effect on the subsequent course of the computation.

Nondeterminate computation can occur only when a schema has a cell that could be assigned a value by one operator either before or after a value is assigned to or read from the cell by the action of another operator or decider. When this can happen we say the schema has a conflict.

By means of known methods it is not difficult to show that any computation schema that is persistent, commutative, and free of conflict is

guaranteed to be determinate. A more interesting problem is to determine the circumstances for which the conflict free property is a necessary condition for schemata to be determinate. We have studied two natural restrictions on schemata such that any determinate schema meeting the restrictions is necessarily conflict free. The first of these restrictions amounts to requiring that each action by any operator or decider in a schema participate in determining some output value. A schema meeting this restriction is said to be normal. The second restriction disallows control sets that permit repetition of a computation or test for the same m-tuple of input values. A schema meeting this restriction is said to be repetition-free. In schema S_4, repetition of the function designated by g_1 occurs. Because of the repetition, the conflict between operators a_1 and a_3 at cell m_3 fails to yield nondeterminate computations -- both dadep graphs define the same composition of functions.



For an elementary schema that is well defined, normal, repetition-free and determinate, all execution sequences yield the same dadep graph. In fact the dadep graph is a canonical form for this class of schemata. Thus the equivalence of any two elementary schemata can be tested by constructing their dadep graphs.

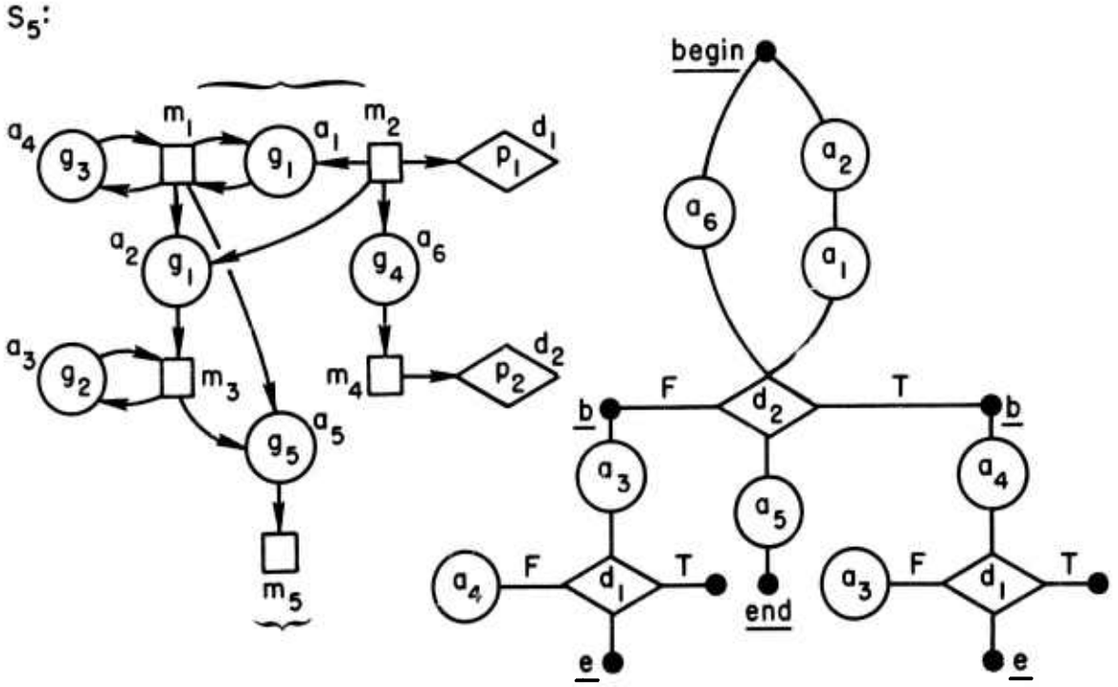
In the case of a normal, repetition-free schema that has deciders but no iteration, the class of computations represented is described by a finite set of dadep graphs, as shown for the schema S_1 earlier. Each pair of input values will be processed as shown in that one of the dadep graphs for which the evaluation of predicates agrees with the truth values given at decision points of the graph.

We can construct a table of two columns, called a conditional expression list, that characterizes the computations represented by a schema. Each row of the table corresponds to one dadep graph. In the left-hand column, we write a conjunction of the predicates that must be satisfied by the input variables for the corresponding dadep graph to describe the computation. In the right-hand column, we write the compositions of functions that specify the corresponding dependence of output values on input values. For S_1 we have:

Condition	Expression
$\overline{p}_1(x_2)$	$g_5(g_2(g_1(x_1, x_2)), g_3(g_1(x_1, x_2)))$
$p_1(x_2) \cdot p_2(g_4(x_2))$	$g_5(g_1(x_1, x_2), g_3(g_1(x_1, x_2)))$
$p_1(x_2) \cdot \overline{p}_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_1(x_1, x_2))$

Now consider the schema S_5:

S_5:



Schema S_5 has four dadep graphs and is characterized by a conditional expression list with four entries:

Condition	Expression
$\bar{p}_1(x_2) \cdot \bar{p}_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_3(g_1(x_1, x_2)))$
$\bar{p}_1(x_2) \cdot p_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_3(g_1(x_1, x_2)))$
$p_1(x_2) \cdot \bar{p}_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_1(x_1, x_2))$
$p_1(x_2) \cdot p_2(g_4(x_2))$	$g_5(g_1(x_1, x_2), g_3(g_1(x_1, x_2)))$

This table specifies the same class of computations as the conditional expression list for S_1, for we have the logical equivalence

$$\bar{p}_1(x_2) \equiv \bar{p}_1(x_2) \cdot \bar{p}_2(g_4(x_2)) + \bar{p}_1(x_2) \cdot p_2(g_4(x_2)) \quad .$$

Thus schemata S_1 and S_5 are equivalent. In general, noniterative schemata may be tested for equivalence by constructing their conditional expression lists.

Since an iterative schema has an infinite set of dadep graphs, its conditional expression list is infinitely long. For schema S_2 we have

Condition	Expression
$\bar{p}_1(g_1^2(x_2))$	$g_1'(x_2)$
$p_1(g_1^2(x_2)) \cdot \bar{p}_1(g_3(g_1^2(x_2)))$	$g_2(x_1, g_1'(x_2))$
$p_1(g_1^2(x_2)) \cdot p_1(g_3(g_1^2(x_2))) \cdot \bar{p}_1(g_3(g_3(g_1^2(x_2))))$	$g_2(x_1, g_2(x_1, g_1'(x_2)))$
. . .	. . .
. . .	. . .
. . .	. . .

We can show that, in general, two normal and repetition-free schemata are equivalent if and only if their conditional expression lists agree in the sense illustrated by our demonstration of equivalence for S_1 and S_5. When the lists are finite the existence of a decision procedure is

clear. At this time it is not known whether or not a decision procedure can be found for the more general equivalence problem.

VIII. CONTROLLED INFORMATION SHARING

The merit of the computer utility concept [15], lies in the ability of the users of the utility to build on each other's work. Thus the utility must provide orderly means for sharing access to procedures and data bases. We believe [16] that, to be successful, a utility must provide an environment in which a variety of information services may flourish and compete as private enterprises. Because proprietary and personal data will reside in the memory of a computer utility, access of users to stored information must be controlled so that only legitimate access is permitted.

Dean Vanderbilt has studied the implications of these requirements for sharing and access control on the organization and representation of procedures and data bases in a computer utility [17]. A computer utility must allow the owner of a program to authorize its use by other users without giving them the ability to view its internal structure. The execution of a program involves access to data and access to other programs. This additional information falls into two categories -- information that is associated with (shared by) all activations of the program; and the information that is associated with a particular activation (and not shared by several activations). The former category (Category I) consists of subprograms considered to be part of the program, subprograms of these subprograms, etc., and any data that are common to all activations of the programs. Category II information consists of all information passed as arguments to and from the program, and all temporary information generated during the particular activation.

During execution of a program, access to Category I and Category II information must be provided. Two aspects must be dealt with: First, the names used by the program to refer to this additional information must be bound to be the correct information. Second, the access control mechanisms of the utility must allow access to the information when it is needed.

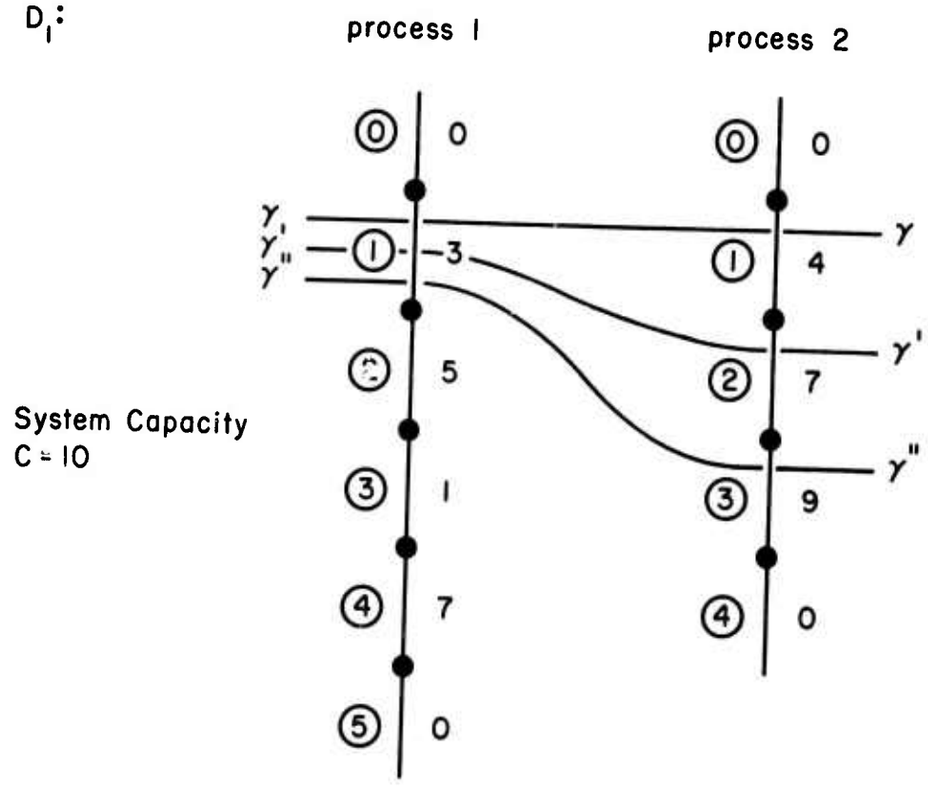
The Category I information is known to the owner -- the creator -- of the program, but not to the borrower. Thus the owner must specify the binding of names in the program to that information, and ensure that the information may be effectively referenced when needed during execution of the program. Since the program borrower should be granted no more access abilities than necessary, it must be possible for the owner to give the borrower the ability to access Category I information only in conjunction with use of the program. Thus, access abilities and binding information must be associated with the shared program so that the appropriate Category I information is available each time the program is executed.

The Category II information consists of information supplied by the program user and information created by the program. For the former, the supplied arguments must be bound to the program's call parameter names. The access abilities pose no problem since this information belongs to the program user. For the latter information, the process executing the program must be allowed to create information and to have it automatically bound to the appropriate names appearing in the program.

Dean Vanderbilt has designed an abstract program-execution environment [17] which offers one solution to the problems of implementing controlled access to shared information in a computer utility. This work uses a directed graph model of structured information that is closely related to the abstract information structures that form the foundation of our development of a base language, and is similar to the abstract "objects" used by the IBM Vienna Laboratory [18] for their work in formal semantics.

IX. RESOURCE SHARING WITHOUT DEADLOCK

Another form of concurrency is the cooperative activity of interacting computational processes, as in a multiprocess computer system. One form of interaction among processes is the implicit interaction arising from the sharing of limited resources. Consider, for example, two independent, sequential processes that progress through several distinct phases of activity.

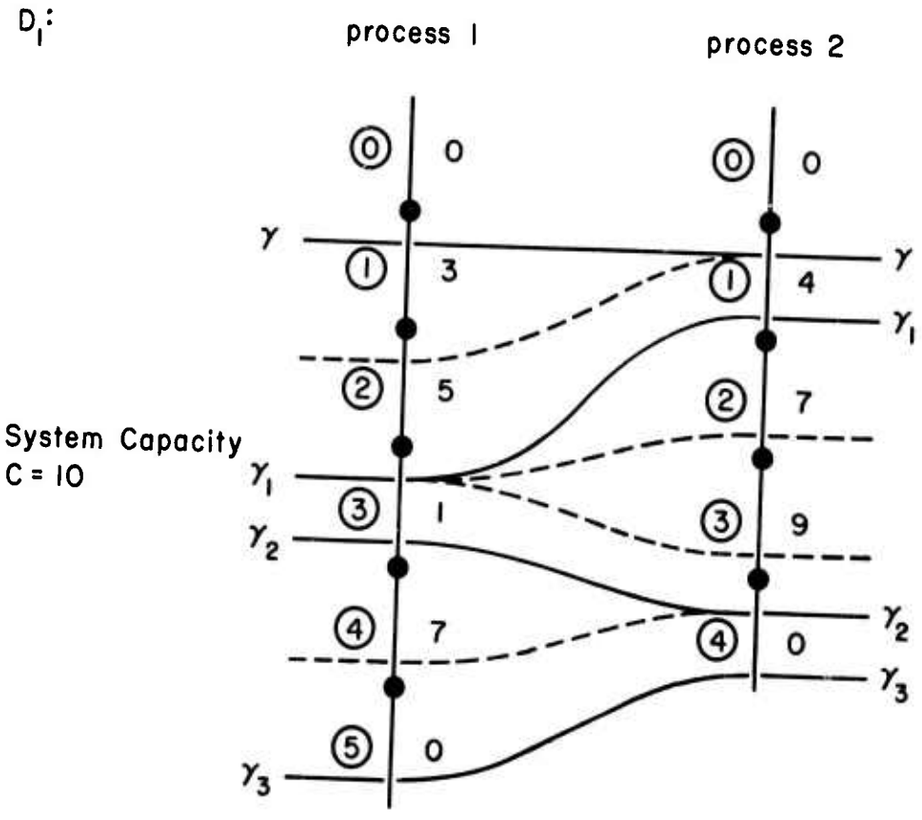


For each of its phases (identified by the circled numbers), a process requires the specified amount of a single resource type. The number of available units of the resource type (the system capacity) is C = 10. This representation of the resource requirements of a system of concurrent processes is called a demand graph. It is convenient to represent the composite state of the processes by a slice through the demand graph. For example, in the slice $\gamma = (①, ①)$ of demand graph D_1, both processes are engaged in phase 1 of their activity. Slice γ is feasible because the total resource units required is seven, which is less than the system resource capacity. If process 2 should complete phase 1, it could immediately proceed with phase 2, for the slice $\gamma' = (①, ②)$ is also feasible. However, process 2 could not continue into phase 3 of its activity because slice $\gamma'' = (①, ③)$ has a total resource requirement larger than the system capacity -- we say that slice γ'' is not feasible. The resource-allocation mechanism of a system should operate so that all processes can complete all phases of their activities, if possible, by a sequence of feasible slices. This kind of scheduling is not simply implemented if processes are assumed to retain control over resources during their transitions to new phases of activity. For instance, we must allow process 2 to retain the four units allocated to it for phase 1 while awaiting the release of three more units for its use in phase 2. Such hoarding occurs in computer systems where the resource may be memory areas, access to locked data bases, tape units, etc. When such hoarding is practiced, deadlocks can occur: Slice γ' in the demand graph D_1 is feasible, and represents a system state reachable by a sequence of feasible slices. Yet neither process can proceed beyond its phase in slice γ' for the lack of needed resource units -- the two processes are deadlocked. To avoid deadlock, the allocator must prevent the system from reaching the state corresponding to slice γ' even though the slice is feasible.

We call a slice γ in a demand graph safe if it is feasible and there is a sequence of phase transitions leading to a succession of feasible slices so that each process completes all remaining phases of its activity. If there is no such sequence of feasible slices, then slice γ is unsafe. Slice γ' in D_1 is unsafe. That slice γ is safe is demonstrated by showing, on the next page, a sequence of phase transitions to successive feasible slices that takes both processes through all remaining phases.

In these terms, the task of the resource allocator is to regulate the transitions of processes to new phases so that each slice attained is safe. It is not adequate to start the system of processes in a safe slice, for unsafe slices may be reached from a safe slice.

For demand graph D_1, we can discover that slice γ is safe by observing that process 1 goes through a phase of reduced demand during which



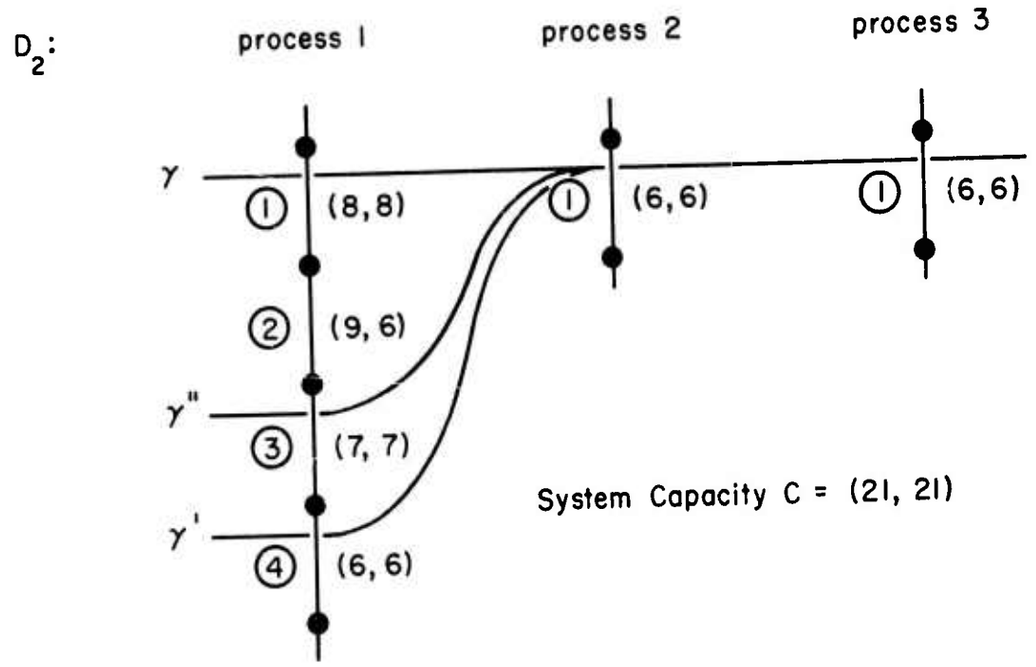
process 2 may advance to phase 4. In the absence of the detailed demand data given by the demand graph, this sort of reasoning cannot be applied, and less-complete resource usage is possible. For instance, if it is only known that processes 1 and 2 require maximum demands of 7 and 9 units, respectively, the system state represented by slice γ could not be permitted to occur.

In principle, one could examine all possible slices of a demand graph and determine whether each is safe before initiating any activity. However, this technique lacks flexibility, since adding a new process to a system of processes would require a suspension of activity while a redetermination of safeness of slices is carried out. It is also wasteful in that few of the slices of a demand graph will occur during a run of the system of processes. Incremental algorithms, which only test for safeness slices that are candidates for becoming the new current slice, are therefore of interest. Prakash Hebalkar [19] has formulated an algorithm for testing safeness that is non-enumerative and in which the amount of backtracking is minimal. This Safeness Algorithm attempts to construct a sequence of feasible slices from the slice under test to the terminal slice of the demand graph. The construction proceeds in steps, each of which consists of a series of phase transitions by one process. A step ends at the first phase that has a demand no greater than the demand for the process at the initial phase of the step. For demand graph D_1, the Safeness Algorithm produces

the sequence of steps $\gamma \longrightarrow \gamma_1 \longrightarrow \gamma_2 \longrightarrow \gamma_3$ to verify the safeness of slice γ. We have shown that failure of the algorithm to generate a sequence of feasible slices by which all processes complete their activity implies the slice under test is unsafe; conversely, success of the algorithm implies safeness. A resource allocator that uses the Safeness Algorithm to restrict system operation to only safe allocation states would make better use of resources without the possibility of deadlock.

For systems in which more than one type of resource is shared, we have formulated an extension of the Safeness Algorithm and established its validity. However, the amount of computation can have a nonlinear dependence on the number of phases of the processes in the demand graph -- a problem that does not arise for systems with a single resource type. This is not a failing of our particular algorithm: We have shown that a local algorithm (one that is not permitted a bird's-eye view of the entire demand graph) will, for some cases, have to exhaustively search a large set of slices to determine that a slice is safe. The following example illustrates why this is so.

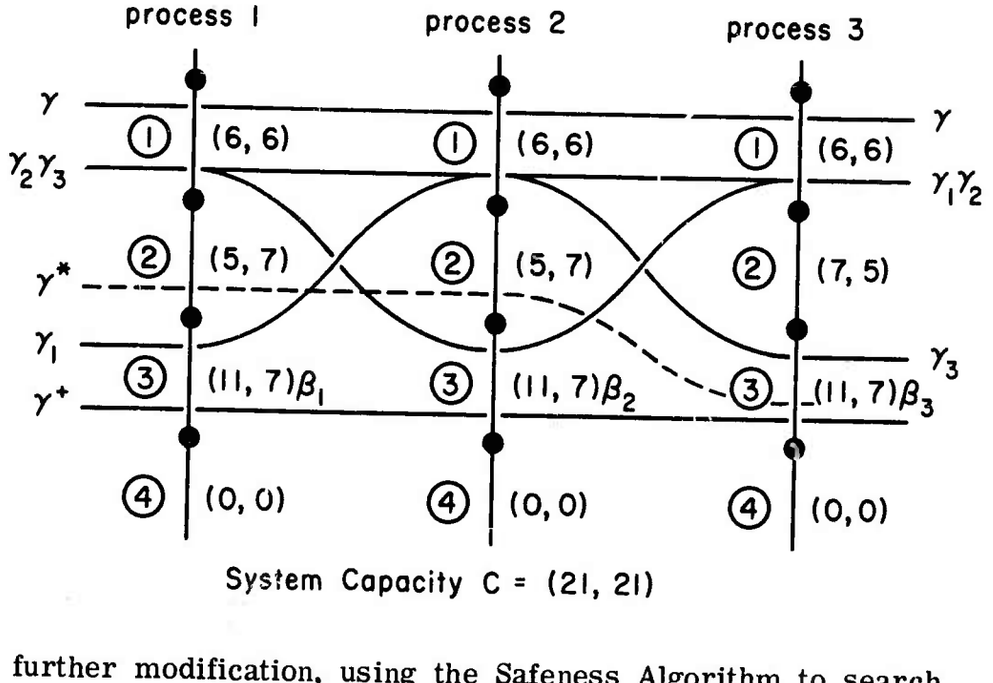
The extended Safeness Algorithm generates feasible slices in steps, as before. However, the series of phase transitions making up a step now ends only at a slice in which each component of demand is no greater than the same component in <u>each</u> prior phase of the series.



Thus in D_2, a step will consist of moving from γ to γ' rather than from γ to γ''.

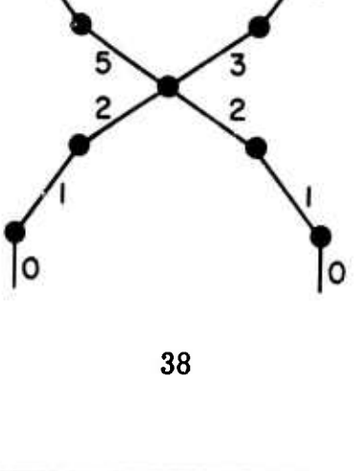
Now consider the demand graph D_3.

D_3:



Without further modification, using the Safeness Algorithm to search for a step from slice γ leads to failure for processes 1, 2 and 3 at slices γ_1, γ_2 and γ_3, respectively. Thus the algorithm would conclude (falsely) that γ is unsafe. A limited backtracking algorithm must discover some way of getting past the slice $\gamma+ = (③, ③, ③)$ consisting of the barrier arcs β_1, β_2 and β_3. From the study of the demand graph, it is evident that the slice $\gamma* = (②, ②, ③)$ must be used. But a local algorithm can determine this only through an exhaustive search of slices. The number of futile trials can be quite large.

The Safeness Algorithm can also be extended to systems in which explicit interactions between processes take place as well as the implicit interactions arising from resource-sharing. In studying this situation, we have discovered an interesting phenomenon, called intrinsic deadlock: There are demand graphs for which no schedule can permit the processes to complete their activity, for example:



This is a result of excessive hoarding of resources at points of (explicit) interaction. For this reason, among others, hoarding of resources at points of interaction should be held to the minimum.

The study of demand graphs is a perfectly general one that is not restricted to computer processes. Deadlock situations can arise from sharing of resources in road transportation, aircraft maintenance, and so on, and these operations can profit from analysis for the prevention of deadlocks.

X. WOODS HOLE CONFERENCE

The culmination of the year's activities of the Computation Structures Group was the sponsoring of an informal conference on Concurrent Systems and Parallel Computation. It was held at the National Academy of Sciences' Conference Center in Woods Hole, Massachusetts, during the first week of June 1970. Participants in the conference included six members of Project MAC and twenty-one persons representing most institutions in the United States that are carrying on theoretical research related to parallelism and concurrency.

The objective of the conference was to bring together people working along four distinct conceptual lines that we have found to be intimately related:

Representations of systems of concurrent events.
Speed-independent switching circuits.
Uninterpreted schemes of programs.
Cooperating sequential processes.

The conference was most successful in acquainting the participants with each other's ideas and in catalyzing many stimulating discussions.

Eleven technical papers were prepared for the conference, and were of such quality that they have been published collectively as a Conference Record [20]. For the conference we assembled an extensive collection of papers and reports related to the concepts of concurrency and parallelism. With the inclusion of a bibliography of this collection, the Record should be a valuable introduction to the field for interested research scientists.

Publications 1969-1970

Dennis, J. B., "Asynchronous Control Structures for a High Performance Processor", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, N.Y., 1970, pp. 55-80.

Gertz, J. L., Hierarchical Associative Memories for Parallel Computation, Ph.D. Thesis, Dept. of Electrical Engineering, June 1970, also MAC TR-69, AD-711-091.

(continued)

Publications 1969-1970 (cont.)

Patil, S. S., "Closure Properties of Interconnections of Determinate Systems", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, N.Y., 1970, pp. 107-116.

Patil, S. S., Coordination of Asynchronous Events, Ph.D. Thesis, Dept. of Electrical Engineering, June 1970, also MAC-TR-72, AD-711-763.

Vanderbilt, D. H., Controlled Information Sharing in a Computer Utility, Ph.D. Thesis, Dept. of Electrical Engineering, October 1969, also MAC TR-67, AD-699-503.

References

1. A. W. Holt and F. Commoner, "Events and Conditions". Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York (1970), pp. 3-52.

2. C. A. Petri, Communication with Automata. Supplement 1 to Technical Report RADC-TR-65-377, Vol. 1, Griffiss Air Force Base, New York, 1966. [Originally published in German: Kommunikation mit Automaten, University of Bonn, 1962.]

3. S. S. Patil, Coordination of Asynchronous Events. Report MAC-TR-72, Project MAC, M.I.T., Cambridge, Massachusetts, June 1970.

4. J. B. Dennis, "Modular, Asynchronous Control Structures for a High Performance Processor". Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York (1970), pp. 55-80.

5. S. S. Patil, "Closure Properties of Interconnections of Determinate Systems". Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York (1970), pp. 107-116.

6. J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture". Information Processing 68, North-Holland, Amsterdam (1969), pp. 484-492.

7. J. L. Gertz, Hierarchical Associative Memories for Parallel Computation. Report MAC-TR-69, Project MAC, M.I.T., Cambridge, Massachusetts, June 1970.

8. E. C. Van Horn, Computer Design for Asynchronously Reproducible Multiprocessing. Report MAC-TR-34, Project MAC, M.I.T., Cambridge, Massachusetts, 1966.

9. J. E. Rodriguez, A Graph Model for Parallel Computation. Report MAC-TR-64, Project MAC, M.I.T., Cambridge, Massachusetts, 1969.

(continued)

References (cont.)

10. F. L. Luconi, Asynchronous Computational Structures. Report MAC-TR-49, Project MAC, M.I.T., Cambridge, Massachusetts, 1968.

11. D. R. Slutz, The Flow Graph Schemata Model of Parallel Computation. Report MAC-TR-53, Project MAC, M.I.T., Cambridge, Massachusetts, 1968.

12. Y. I. Yanov, "The Logical Schemes of Algorithms". Problems of Cybernetics, Vol. 1, Pergamon Press (1960), pp. 82-140.

13. R. M. Karp and R. E. Miller, "Parallel Program Schemata". J. of Computer and System Sciences, Vol. 3, No. 2 (May 1969), pp. 147-195.

14. M. S. Paterson, "Program Schemata". Machine Intelligence, Vol. 3, American Elsevier, New York (1968), pp. 18-31.

15. R. M. Fano, "The MAC System: The Computer Utility Approach". I.E.E.E. Spectrum, Vol. 2, No. 1 (January 1965), pp. 56-64.

16. J. B. Dennis, "A Position Paper on Computing and Communications". Comm. of the ACM, Vol. 11, No. 5 (May 1968), pp. 370-377.

17. D. H. Vanderbilt, Controlled Information Sharing in a Computer Utility. Report MAC-TR-67, Project MAC, M.I.T., Cambridge, Massachusetts, 1969.

18. P. Lucas, P. Lauer and H. Stigleitner, Method and Notation for the Formal Definition of Programming Languages. Technical Report TR 25.087, IBM Laboratory, Vienna, June 1968.

19. P. G. Hebalkar, Deadlock-Free Sharing of Resources in Asynchronous Systems. Report MAC-TR-75, Project MAC, M.I.T., September 1970.

20. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York (1970).

COMPUTER SYSTEM RESEARCH

Prof. F. J. Corbató

R. Beatty
V. M. Berardinelli
M. Bromberg
M. C. Burnham
R. H. Campbell
O. D. Carey
J. R. Cecil
D. D. Clark
R. C. Daley
C. P. Doyle
S. D. Dunten
R. J. Feiertag
R. Frankston
R. L. Gardner
C. C. Garman
S. Garner
J. M. Grochow
D. L. Jones
R. K. Kanodia
E. W. Meyer, Jr.
N. I. Morris
M. A. Padlipsky
Prof. J. H. Saltzer
R. R. Schell
M. D. Schroeder
A. Sekino
T. H. Seymour
T. P. Skinner
N. J. Smith
J. W. Spall
M. J. Spier
A. Testa
M. R. Thompson
V. L. Voydock
M. B. Weaver
S. H. Webber

Guest

N. A. Adleman

COMPUTER SYSTEM RESEARCH

I. INTRODUCTION

The year from July 1969 through June 1970 was a critical year for the Multics (Multiplexed Information and Computing Service) system, which was again the major concern of the Computer System Research Group. During the reporting period, research and development efforts on Multics continued to be performed jointly with the General Electric Company's Cambridge Information Systems Laboratory personnel; and the assumption of responsibility for the administration, operation and maintenance of the system by the M.I.T. Information Processing Center was initiated. In addition, members of the group participated in work on the ARPA Computer Network and on computer graphics, reflecting the Group's shift in emphasis toward exploitations of the research base which the Multics system represents. The dominant role of Multics in the Group's efforts dictates that the bulk of this report will address itself to Multics; however, the new areas of interest will also be discussed in more detail subsequently.

As indicated in last year's report, 1 October 1969 was firmly set as the date on which Multics service would be made available to the general Project MAC and M.I.T. user community. This goal was achieved, offering a version of the system that was considerably improved over what was available at the end of the previous reporting period. Because the success and acceptance of the system by the user community is a key issue in the desired dissemination of the underlying concepts of Multics, much stress was laid on making the system more attractive to general users (i.e., not just system programmers). To this end, expansion and refinement of functional capabilities and improvement of performance were the areas on which the Group concentrated. An index to the success of these efforts may be found in Fig. 1, which shows the growth of the user community. By the end of the present reporting period, the operational version of the system again represented a considerable over-all improvement over the October First version. Table I (which will be discussed in greater detail in the section on Performance) furnishes a good indication of the improvement of the system over the year, in terms of gross performance. It is worth noting here that Multics, as of October 1969, was furnishing performance superior to that of the Compatible Time-Sharing System (CTSS) and is continuing to improve.

Of the conceptual goals discussed in the 1965 Fall Joint Computer Conference papers on Multics, most have been fulfilled, although a few key functions are still being worked on and should be installed in the next year. On the performance side, it is expected that the coming year will see the system beginning to support the original predictions of simultaneous users. Moreover, the original decision to implement

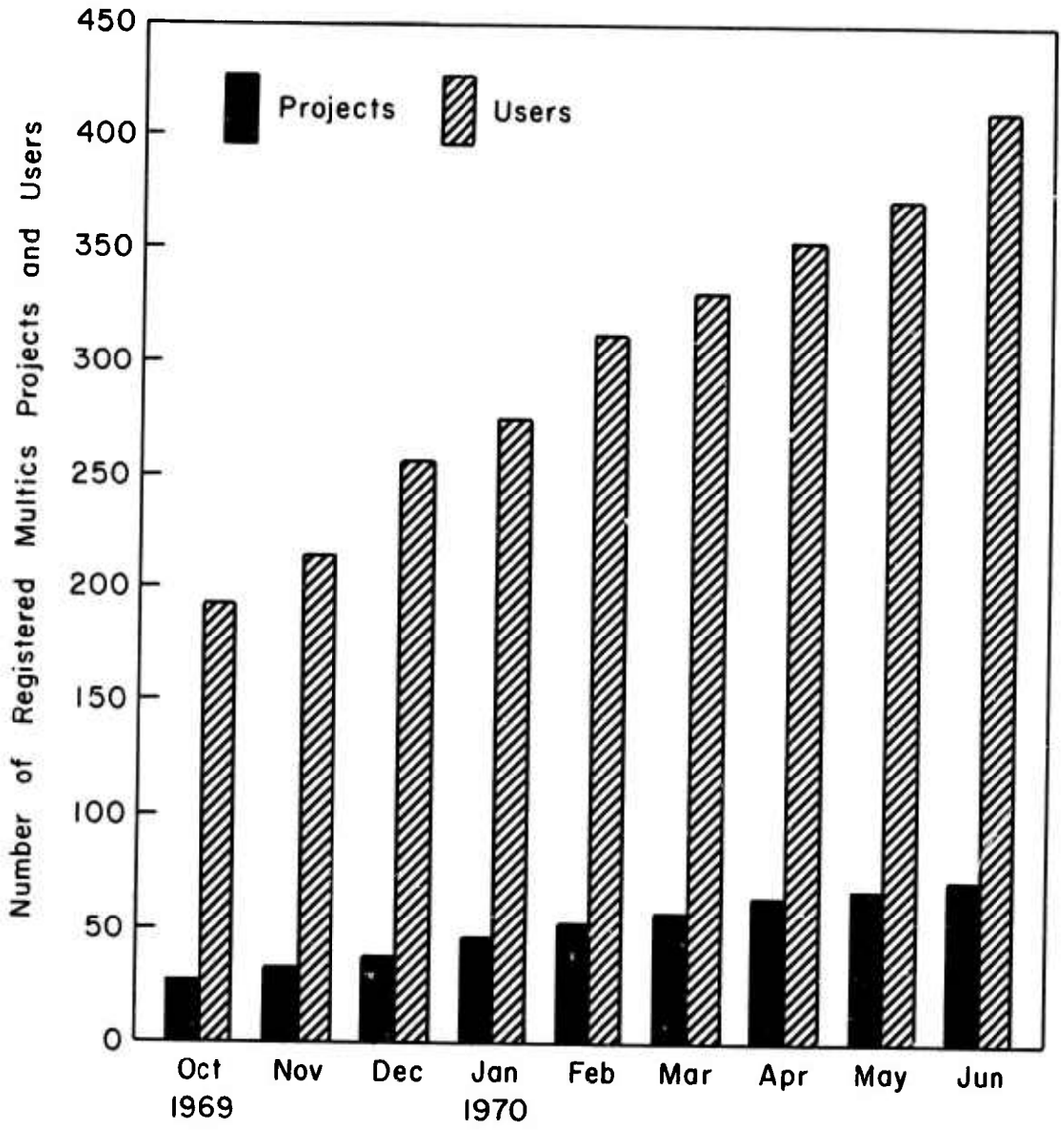


Fig. 1.

Multics in a high-level language has shown itself to be a wise one, as both the quantity and quality of the changes effected during the reporting period attest. Now that the basic Multics framework has been established, there is still much to be learned. New work is under way in two broad areas of interest:

(1) Work on the system per se addresses itself to deeper understanding of the issues involved in large, complex systems; and work is intensifying on the propagation of our results to other workers in the systems field.

(2) Using Multics as a springboard, new areas of interest in the use of sophisticated systems are being explored, especially the con-

Table I.

PDP-8 Script Performance Comparison between CTSS and Multics System 7.0, 4.0, and 3.0.11

	Multics 3.0.11 6/25/69	Multics 4.0 9/25/69	Multics 7.0 3/30/70	CTSS 8/1/69
Average load while measuring	12 users	24 users	21 users	17 users
CPU time charged	87.9 sec	50.3 sec	28.7 sec	70.97 sec
Real time required	2702 sec	2520 sec	2298 sec	2434 sec
CPU time per interaction	1.91 sec	0.79 sec	0.436 sec	1.07 sec
Average response time	~18 sec	12.4 sec	5.2 sec	5 ± 1 sec
Dollar charge	$11.01	$6.55	$3.99	$6.50
Cost per console hour	$14.70	$9.10	$6.27	$9.62

Notes:

1. All figures are for two passes through the standard "Fortran debugging" script.
2. All prices are based on a Multics CPU charge of $420/hr, a Multics dialup charge of $1/hr, a CTSS CPU charge of $330/hr, and no CTSS dialup charge.
3. Note that this script is, on balance, a smaller than average user of core memory, and is unfavorably charged by a Multics CPU price based on average core memory usage. (It is expected that in the future the Multics charging policy will be revised to correct this inequity.)

cern with the ARPA Network and with the support of terminals with graphical capability. There is also a continuing interest in dealing with problems of data base management and of protection.

II. THE OCTOBER FIRST SYSTEM

When Multics was formally released to the user community, the version of the system made available at the time was designated "System 4.3". For convenience of subsequent reference, it should be explained that the numbering system adopted for system versions indicates major changes in the supervisor by incrementing to the next whole number, and less far-reaching changes to the basic ".0" system by incrementing to the right of the decimal point. Accompanying the "October First

System" was the initial release of the Multics Programmers' Manual (MPM). This users' manual originally comprised some 500 pages, under the following major headings: Introduction to the Concepts of Multics; Introduction to the Use of Multics; Reference Data, Standards, Conventions, Formats, and Codes; and detailed descriptions of the various commands and subroutines -- both system-supported and user-furnished -- available. By year's end some 100 more pages had been added to the MPM.

It is interesting to note that even before the 1 October deadline, Multics had been successfully used by non-system programmers: An M.I.T. Summer Course in programming linguistics furnished the system's first extensive use by non-members of the Group as Multics was employed on an experimental basis for the performance of assignments by students in the course. The assignments involved use of the PAL translator, which was implemented by graduate students working under Prof. Arthur Evans, Jr. The PAL translator itself was coded in BCPL (discussed in last year's report). The success of this experiment was gratifying for several reasons, in that it not only demonstrated the general utility of Multics for practical applications, but also supported the belief that subsystems would be relatively easy to embed in Multics, and, of course, furnished evidence that the system was actually shaken-down enough to support general users.

Functional Capabilities: The Standard Service System Approach

A major feature of the October First system was the "Standard Service System", a set of closely audited, optimized commands and subroutines of interest to general users. The Standard Service System includes the command processor (the "mini-Shell", discussed below) and such basic commands as the "edm" context editor, the file and directory manipulation commands, and a Fortran compiler. All modules in the Standard Service System are carefully coded and audited for high performance and small working sets, adhere to a standard user interface for argument specification, and are implemented in either a subset of the EPL language selected for efficient code generation or with the more efficient PL/1 compiler. All the components of the Standard Service System are organized into a special system library, which is the first to be searched when a procedure is initially referenced in a process. The net result is to furnish the general user with nearly optimum performance, particularly for tasks that do not inherently require a large share of the system's resources. Indeed, this conscious bias in favor of the small user has emerged as a cornerstone of the system. To aid the user in the identification of Standard Service System routines, the individual writeups section of the MPM is so organized as to group Standard System commands and subroutines in sections of their own.

The most important single aspect of the Standard Service System is the command processor. The version of the command processor in the October First system was a considerably improved one, having been completely recoded to adhere to Standard Service System standards. A more efficient subset of command language features was isolated, and only this subset was handled directly by the new command processor, or "mini-Shell". The full command language was still available, but was still processed by the relatively cumbersome "full Shell". The new command loop also speeded up console input/output operations, by avoiding the invocation of the full I/O System until and unless the user explicitly invoked it himself. I/O was also minimized by the elimination of the "wait" message which was previously typed out on receipt of each command. Further, various support subroutines were bound together with the mini-Shell.

Functional Capabilities: Resource and Access Control

Another important functional capability added to the system for its public debut was resource control: With use of the system no longer limited to system programmers, it was necessary to install quotas for disk storage. Also, a minimal accounting system was incorporated, to allow for the maintenance and billing of user accounts, involving both disk storage and central processor time used. Accounting also required reorganization of the data bases employed in the System Control subsystem. Other changes in System Control for the October First system were the ability to automatically log out a process when its console was hung up, and the ability to automatically create a new process when a running process became incapable of proceeding. The presence of non-system programmers on the system also necessitated the installation of full access control; each user upon logging in is assigned his proper process group identification which allows the access control machinery to function correctly.

Performance Improvements

In addition to the extensions of functional capability discussed above, several performance improvements were included in the October First system. The primary one was a fully reimplemented Traffic Control module, which constituted System 4.0. The basis of this change was a tightening of the interface between the Traffic Controller and the File System, allowing the highest-priority process in the scheduling queues to be specially handled when requesting paging. This is quite desirable, in that it minimizes the likelihood of "thrashing" -- that is, of having a lower-priority process bring in pages (while the higher-priority process is waiting for its own pages) which dislodge pages still needed by the higher-priority process.

Other performance-improvement tasks included the following:

(1) Procedure linkage sections were redesigned in order to reduce the number of linkage faults in the system.

(2) The need for several data segments which were carried in each process was eliminated by means of consolidation with other segments, thus minimizing the working set of each process; this strategy pays large dividends, and the reduction of "per-process segments" is an on-going task.

(3) The system libraries were reorganized to allow more efficient searching.

(4) The device interface module (DIM) for the high-speed printer was recoded in order to improve its buffering strategy and moved to ring 0 (the supervisor's protection ring); these two performance improvements were rather important, as many users rely quite heavily upon listings from the high-speed printer.

(5) Considerable attention was paid to the metering of such key areas as the Traffic Controller and the typewriter DIM, to furnish vital performance information under full user load as a guide to further refinements.

Maintenance Tools

Maintenance tools made available for the October First system included the following: The backup reloader was extended to restore full hierarchy information (e.g., dates modified and used, access control on directories) for reloaded files. The backup dumper was modified to facilitate the taking of complete dumps by the Operations staff. Also, the reloader was modified to allow selected files to be retrieved from the backup types. The "Salvager" program discussed in last year's report (which corrects inconsistencies in the File System hierarchy after a system crash, allowing prompt restoration of user service) was improved and made a standard operational tool. It should be emphasized that these File System maintenance tools have been of crucial importance in maintaining satisfactory system operation for the general user. Also, a set of commands was developed to facilitate changes to the standard system and to allow rapid installation of new systems.

III. THE CURRENT STATE OF THE SYSTEM

Functional Capabilities: System Version Iterations

System version 4.0, an iteration of which was in operation on 1 October, included the new Traffic Controller and the first wave of Standard Service System modules. Four more major changes to the supervisor had been introduced by the end of the reporting period, designated by the version numbers 5.0 through 8.0:

System 5.0

The first changeover, to System 5.0, took place with the introduction of new hardware, DSU270 disks.

These devices, which are fixed-head disks, offer more efficient secondary storage than the original DSU10 units. Software changes necessary to support the new hardware were made in such a fashion as also to facilitate the anticipated introduction of new, larger DSU170 units (equivalent to the IBM 2314) at some time in the future. Both the DSU270's and the DSU170's offer, in addition to speed, the highly desirable feature of expandability. That is, as the need for more secondary storage increases with the increased user population of the system, more disk units may simply be brought in and connected.

System 6.0

System 6.0 featured a major reworking of the command processing loop and a general expansion of the scope of the Standard Service System.

The command loop was redesigned and recoded in PL/1, which had by this time become appreciably more efficient than EPL both in compile time and in generated code. It no longer avoided the full I/O system, since the performance of the central module of that subsystem, the "I/O switch", had also been upgraded appreciably. These changes dropped the distinction between "mini" and "full" Shell, allowing users to employ whatever features of the command language they wished, at low cost in time. The key issue here was a reorganization of the command processor (the term "Shell" having also been dropped) such that the user pays only for those features which he explicitly invokes, rather than having to use the full machinery for even minor tasks. The new command loop also enabled writers of private (and public, for that matter) commands to acquire the commands' arguments in a less-awkward fashion, and in general to interact with the command loop more cleanly. A deeply embedded per-process segment was eliminated ("process_info"); this was an independent task since it involved changes to a large number of segments, as well as rather extensive changes to the Interprocess Communication facility, which were not practicable earlier. Also, more commands, subroutines, and the Interprocess Communication facility were converted to PL/1 and made to conform to Standard Service System criteria. Finally, the handling of conditions and signaling was reimplemented.

System 7.0

System 7.0 was a major performance breakthrough, since it incorporated a new, "fast" Page Control module.

The key here was the exercising of an option which had always been open -- that of going from compiler language to assembly language for crucial system modules. As mentioned in previous reports, the use of compiler language in implementing a system is quite powerful in terms of the ability to make major strategy changes quickly and the ability to present underlying algorithms clearly. A further advantage is, of course, the fact that there is some "leeway" inherent in the object code, so that hand-coding can be used to squeeze out even higher performance once the issues of the strategy and algorithm have been iterated to an apparently final level. The fast Page Control module performed as well as had been expected, decreasing its portion of page-fault processing time from on the order of 10 msec to on the order of 2 msec.

System 8.0

System 8.0 also involved the paging mechanism, furnishing the ability to selectively pre-page a process when it was about to be run and the ability to "post-purge" (i.e., release core space occupied by) selected pages when the process has finished its time quantum.

The pre-paging aspect further biases the system toward the "small user" in that it increases the likelihood that he may be able to interact without taking any page faults. (Actually, all users benefit from the pre-paging, since the number of page faults is at least decreased, even if it is not eliminated, for each interaction.) By the end of the reporting period, the exact impact of pre-paging and post-purging had not yet been evaluated, because tuning experiments were still being performed.

Functional Capabilities: Supervisor Changes

Numerous other significant changes to the supervisor were made, although they did not entail new system version numbers. One of the most important of these was the solution to the so-called "unlinking problem": When a program has been executed in a process, both a pure-procedure object segment and a linkage section (which contains process-dependent information) are introduced into the process's address space. However, if the program needs to be recompiled, the linkage section of the old version, which has been added to a single "combined linkage segment" in the appropriate protection ring, will erroneously be employed when the new object segment is executed. Previously, the solution was to create a new process after recompiling, but this is time-consuming, both in terms of process creation and in terms of having to re-do links which had been "snapped" in the old process. Therefore, in conjunction with an improved PL/1 version of the system Linker program, a facility was added which allows compilers

automatically (and users, by explicit command call) to cause removal of a named program, including its linkage section, from the address space of a process. This makes the debugging process far less cumbersome. Additionally, the PL/1 Linker is more efficient than the older EPL version.

Another area of the supervisor that underwent significant change was the "Gatekeeper" module. This module manages the transfers of control among Multics protection rings. A large improvement in speed was effected by a new Gatekeeper which handles ring-crossings into the supervisor's protection ring as special cases. (Such a strategy is made possible by the fact that the ring-0 environment is rigorously defined, so that the premium for generality of preparation need not be paid when the ring being entered is known not to require that generality.) Also, an important advance in system security came about from the inclusion of full argument validation on calls to the supervisor; for, by accident or by design, it is possible for a non-supervisor program to furnish bad arguments when calling the supervisor, and if the arguments are not validated (by the Gatekeeper) when the call is made, the highly privileged supervisor routine could inadvertently either destroy or reveal vital system information.

Functional Capabilities: Command Repertoire

The system's command repertoire also has been strengthened considerably. Perhaps the most valuable command of all is the PL/1 compiler. As has been seen, and as will be seen further, recompilation in PL/1 has led to improved performance in all areas of the system from the supervisor modules to individual commands. Furthermore, it should be noted that the G. E. Cambridge Information Systems Laboratory team responsible for the compiler won a General Electric corporate award for their work, and that the compiler is believed to be the best PL/1 compiler yet implemented. Aside from general improvement, full implementation of most defined language features, and the addition of object-code optimization, the compiler also benefited from the inclusion of more complete I/O facilities, in accordance with the full PL/1 language specification. PL/1 is expected to lend itself quite well to the implementation of data-base management subsystems, particularly because it is able, both directly (through its "based storage" facility) and indirectly (through PL/1 I/O), to take full advantage of the virtual memory that Multics affords. This should allow very large data bases to be manipulated with far greater ease than they could be by a PL/1 on a conventional system. Another feature added both to PL/1 and to the command loop in general is an improved mechanism for signaling and handling "conditions" as defined in the full PL/1 language specification. Signaling is already employed in the system's quit-handling and fault-handling mechanisms, and will be playing a larger role in error-handling in the future.

Another heavily used command which was vastly improved is the Multics context editor, "edm". Taking advantage of the power of PL/1, a new edm command was installed which offered an order-of-magnitude improvement in the time needed to locate an arbitrary string in a file.

Finally, several new commands were added to the repertoire, including a BASIC compiler, which will be discussed in more detail below. Of particular interest to system programmers has been the development of a new interactive debugging program, "db", which is far more rich and flexible than the earlier "probe" command. By requesting the generation of a symbol table during compilation, the user of db may refer to variables symbolically when debugging and refer to source-code statements by location. The ability to set breakpoints is also afforded. Another new command, called "help", is of considerable interest to general users, for it prints out usage information about named commands on-line; working from ordered files, the command pauses between levels of complexity to interrogate the user as to whether more help is desired. Still another interesting command, called "mail" allows users to send messages to other users by placing files in the others' directories. Card reading and punching facilities were also made available; the former is of great value for CTSS users shifting over to Multics. In addition, a large number of already existing commands were converted to Standard Service System standards during the reporting period.

The Student Information Processing Board Subsystem

The BASIC compiler (borrowed from the Rome Air Development Center time-sharing system) led to a windfall in that it became the vehicle for a subsystem developed by the M.I.T. Student Information Processing Board (SIPB). Using Multics, the SIPB group developed in a surprisingly short time the "SIPB-O System", a subsystem that gained many adherents among student users. This subsystem constitutes a closed environment that makes available a limited number of low-cost, easy-to-use commands. The commands include a subset of Multics commands as well as special SIPB-O ones. It was found that many students availed themselves of the subsystem for homework assignments. Among the features of SIPB-O are the BASIC compiler and editor, a special command processor, numerous "help" files, the ability to have many simultaneous users with a single working directory, and a library of game programs for demonstration purposes. This venture was gratifying not only because of the interest generated, but also because it further demonstrated the relative ease with which subsystems could be developed under Multics -- in this case, in two or three weeks (during which work on the subsystem was only part-time) by users who, although highly motivated and quite talented, had no previous Multics experience.

IV. PERFORMANCE

Summary

Progress was quite good in the area of system performance. By the end of the reporting period, loads in excess of 30 users were common during day-to-day "one-CPU" operation, and system response was adequate at that level. (During the next year, it is hoped to increase the acceptable load to approximately 50 users for the single-processor, 256K of core memory configuration; with both processors and 384K of core, a considerably larger load will be supportable.) Note that, although the system is operated most of the time with the system partitioned (one CPU and 256K for service, the other CPU and 128K for development work), there are scheduled sessions each week during which the full dual-processor configuration is used as the service system. These full-configuration sessions allow particular attention to be paid to issues of reliability, tuning, reconfiguration, and operational procedures. The remainder of the time, the development system is reserved for the testing and checkout of new systems, so as not to disrupt normal service.

Principles

Three principles worthy of note emerged from the Group's work in the area of performance improvement. The first is a design principle -- or attitude -- which holds that system modules should be so arranged as to cause the user to pay for only those features that he explicitly invokes. This was the approach taken in, for example, the redesign that changed the mini- and full-Shells into the command processor (System 6.0). It has also been encountered in many other areas of the system. The basic point is that generality of function need not be sacrificed to efficiency, but neither should it be achieved through an implementation that is inefficient for commonly used specific functions. The second principle is that careful and extensive metering of system performance is indispensable. Indeed, much valuable tuning information was derived from a few simple commands that allowed Group personnel to "browse" over the performance characteristics of the running system. This information, in conjunction with tuning commands, allows system administrators to dynamically change from any console parameters affecting system performance. (It should be noted that controlled system experiments are elaborate to set up, and nearly impossible to repeat; as a consequence, on-line tuning with a "live" user population has been an important stratagem.) The third principle is that judicious choice of which system modules to convert to hand-coding pays large dividends. The striking success in the Page Control area was mentioned previously. At year's end there were strong indications that hand-coding the argument validation portion of the Gatekeeper would also have very significant effects upon system performance. However, most of the system will continue to remain in PL/1.

Comparison with CTSS

Table I shows the rather dramatic improvements accomplished during the year, with both CPU time and dollar charge decreasing to roughly one-third of the levels at the beginning of the period. The figures are gathered from the execution of a fixed series of commands, input to Multics by using the PDP-8 display computer over a Dataphone line. The "script" entails the inputting, compilation, editing, recompilation, and execution of a Fortran program that calculates prime numbers. In broad outline, this "debugging sequence" is fairly typical of time-sharing system use, although it does not really take advantage of or exploit the greatly increased generality and functional features of the Multics system; nevertheless, the Fortran "script" is run on a regular basis since it can be considered a "worst case" comparison between Multics and other systems. As can be seen, the version of Multics that was current when the system was made generally available on 1 October furnished somewhat superior performance to that of CTSS in most categories. Only the average response time (how long it takes for a command to be reacted to by the system) compared unfavorably at that point in time. By the Spring of 1970, Multics showed a clear superiority over CTSS in all categories. (In the area of response time, where the two systems appear from the table to be essentially equivalent, it should be noted that Multics was more heavily loaded than CTSS at the time the statistics were gathered; for equal loads, the response time on Multics beats that of CTSS). Unfortunately, the 8.0 version of Multics (which was installed in June of 1970) had not yet been measured in all these categories at the time this report was written. However, indications from other metering tools and the subjective "feel" of the 8.0 system are that it has surpassed the performance of the 7.0 system by a substantial amount. (Further tuning of the 8.0 system is expected to make the difference even more noticeable.)

V. HARDWARE

Hardware on Site

The major change in the hardware complement during the year was the previously referenced changeover to fixed-head DSU270 disk units. These devices offer a transfer rate of approximately 55,000 36-bit words per second, with a mean access time of 26 milliseconds. Each unit has a capacity of 2.5 million words, but the number of units on-line to the system is essentially limited only by the number of controllers. Currently, 10 to 15 disks are being employed through a single controller. An important consideration here is the fact that, as system usage increases and a need for increased disk capacity is felt, more disks (and controllers) can be added without necessitating major software changes. Actually, it is anticipated that the DSU270's will

eventually be augmented by DSU170's, to which similar considerations apply. The DSU170's (which are GE's equivalent of the IBM 2314 multiple disk-pack drive) offer a transfer rate of 69,000 words per second, have a mean seek time of 75 msec, a mean latency of 12.5 msec, and will be more economical when they become available. (Initially, an IBM 2314 unit will be used.)

Hardware Performance

Difficulties with the performance of the hardware, leading to system crashes, were experienced during the reporting period. Indeed, with the increase in the stability of the software, hardware-caused crashes came to outnumber those attributable to software. System-crash analysis revealed that the electrical grounding of the GE 645 was inadequate. (At one point, the use of a vacuum cleaner to remove the chads from the card punch could cause the system to crash!) Suitable changes were made in the machine room, but, at year's end it proved necessary to run a ground cable to the basement of the building to make the grounding adequate. Another hardware problem that showed up in crash analysis was the occasional mis-writing of the first 64 words of 1024-word drum records. This proved to be the result of a subtle design error in the drum timing which was corrected at the end of the reporting period. System crashes were also caused with some frequency by core-memory failures and DSU270 problems. These problems are currently receiving intense attention from hardware specialists who have been brought in by General Electric from its Phoenix facility. Many of these hardware problems were only exposed under the heavy system loads experienced during the latter half of this reporting period. It is expected that system reliability will be brought to a satisfactory level during the next year.

VI. ADMINISTRATION AND OPERATIONS

In November 1969, the day-to-day administration and operation of Multics was turned over to the M.I.T. Information Processing Center (IPC), allowing the Computer System Research Group to devote itself more fully to its research and development role. Responsibility for providing user consultation and for distributing documentation was also assumed by the Center. The GE 645 operators are also now administratively responsible to IPC.

Excerpts from the announcement to the M.I.T. Faculty and Staff by Richard G. Mills (then M.I.T. Director of Information Processing Services) are of interest:

> "The Center will cooperate with Project MAC in the continuing development of the Multics system, while providing the M.I.T. community with a powerful, sophisticated, and reliable remote-access computer utility. We expect that many of the application

programs now operating under CTSS, and possibly other user-developed subsystems now running in other time-shared computers, will be transferable to the Multics environment with an acceptable level of program-modification effort. Once in Multics, these programs can begin to benefit from the improved capabilities that are available in the new system. The cost-performance ratio for Multics, initially about equal to that of CTSS, will improve considerably over the three-year initial period. Multics will supplant CTSS as the Institute's primary general-purpose time-sharing system, and CTSS service will be terminated (with appropriate notice) when Multics has proved its ability to support the work now being done on CTSS . . .

"This announcement carries out the new policy of explicitly committing to provide a major service component for a stated minimum time period. In the case of Multics, a commitment period of three years, beginning 1 October, 1969, was recommended by the Information Processing Advisory Committee, endorsed by the Information Processing Advisory Board, and approved by the Provost.

"The significance to users of such a commitment is that it provides a basis for planning and proposing research and educational tasks with the assurance that the computer-system base on which they rest will not be unexpectedly swept away. A three-year minimum commitment, of course, does not mean that we expect Multics to be terminated at the end of three years; in fact, we would hope after a year of experience with the system to announce a substantial extension of the commitment period."

At the end of the reporting period, the new arrangement was working well, with IPC personnel becoming increasingly involved with system-maintenance functions. In addition, some IPC personnel were making contributions to the system's command repertoire, the "help" and "mail" commands mentioned above being prominent examples.

VII. ARPA NETWORK

Background

In order to study the issues involved in large-scale computer networks and to enable a widespread user community to benefit from the work being performed on a number of advanced computer systems, the Advanced Research Projects Agency (ARPA) has initiated research into the formation of a computer-to-computer network among those systems that have been developed under its sponsorship. Multics is, of course, one of these systems, and considerable work was done on the network by members of the Group during the reporting period. (In the following discussion, each system that is a node of the network is referred to

as a "host'. At any given network site, there may be one or more host systems; at Project MAC, both the Multics GE 645 system and the Dynamic Modeling and Computer Graphics Groups' PDP-6/10 system are network hosts.) Since the network is in itself a very large-scale computer utility system, it is expected that the network will play an increasingly larger role in the Group's activities in years to come.

Physical Message Communication

The ARPA Network involves a multi-level hardware and software system. At the lowest level is the physical-communication network, managed at each node by a specially built device known as an Interface Message Processor (IMP). This hardware, along with standard control programs in the IMP, provides the basic facility of sending raw messages from one computer to another. A special hardware interface between M.I.T.'s IMP and Multics has been constructed by A. K. Bhushan, and, as of July 1970, the communication path between the IMP and the Multics I/O module is in final checkout.

Logical Communication Paths

A higher level of network control protocol must be supported by a software module known as the Network Control Program (NCP). This module must be implemented within each host computer system to factor the raw message-transmission capabilities provided by the IMP network into generalized communication facilities for individual user processes at a host.

The network participants at Project MAC, several of whom are members of the Computer System Research Group, have been engaged in discussions with representatives of other sites concerning the definition of a common network protocol. Each site will implement an NCP which acts jointly with other NCP's according to the protocol in creating, maintaining and destroying communication paths. As of June 1970, the Network participants appear to be in fundamental agreement concerning the protocol and are proceeding to settle the details.

Software modules implementing communications between the Multics NCP and its IMP are currently being coded. The NCP is under design, and preliminary coding has begun. We intend to have an operational NCP communicating with the Network by early Fall.

Inter-Host Software Protocols

Several still-higher-level software protocols are possible, the most important of which is a "logger" protocol. Because the NCP only provides communication facilities between two existing processes, some process must exist on each host system which agrees to listen to the Network and create processes for Network users upon proper identification. In Multics the Answering Service process performs this function for users

dialing up over typewriter channels, and it will be modified to provide this function for Network users as well. However, there must also be some protocol by which a potential user first gets the attention of the logger process, establishes communication with it, then establishes communication with the created process. As of June 1970, the issue of a standard logger protocol for all hosts is unresolved. Network participants at Project MAC and Lincoln Laboratory have agreed to take the lead in formulating such a protocol. The Network participants will probably take up this issue once the basic NCP protocol is officially promulgated.

VIII. GRAPHICS

Another area with which the Group is becoming more involved is that of terminals with graphical-display capabilities. The basis for this involvement is twofold: not only is it desirable for Multics, as a general-purpose system, to support such display terminals, but it is also the case that a major point of interest of the ARPA Network is a wealth of sophisticated display hardware and software that will be made available. Graphics-related work during the reporting period focused on the development within Multics of a General Graphics System and on a new teletypewriter device interface module (TTY DIM).

The Multics General Graphics System

The first version of the Multics General Graphics System was developed during the reporting period and is currently employing the DEC PDP-8/338 display computer as its display device. Because of the wide variety of displays that exist and may be connected to Multics, a particular objective of the graphics effort is to avoid the kind of fragmentation among users that occurs when a given group writes a software package that is keyed to a single device: not only is the user tied to that device, but the resulting software cannot be shared with users of slightly different but functionally equivalent hardware. To avoid this situation, it is planned that a Multics graphics user will manipulate device-independent three-dimensional picture descriptions in a per-user "working graphic segment" through the use of General Graphics System primitives. When a user issues a display call, a "graphic structure compiler" for his particular device type is invoked to produce a display command stream and dispatch it to his particular type of display device through the Multics I/O system. A graphic structure compiler which produces an ASCII-encoded command stream for a two-dimensional static display such as the ARDS console is currently working, with the PDP-8/338 used as a simulated ARDS. When the new ring-0 typewriter DIM (discussed below) is installed, it will convert this command stream into actual hardware commands for the ARDS. Alternatively, the command stream can be sent as-is through the ARPA Network for subsequent interpretation by the computer

serving the remote user's display. Work is currently under way to develop a graphics structure compiler capable of providing a general three-dimensional representation which may also be transmitted through the Network. Note that, when new graphical devices are added to Multics, it will be sufficient simply to add another conversion table to the TTY DIM; no changes to the graphics system itself will be necessary. There also exists a graphics editor through which a user can manually edit and display graphic items. These items can be stored in a "permanent graphic segment" for later pickup and use by a graphics program.

The New TTY DIM

As mentioned above, the Multics General Graphics System will drive the ARDS console through a new teletypewriter device interface module. This TTY DIM, which was in the final stages of checkout at the end of the reporting period, is of both abstract and practical interest. It is table-driven, the table basically representing a directed graph of operations and branches. The logic is sufficiently general to be able to accommodate a wide variety of character-oriented terminals. (Such terminals are becoming more and more numerous of late; the new TTY DIM will be able to deal with both the familiar types of hard-copy devices and the new soft-copy devices such as the storage-tube ARDS and the refresh-display computer IMLAC PDS-1.) The terminals may be connected over different types of channels, or according to different disciplines on the same channel. It is particularly interesting that the new TTY DIM is an instance of a table-driven device interface module that allows a device's characteristics to be specified in sufficient detail for it to be completely responsible for operating a terminal. The fundamental solution to terminal operation which it represents is felt to be relatively easily exportable to other machines; for, although some details of the table format and the interpreter program are specific to the GE 645 Generalized Input Output Controller, the bulk of both is general, and the GIOC-specific aspects could be replaced by more abstract representations. The interpreter program itself is written in PL/1 and should be usable elsewhere. Also under development is an unsophisticated compiler to facilitate the creation of tables for new devices.

IX. WORK IN PROGRESS AND FUTURE PLANS

Functional Capabilities and Performance Improvements

Multics projects in progress at the end of the reporting period include the following.

Standard Service System

The policy of causing all present system modules to adhere to Standard Service System criteria continues. In addition, the scope of the Standard

Service System is broadening with the planned introduction of such features as the commands of the Dartmouth System, a LISP interpreter and an APL interpreter. All Standard Service System programs are also being converted from the EPL to the PL/1 language. This not only offers the improved efficiency of the PL/1 object code, but allows halting of EPL maintenance.

File System

Also being converted to PL/1 is the File System. In conjunction with the recoding, certain design modifications are being effected as well: A new format is being introduced to allow for "small" directories (less space-consuming when the number of entries is low). The most frequently used paths through the File System are being optimized, which should result in significantly improved performance since some 90% of processing is performed by some 10% of the code. The binding of the various modules is being altered to minimize the number of page faults incurred by the paths chosen for optimizing.

Fault Interceptor Module

Many modules are being reworked to add new functional capabilities and to improve existing functions. Most basic of these modules is the fault interceptor module (FIM), which invokes the appropriate system routines in response to hardware-fault signals. Because of its central role in a process, the FIM has gained numerous responsibilities by accretion; therefore, the redesign is aimed at making the FIM more modular, to increase its speed for performing its basic fault-handling functions, and to clarify its other roles.

Other Modules

Other key modules being reworked are the interprocess communication facility (which is undergoing a thorough redesign to speed up its performance and extend its capabilities to include a secure, general message-passing mechanism), the Linker and the Gatekeeper. Still other important areas include the backup facility, management of the storage-device hierarchy ("multi-level storage" -- which is responsible for assigning less frequently used segments to slower-speed devices), the Binder, the User Control module (which manages logging in and out of the system), and the maintenance tools used in generating new Multics System tapes ("MSTs").

New Features

Two console-related additions to the system are the ability for a single process to drive multiple consoles, and a facility for on-line console-to-console communication. The supervisor is being modified to allow flexible specification by the user of which directories to search for a segment when a linkage fault occurs; this will be a more

general form of an interim facility introduced during the reporting period which allows the insertion of a single user-specified directory into the search path. Another supervisor change (referred to as "limit stops") will broaden the system's accounting facilities to allow for interruption of a process that has exhausted its allocated resources -- particularly with regard to machine time. Work is also being performed (by G. E. Cambridge Information Systems Laboratory personnel) on the APL language, which has become quite popular at some installations, and the embedding of the GE 600-series monitor GECOS under Multics.

Performance Improvements

Finally, continuing system-improvement tasks include the reduction of per-process segments, the continued propagation throughout the system of the object segment format, and the reduction of the amount of wired-down (non-paged) memory employed by the supervisor.

Thesis Research

Among the new features under development, three are related to thesis research. One of these, the ability to alter the hardware configuration while the system is running ("dynamic reconfiguration") is being implemented by a graduate student as part of his thesis; by the end of the reporting period, the ability to reconfigure processors had been demonstrated in a test system. The other two (a "save and resume" facility whereby a process may be restarted after an automatic logout resulting from a system crash, and an "absentee user" facility whereby a process can be created to perform a series of commands without console interaction) are being implemented by staff members who have benefited from the thesis research performed by graduate students.

Student Participation

Undergraduate students are playing an increasingly large role in Multics development. Summer projects being performed by undergraduates include the introduction of a major portion of the Dartmouth "SIMON" (simple monitor) system under Multics in a simulated environment, the development of a LISP interpreter, and the conversion of the Multics assembler from a combination of GECOS Fortran and GMAP (635 assembler) code to PL/1 and 645 assembly code, which is directly maintainable under Multics. Additionally, the Student Information Processing Board intends to expand and improve its SIPB-O subsystem in the Fall.

Future Plans

Aside from expanding efforts in the areas of graphics and the ARPA Network, two Multics-related areas are of particular long-range interest to the Group. The first of these is the topic of follow-on

hardware to the GE 645. At the request of the M.I.T. Information Processing Center, members of the Group are working in conjunction with G. E. Cambridge Information Systems Laboratory personnel on the specification of a design for hardware that will stand in approximately the same relationship to the new GE 655 as the 645 stands to the 635 -- that is, the new machine will employ the same technology as the 655 (some 2 to 3 times faster than the 635/645), with the addition of appending hardware and a few operation codes. A specification is being developed that requires only a modest redesign of the 655, and that is upward-compatible with the Multics software for the 645. Along with improved performance, the follow-on hardware should offer greater stability by virtue of being closer to the standard-product line 655 than is the 645 to the 635. By the end of the reporting period, early drafts of the specifications had been produced.

The second major long-range issue is that of the "exportability" of Multics. This is not taken literally to mean the running of Multics on a totally different machine. Rather, it is meant to imply the transferability of ideas -- in the sense of the promulgation of the conceptual bases of Multics, and of the lessons about the development of large systems learned in the course of the Group's work on Multics. The latter aspect is, of course, covered in the journal articles, reports, talks, papers and books about the system that have been or are being produced by members of the Group. In addition, work has begun on bringing both the system's internal (i.e., system-programmer-oriented) documentation and the system programs themselves up to publication quality. This attempt to describe the system lucidly, along with a like effort in regard to external (i.e., user-oriented) documentation should be a major means by which the ideas of Multics are propagated. Present plans are to publish in book form, in the near future, the Multics Programmers' Manual (MPM) and an examination of the system structure prepared by Prof. Elliot I. Organick. Subsequently, publication of the Multics System Programmers' Manual (MSPM) (which is currently undergoing heavy revision) is planned.

Technical Papers about Multics

Saltzer, J. H., and J. W. Gintell*, "The Instrumentation of Multics", ACM Second Symposium on Operating System Principles (October 20-22, 1969) Princeton University, pp. 167-174.

Spier, J. M., and E. L. Organick*, "The Multics Inter-Process Communication Facility", ACM Second Symposium on Operating System Principles (October 20-22, 1969) Princeton University, pp. 83-91.

Grochow, J. M., "Real-Time Graphic Display of Time-Sharing System Operating Characteristics", AFIPS Conf. Proc. 35 (1969 FJCC), AFIPS Press, 1969.

Saltzer, J. H., and J. F. Ossanna*, "Remote Terminal Character Stream Processing in Multics", AFIPS Conf. Proc. 36 (1970 SJCC) AFIPS Press, 1970, pp. 621-627.

Bensoussan*, A., C. T. Clingen*, and R. C. Daley, "The Multics Virtual Memory", Second ACM Symposium on Operating System Principles, Princeton, New Jersey, October, 1969.

M.I.T. Theses Related to Multics

Vogt, C. M., "Suspension of Processes in a Multiprocessing Computer System", M.S. Thesis, Dept. of Electrical Engineering, February 1970, also MAC TM-14, AD-713-989.

Frankston, R., "A Limited Service System on Multics", B.S. Thesis, Dept. of Electrical Engineering, June 1970.

* Non-MAC author

INTERACTIVE MANAGEMENT SYSTEMS

ORGANIZATIONAL DECISION MAKING

Prof. M. M. Jones

P. Brandler
S. Carney
D. R. Dawson
R. M. Elkin
R. S. Goldhor
R. C. Goldstein
E. T. Moore
S. E. Niles
L. K. Platzman
Prof. M. S. Scott-Morton
A. J. Strnad
D. M. Wells

Electrical Engineering Department

H. E. Brammer
Prof. J. D. Bruce
A. I. Fillat
L. A. Kraning

Civil Engineering Department

J. N. Jackson

Mechanical Engineering Department

C. E. Barringer

Project TIP

M. M. Kessler
W. D. Mathews

* * * * * *

SIMPLE PROJECT

Prof. M. M. Jones

R. M. Berman
C. R. Mehta
S. K. R. Murthy
R. C. Thurber, Jr.

INTERACTIVE MANAGEMENT SYSTEMS

ORGANIZATIONAL DECISION MAKING

I. INTRODUCTION

At the outset of this project in June 1969, two major lines of research were proposed. The first took as its starting point the computer facilities then available at M.I.T. and examined the ways in which these could be used to further general management objectives. Inextricably linked with this was our second objective of studying the decision-making process itself in order to discover how it could be improved through the utilization of a highly interactive computer system.

Our study took the form of a series of experiments conducted in several of the laboratories and academic departments within M.I.T. which were more or less similar to general administrative organizations found in government and industry. In each case, the administrative personnel concerned were given access to the Compatible Time-Sharing System (CTSS) computer. During the relatively long period of CTSS's existence, a large number of programs had been developed which offered promise of being useful for management functions. These ranged from simple text-editing and printing programs to elaborate facilities for sorting, plotting and retrieving information.

Without going into detail, the results of most of these experiments indicated that CTSS was not very effective as an interactive problem-solving and decision-making system for managers, for it appeared that the real problems faced by most administrators and managers are significantly more complex and require a more highly interactive environment than could be provided by the existing computer programs and systems.

Despite the disappointing conclusions, these initial experiments provided a store of very practical experience and knowledge essential to the development of useful decision-making tools -- information difficult to obtain by other means. Also, it should be emphasized, not all our results were negative. The two largest efforts, those within the Electrical Engineering Department and in Project MAC itself, led to systems that were of significant operational value and continue to be actively used.

II. E.E. DATA MANAGEMENT SYSTEM

The Electrical Engineering Department management system maintains information on each course offered by the Department each semester, and on each faculty member, course secretary, and graduate teaching assistant in the Department. This information is stored as one large file of plain text in CTSS, with special codes to identify the specific items in the file. The standard CTSS text editing commands (TYPSET and RUNOFF) are used to update the file. Special sorting and report-generating programs are used to prepare extensive hard-copy reports used within the Department.

This system was originally developed (with internal Electrical Engineering Department funds) to aid in solving the complex problem of assigning the very large E.E. teaching staff to the large numbers of courses taught each semester. Once that work was completed, the data base was expanded to include salaries, charges to research grants and contracts, and information on funded chairs. Then additional programs were written to perform other report-generation functions. Only the latter, more general work and the extensive documentation of the system have received Project MAC support.

Although the system is actively used, it is quite primitive in many respects. For example, it employs a very simple, sequential data organization, does no input-data validation, and is oriented more toward report preparation than interactive decision-making. Despite these drawbacks, the system serves a very useful function and is now an integral part of the Electrical Engineering Department operations. In addition, it has attracted widespread interest in the Chemical and Civil Engineering Departments of M.I.T. as well as by the M.I.T. Comptroller's Office. The enthusiasm with which this information-management system has been received by these academic and administrative departments at M.I.T. is a strong indication of the seriousness of the need for such management tools both here at the Institute and elsewhere. The E.E. system building demonstrated once again the value of one consistent central data base that can be used for a variety of purposes.

III. MACAIMS

The work at Project MAC on MacAIMS (Advanced Interactive Management System) was designed more to extend the state of the art in information management than to solve a specific management problem. However, throughout MacAIMS development, the problems involved in managing Project MAC itself and similar research organizations -- both government and industry -- were kept clearly in mind.

Several facilities of the MacAIMS system are currently in use by MAC Headquarters. In contrast to the system used by the Electrical Engineering Department, MacAIMS employs a rather complex internal data organization designed to facilitate interactive retrieval of information. It also has a much more sophisticated user interface, including a considerable degree of input-data validation. Function-oriented programs written so far are in the areas of personnel-data management, budgeting, equipment inventory, and purchasing.

The personnel management system is the most extensive of the four functional applications and has been most fully developed. In addition to storing the standard information on name, address, telephone number, age, salary, etc., we have found it important to associate effective dates with a number of these fields and thereby be able to store non-

current information as well as changes that are anticipated but not yet formally effected. Thus, the system is useful for showing comparisons over time. The personnel data-base structure itself is hierarchical and directly reflects the structure of most organizational entities. For example, each Project MAC person is associated with one or more research groups, and each group is supported by one contract. For every research group there is one group leader.

Careful attention has been paid to the issues of privacy of information and of access control. Thus, each individual may inspect the entry that pertains to him, and each group leader may access information about the people in his group but not in other groups. The Headquarters staff may have access to all information in the file.

The budgeting system developed within MacAIMS effectively demonstrates the inadequacies of the CTSS environment for most real management problem-solving applications. Project MAC has about 30 distinct research groupings (when separate contracts and all other pertinent factors are considered), for each of which it is necessary to store about 25 discrete pieces of financial information. Furthermore, in order to provide adequate space for both the past and the future, it is necessary to store about two years (24 months) of figures: $30 \times 25 \times 24$, or 18,000 individual items of information. Since this information is highly interrelated by pointers which link associated items, it is necessary that the entire data base be stored in core memory for fast retrieval. By the time the structured information and the programs themselves are added, the total space required easily overflows the available core memory of the CTSS system. Faced with this dilemma, we decided to work with only a subset of the complete organization structure and to continue system design in order to gain some experience with interactive budgeting systems.

The purchasing subsystem of MacAIMS is designed to help with the general procurement problem. As currently implemented within Project MAC, it assists with preparation of standard M.I.T. purchase orders. Also it maintains an on-line purchase journal which may be searched to determine the status of any individual purchase order, the current outstanding commitments, the total orders given to any vendor, and the total expenditures by any research group, etc.

The equipment-inventory programs maintain records on typewriters, dictating machines, and data-communication equipment, and permit interactive retrieval of specific information as well as preparation of standard reports. Full historical data on utilization and charges is also maintained.

In addition to these specific functional programs, much of the initial MacAIMS development effort was devoted to building a suitable general

data-management environment. Once this environment was available, the implementation of any specific function was a relatively straightforward task.

The substantial investment made in developing this general environment has paid off. Initially, functional programs were written to perform the same operations that previously had been performed manually. Once these programs were available, the administrative personnel began to request additional capabilities. They also came to MacAIMS with unusual one-time information requests, e.g., the annual survey on data-processing employment or the annual Project MAC audit. Because the basic MacAIMS environment had been built with great care, the most complex additional capability was implemented with only a few days of part-time work by a student programmer. Frequently, the one-time, special requests could be handled in minutes. Thus, we feel that we have demonstrated the general utility of the MacAIMS system, at least in part.

During this year we also began the movement of the entire MacAIMS system from CTSS to Multics, including a substantial redesign in order to take advantage both of the unique capabilities of Multics and the experience gained through use of the original system. By the end of June 1970, coding had begun on most of the key modules of the new system with the expectation that initial applications tests could be run in the Fall of 1970.

In summary, the problem of developing a suitable general-purpose, man-computer, decision-making environment is considerably more complex than might at first be thought. However, as a result of our work, we now believe that we know how to solve most of these problems and look forward in the coming year to developing a system that will be of significant practical as well as theoretical importance in a wide variety of application settings.

IV. COMPUTERIZED BUDGET CONTROL

The system initially proposed for computerized budget control in the Mechanical Engineering Department had earlier been developed in an M.I.T. interdisciplinary research laboratory, but when the system was transposed to an academic department, a wholly different set of problems had to be solved, e.g., the various contract budgets were unrelated in terms of sponsor, initiation and termination dates. Moreover, as was discovered after several months' experience, the time-lag in notifying the principal investigators proved too great in terms of timely budget decisions. Nevertheless, because of the lessons that had been learned from the experience, the system was fully documented in an internal report ("Computerized Budget Generation and Expenditure Control System", by C. E. Barringer and K. R. Crossen, June 1970).

Subsequent work in this period included efforts to draw together a basic personnel file data bank that could be used for various departmental purposes.

V. PROJECT MANAGEMENT IN BUILDING DESIGN

This effort was part of the doctoral thesis research in computer-assisted building design by James Jackson in the Department of Civil Engineering. The over-all purpose was to develop means for integrating the various computer systems used in such designs. Among the goals of the research was the devising of a data structure for a project file and a scheme for communication among the multiplicity of computer systems involved in a building project.

The work concluded with the completion of the Ph.D. thesis in June 1970 ("Building Data Management System").

VI. PROJECT TIP

During the reporting period, several activities in the Technical Information Program (TIP) received support from Project MAC, including the following:

1) Improvement of the TIP System's capability as an administrative information-handling facility;

2) Devising means to enhance the formatting ability of the TIP retrieval subsystem;

3) Creation of demonstration data bases.

Details of work in all these areas are reported by TIP in the various reports issued by that Project.

Thesis

Fillat*, Andrew I., and Leslie A. Kraning, "Generalized Organization of Large Data Bases", Bachelor of Science and Master of Science, Department of Electrical Engineering; also MAC TR-70, June 1970, AD 711-060.

---

* Non-MAC author.

SIMPLE PROJECT

From June 1969 until January 1970, the SIMPLE group continued the experimental implementation of the SIMPLE Simulation System on the IBM 1130 computer. (Progress Report V describes the origin of the program.) In the middle of January, however, we decided to terminate the effort to implement the system on the 1130. It appeared at that time that the additional time and effort needed to complete that implementation could be better spent beginning the implementation of SIMPLE on the Multics system now that the latter was available. SIMPLE differs from existing simulation languages in that it was designed for use on a system permitting both time-sharing and the operation of a graphical-display device.

Since January, a preliminary SIMPLE system for Multics has been designed, and implementation is proceeding on schedule. As of August 1970, a preliminary system will be operational. This preliminary system consists of a translator which translates a SIMPLE program into a PL/1 program acceptable to the Multics PL/1 compiler. We decided to write a translator rather than to try to modify the Multics PL/1 because a translator could be written comparatively quickly and simply, and still provide the user with all the features of the SIMPLE language. Also, the PL/1 compiler is still being constantly changed and updated.

Currently, we are testing and upgrading this preliminary translator. We plan to use the system during a Sloan School Summer Session Seminar on simulation techniques at the end of August. The participants in the Seminar will use the SIMPLE language to construct, debug and execute simulation models. Thus, the Seminar should provide a good "first test" of the new system.

The SIMPLE language, as currently implemented, is described in the "SIMPLE User's Manual", an internal document almost completed which will be distributed to participants in the Sloan School Seminar and subsequently will be distributed more generally.

Once this initial version of the SIMPLE system is thoroughly debugged, we plan to spend the Fall and Winter adding additional features to the system such as more statistical routines, more tracing and debugging facilities, better error diagnostics, and possibly a graphical-display language. Also, we expect to complete the design and start the implementation of an interpreter to replace the present translator. When this interpreter is available, all the design goals for incremented simulation will be satisfied.

PROGRAMMING LINGUISTICS/EXTENSIBLE LANGUAGES

PROGRAMMING LINGUISTICS

Prof. R. M. Graham

D. D. Clark
J. D. DeTreville
R. S. Eanes
J. F. Haverty
B. P. Lester
P. S. Malek
P. L. Miller
S. R. Murphy
J. I. Seiferas
H. J. Siegel
J. E. Sussman
C. A. Vogt
M. W. Webber
S. N. Zilles

EXTENSIBLE LANGUAGES

Prof. A. Evans, Jr.

W. D. Bilofsky
A. J. Davidoff
M. W. Dickens
D. A. Henderson, Jr.
E. C. Horvath
B. D. Hubbard
P. M. Ledoux, Jr.
R. F. Mabee
J. R. Nestor
J. L. Piggins
J. E. Piñella
L. I. Reich
B. Rosenbaum
R. H. Thomas

PROGRAMMING LINGUISTICS/EXTENSIBLE LANGUAGES

INTRODUCTION

The main theme of the research of the Programming Linguistics/Extensible Languages (PL/EL) Group continues to involve improving our understanding of the basic concepts in programming languages. Our work centers about language formalization and language extension. We continue the close relationship between teaching this material to undergraduates and developing fresh approaches to it.

Motivation and Background

The function of a programming language is to serve as a set of conventions for communicating algorithms -- the communication being either between people and people or between people and machines. The efficiency of the communication process is clearly improved as the conventions are better understood. This is just another way of saying that it is advantageous that the programming languages we use be accurately defined. Conventionally, programming languages have been defined by English language descriptions, as written in manuals. The modern idea in this area is to formalize the definition, using some suitable notation. The discovery of what sort of notation is "suitable" is a major unsolved research problem in this area, a problem that we have been attacking.

The pay-offs for success include rather obvious ones such as enhancing the process of teaching new languages and providing standards by which to judge compiler performance. However, there are other advantages. One application has to do with proof of correctness of algorithms. A programmer producing a program which he claims to be a solution to a problem should do more than just show that it works on one or two selected data sets: He should also be able to prove that the algorithm is in fact correct. Even more, he should prove that his implementation of the algorithm is a correct one. Doing the latter requires that he make statements about the program he has written, and such statements can take on mathematical significance only if the language in which the algorithm is expressed has been formalized. For example, as part of such a proof, one might make an assertion like: "The effect of obeying this statement is thus-and-so". Since such an assertion can be supported only by appeal to the definition of the programming language, formalizing that definition is a prerequisite for making any formal proof.

Another pay-off from language formalization, one that up to now has received much attention in the PL/EL Group, concerns language extension. A user of an extensible language facility is provided with a base language and with tools so that he may build on to that base the features that he needs. Communications from the user to the facility are

of the form, "Please understand that when I say thus-and-so I mean thus-and-so". Since the user's activity is specifying to the system the semantics that he wants his proposed constructs to have, what he needs is a notation for expressing semantics. Clearly, then, any progress made in the area of language formalization will have an obvious pay-off in an extension facility.

The word "semantics" which we have been using is a word that means different things to different people. Let us discuss it briefly. The usual dichotomy in programming-language definition has to dc with syntax and semantics. Roughly speaking, the former relates to the legal utterances in the language in question, and the latter to the effect of such an utterance when the program of which it is a part is executed. We do not concern ourselves with what some refer to as pragmatics, which we term user interpretation, since it has to do with how the user understands these effects. For example, the assignment

$$x := x + 1$$

might be a legal utterance (i.e., syntactically correct) in some language. Its semantics involves something like determining the current value of the variable x, adding one to it (if the value is a number) and storing the result back into x. The user interpretation might be, "Tally one more apple", or "Go to the next row", or "Step to the next case".

This discussion should be kept in mind by the reader when we refer to "formalization of semantics", since it implies certain limitations on our current goals. For example, suppose the expression

$$A + P$$

appears in a program. If both A and P denote integers, our semantics definition scheme will without difficulty ascribe semantics to this phrase. But the user interpretation might be that A counts apples and P counts pears, and that such a sum is meaningless. While a language processor that assists the user by taking cognizance of such ideas is clearly of interest, our current research has not been in that direction.

Long-Term Activities

Two activities continue to occupy our attention on a long-term basis: teaching the undergraduate subject 6 231, "Programming Linguistics", and support of the languages PAL and BCPL. Development of the 6.231 material and the research of the PL/EL Group have long gone hand-in-hand. In 6.231, the basic concepts in programming languages are taught largely by exhibiting a formalization of the semantics of PAL, a language that we have devised and whose sole purpose is to be taught. Although PAL, by design, is susceptible to a straightforward formal definition, the creation and polishing of that definition have provided both insight into the problems and valuable suggestions for ways to

proceed. PAL is implemented on several computers, the Multics implementation being used currently by our students.

BCPL has been an interest of the group since it was devised by Martin Richards when he was at Project MAC in 1967 and 1968. The language was designed to be useful for compiler writing, and must be judged a success. Since the PAL implementation is in BCPL, and since use of BCPL is fundamental to the LPS project of Prof. Graham, maintenance and improvement continue to be important to us.

REPORT ON PROGRESS

A major part of the effort of the group has gone into teaching and improving the subject "Programming Linguistics". We continue to derive a large pay-off from the feedback from the teaching activity. The PAL formalization is much more complete than it has been in the past, and the documentation of it is almost entirely finished. Currently, this documentation is in the form of notes to be used by students, and an important job for the next year is to produce a more concise documentation of the formalization. The entire formalization is rather long, and it is not clear what publishing route should be followed.

Doctoral Research

Robert H. Thomas has been concerned with one particular aspect of language extension; he has been developing a model in terms of which the user of an extension facility can specify the semantics of his constructs. The model involves a conceptual mechanism whose characteristics are derived (although are rather different) from Landin's SECD machine which has so influenced the PAL development. Many features that are built into the SECD machine are programmable in Thomas's machine, thus providing a very desirable form of generality. For example, the binding of parameters in function application is under the control of the programmer rather than being built-in. This promising research will lead to a better understanding of an underlying model for a language-extension facility.

D. Austin Henderson, Jr. has been pursuing doctoral research on the problem of transduction of graphical input -- a problem related to but not directly in the mainstream of the rest of the group. The user of any computing facility, whether it be for engineering design, language extension, numerical analysis, or other purpose, must communicate to the facility what he has in mind. Conventionally, such communication has involved linear strings of characters, as for example in the text of a program. All available experience with graphical communication shows that there are problem areas for which linear text is inappropriate, at least when there is an alternative. Important developments such as Sketchpad have been almost exclusively a matter of making available to the user the ability to submit structured graphical data to

a computer through a (usually interactive) graphics interface. Most existing application programs in this area have used rather ad hoc methods for deducing from the graphical information what the user intended: The developments of lexical and syntactic analysis (which have so strongly influenced the programming-language game during the last few years) have received little attention in this area. Henderson's research has been concerned with developing techniques for the analysis of scenes. An analogy, which is not exact but which nonetheless may be useful, is the following: Henderson's work is to the earlier ad hoc schemes as compiler techniques such as precedence analysis and LR(k) recognizers are to the crude schemes of the late 1950's. This project has by now proceeded sufficiently far that useful results are anticipated.

BCPL Development

As mentioned earlier, the PL/EL group continues to be concerned with the BCPL language and its compilers on Multics and other machines. Both Prof. Evans and Robert F. Mabee have been active in a BCPL Users Group, consisting of people from different installations in the country who are interested in BCPL. We have had two meetings at Lincoln Laboratory at which various language topics have been discussed. We have produced at Project MAC a computerized version of the BCPL Reference Manual, and the Users Group will consider revising this to reflect proposed language changes. Of course, no one can legislate these changes, but it seems likely that many of them will be implemented.

Locally, Mabee has been concerned with rewriting the BCPL compiler on Multics. All the programming done in the group is in BCPL, and much of it would be improved significantly by a better compiler. We hope to make the compiler run faster, and also to produce better object code. The design is almost complete and much of the coding is done. The code has been partially debugged.

As a separate activity, we have worked to make BCPL available outside M.I.T. The most exportable version of the language is that on the IBM 360, and we have provided tapes of the compiler for some 16 installations. We continue to regard a certain amount of proselytizing as within our charter.

PAL Development

The PAL implementation has been little improved during the past year. Any improvements in BCPL will, of course, make all PAL programs run faster. During this Summer, we hope to add some significant improvements to the PAL run-time system. This system is virtually identical to what was written three years ago on CTSS. There is room for improvement.

Starting last Fall, we have worked with 80 to 100 students a semester on Multics using PAL. For this purpose, we have written a subsystem that runs under Multics and which permits our students to use only PAL and an editor, along with certain useful utilities. Continual improvements have been made in this operating system, and we expect to make more during the Summer.

Research by Undergraduates

A senior thesis has been written by Edward C. Horvath on "APL on Multics: Lexical and Syntactic Considerations". Horvath worked with some of the Multics design team who were concerned with APL, and he has documented certain aspects of the design produced.

Development of code-optimization strategies for the BCPL compiler has been a senior thesis project of Paul Ledoux who expects to complete the work during this coming Summer.

Several students associated with the PL/EL Group on a project basis have made contributions. Judith L. Piggins has been concerned with improving the PAL programming which is part of the PAL formalization. A major conceptual change was made in these programs, and she has been developing the new algorithms on the computer. The work is substantially done and has already become a part of the course notes.

Michael W. Dickens has continued to work on the PAL compiler and run-time system on Multics. Dickens's work has been a matter of polishing the interface between the language and the operating system.

Louis Reich and Alan Davidoff have been concerned with improving the PAL run-time system. Most of their effort has gone into devising a proposal for an improved string-handling package. One of the problems of improving the efficiency of any program is that it is frequently difficult to measure exactly how much improvement has resulted. Since there is a PAL implementation on the TX-2 computer at Lincoln Laboratory, and since the TX-2 has sophisticated hardware and software tools for measuring the performance of existing programs, we took advantage of the opportunity to make some changes to TX-2's PAL and to examine the improvement. The results were gratifying, and have shown where to change the Multics implementation. Hopefully, the two students will be available next Fall to continue this work. (If not, an attempt will be made to find other students to do this.) We hope that measuring tools on Multics will soon become adequate to do such work here, but, if not, we shall continue to make use of the facilities at Lincoln Laboratory.

John Nestor last Fall worked on improvements in the BCPL library; he produced a general clean-up of a previously rather chaotic situation, and supplied documentation of the results.

Bruce Hubbard has made significant contributions to the machine-language library for BCPL. During this coming Summer, he expects to write those parts of the library for the new BCPL that must be coded in machine language.

Publication 1969-1970

Horvath, E. C., APL on Multics: Lexical and Syntactic Considerations, B.S. Thesis, Dept. of Electrical Engineering, June 1970.

PROGRAMMING LANGUAGES

Prof. J. J. Donovan

H. Adler
A. Bagchi
P. G. W. Bras
C. A. Dancy
R. J. Fleischer
J. W. Johnson
C. A. Kessel
W. J. Klos, Jr.
N. V. Kohn
J. C. Lind
S. E. Madnick
R. Mandl
E. Nangle
R. Petrivalle
C. Ramchandani
J. L. Reuss
H-M. D. Toong
L. E. Travis

PROGRAMMING LANGUAGES

INTRODUCTION

The focus of all research and teaching activities of the group is programming languages. The research has ranged through theoretical models of programming languages, implementation of compilers, to implementation of operating systems supporting these languages. The theoretical work has been aimed at formalizing the relationship of canonic systems to other formal grammars, using canonic systems as a basis for a generalized compiler and for the development of quantitative measures for programming languages.

The theoretical work uses a formal system called canonic systems that was partially developed and enhanced here at M.I.T. A canonic system is a simultaneous recursive definition of several sets of strings over a finite alphabet. Canonic systems have been used to completely specify a programming language and its translation (as reported in 1969 Progress Report, the work of Ledgard). They include many of the so-called "context sensitive" features of a programming language. A canonic system's specification and its translation of a language has been used to derive a generalized translator (as reported in 1969 Progress Report, the work of Alsop). Complexity measures for canonic system specifications have been studied.

RESEARCH ACTIVITIES

Relation of Canonic Systems to Other Formal Systems

We have proven a number of theorems relating canonic systems to Post systems, Smullyan's elementary formal systems, and Chomsky's hierarchies of grammars, the major theorem being that the class of type i grammars is strongly equivalent to the class of type i canonic systems. i equals 0,1,2,3. The class of linear, one-sided linear, meta-linear, sequential, etc., grammars are strongly equivalent, respectively, to the classes of linear, one-sided linear, meta-linear, sequential, etc., canonic systems. The practical motivation for proving such a theorem is to assure that a generalized compiler using a canonic system as a data base must halt. Therefore, we must limit the power of a canonic system only to describe decideable sets. In classical linguistics theory, it is known which grammars describe decideable sets and which do not. Therefore, we are motivated to find some of the classes of canonic systems that describe decideable sets and use this restriction on canonic systems as input to our generalized compilers.

Power of Canonic Systems

Canonic systems, which were first defined to meet the definitional needs of programming languages, were felt to be too powerful since they could generate nonrecursive sets. It is felt that a restriction should

be placed on canonic systems to render them unable to specify nonrecursive sets yet powerful enough to specify computer programming languages. The search for such a restriction is the motivation for the theorems reported above relating canonic systems to existing formal grammars. Yet we must find where programming languages are in the hierarchies. We have proven that the set of legal programs of PL/1 is nonrecursive. Thus to specify PL/1 we do need the full power of canonic systems.

Measures of Complexity

We have developed several simple measures of complexity using canonic systems that are proportional to the resources (e.g., accesses to memory, computer time) used by computers in translating languages specified by canonic systems. One of these measures has turned out to be proportional to the time of translating a program using a generalized translator that has been implemented on CTSS (reported in 1968 Progress Report, done by Alsop).

Using a canonic system C, which describes a set of strings, it is possible to generate a system C_n, called a proof measure function, that has two arguments, the canonic system C and a string T that can be generated by that canonic system. We have defined these proof measure functions so that they are proportional to the length of the derivation of a string T within C or proportional to the number of predicates used in evaluating and producing the string T in the caninic system C. Both of these are an indication of the complexity of the number of resources a computer may use in generating these strings. For certain classes of canonic systems, algebraic bounds upon these functions can be derived from the structure of the system. A practical computer programmer is not interested in how long or how many resources it takes to generate a string. He is interested in how long it takes to translate a string. We have produced another transformation on C that produces a system C^{-1} that characterizes the recognition of strings generated by C. We have proven theorems relating the major functions of C and of C^{-1}, thus relating the complexity of the recognition procedure to that of the language description.

Canonic System Translator

We have implemented a program on Multics that accepts canonic system descriptions of a language and produces Floyd's reductions. We were motivated to write this program because a method of producing a computer is to have the syntax phase of a compiler driven by reductions, and a common form of these reductions is Floyd reductions. Honeywell Corporation has extended this work to produce an entire compiler generating system.

Programming System Environment

As the distinction between the compiler, the operating system, and the source code becomes less and less distinct in modern computer systems, we find investigations in programming languages becoming more involved with operating systems. For example, storage assignment and allocation of resources are all handled by the operating system, yet the compiler and the compiler object code must interface with the operating system and the file system in which it finds itself. Therefore, we have within our group devoted some research to the area of file systems, since we feel this is the heart of an operating system. We have developed a systematic approach for the design and study of file systems. This work is analogous of Dijkstra's development of a systematic approach for the design of an operating system. He developed a way of looking at an operating system in a modular approach. Our work on file systems has been to develop a modular approach to the design and study of file systems. This work was conducted by Stuart Madnick. In his work he has developed seven modules, each of which is independent of the others except through well-defined calls.

Publications 1969-1970

Dancy, Charles, A Cobol Compiler for the IBM 1130, S.M. Thesis, Dept. of Electrical Engineering, January 1970.

Johnson, Jerry, File System to Support Time Sharing in a Multiprogramming Environment, M.S. Thesis, Dept. of Electrical Engineering, June 1970.

Mandl, Robert, Further Results on Hierarchies of Canonic Systems, M.S. Thesis, Dept. of Electrical Engineering, September 1969.

Ramchandani, Chander, Debugging System to Run Interpretively in Virtual Memory, S.M. Thesis, Dept. of Electrical Engineering, January 1970.

Madnick, S. E., "MIS -- Problems Plus a Solution", Computer Forum Report, Vol. 1, No. 4, July 1969.

AUTOMATA THEORY

Prof. F. C. Hennie

M. E. Baker
V. M. Berardinelli
G. G. Bruere-Dawson
M. Edelberg
M. M. Hammer
D. J. Kfoury
Prof. C. L. Liu
Prof. A. Meyer
R. N. Moll
B. Ong
B. J. Vilfan
C. Ying

AUTOMATA THEORY

Research in the Automata Theory Group has been fairly eclectic this year, reflecting diverse interests of new members of the group. Although certain familiar themes in the theory of computation remain evident in this work -- e.g., the interrelations between the structure and behavior of abstractly characterized computing devices -- we shall not attempt to synthesize further common themes in the assortment of theoretical problems now under study. Instead we summarize briefly below the main results obtained by members of the group during the period covered by this report.

Abstract Complexity Theory

Abstract complexity theory is concerned with the consequences of classifying computations by the amount of computing resources, such as time or space, required for their execution. The results are abstract in that the computing resource requirements of particular interesting computations (for example, computing the product of two numbers) are not considered. Clearly, the ultimate justification of the theory must come from the insight it helps provide about real computations; but we cannot expect to fully understand the particular behavior of a problem like integer multiplication until we have some notion of the behavior of computations in general. One would like to discover the most efficient method for multiplying integers. Abstract complexity theory, specifically the Speed-up Theorem of Blum, points out that there cannot be any most efficient method for computing certain functions. Thus the apparently practical problem of optimizing multiplication algorithms may be impossible to solve because there is no optimal solution. Abstract complexity theory at least enables us to recognize this possibility.

A monograph summarizing the development of abstract complexity theory in the past decade is now being prepared by Prof. Meyer. As an illustration of the nature of this area, we shall discuss a theorem due to Meyer, jointly with Prof. M. J. Fischer, which was presented at the Logic Symposium of the University of Manchester, August 1969.

Consider programs for deciding predicates on the integers. A program decides a predicate P if, started with any integer x as input, the program eventually prints out the truth value of P(x) and halts. The complexity of P is measured by the amount of time or memory space which programs deciding P require. It is intuitively clear that predicates may be very complex, and moreover they may be complex for different reasons. The latter concept is frustrating mathematically, because no one has yet characterized what might be the "reasons" why a predicate is computationally complex. (Proofs that predicates are complex are invariably diagonal arguments of recursive function theory,

with the result that predicates are known to be complex only because they differ from all predicates that are not complex. This is not a very satisfactory "reason" for their complexity.)

Suppose that one has two predicates P_1, P_2 which are complex. Suppose further that even if one had the ability to look up truth values of P_1 in a table, or equivalently if one could evaluate the truth value of $P_1(x)$ in one step for each integer x, it remained just as hard to decide P_2 as it was without the table. This state of affairs could be interpreted as meaning that P_2 was complex for different reasons than P_1.

Definition. Let P_1 be a predicate on the integers which can be decided by some program. Let P_2 be another predicate such that for any program which decides P_1 and which has the ability to evaluate $P_2(x)$ in one step for each integer x, there is another program deciding P_1 which runs just as fast on all inputs and does not have the ability to evaluate P_2. Then P_2 is said not to help P_1.

Theorem. For any computable function t(x), there exist predicates P_1, P_2 which can be decided by programs such that

1) any program deciding P_1 (or P_2) requires t(x) steps for its computation on input x for all sufficiently large integers x, and

2) P_1 and P_2 do not help each other.

Active research in this area is now directed at two issues: the structure imposed on computable functions by a complexity classification, and the relation between size of programs and complexity.

Random and Pseudo-Random Sequences

The statement that a particular infinite binary sequence $\alpha = \alpha_0, \alpha_1, \ldots$ is "random" is meaningless from the point of view of classical probability theory. An effort to formalize the idea of a particular sequence's being random dates back half a century to Von Mises, and new approaches have recently been proposed by Kolmogorov and Martin-Lof. These three notions of randomness are described informally below.

A particular infinite sequence is random in Von Mises's sense if it is unpredictable. If one were gambling, using tosses of a fair coin, for example, and the sequence described the outcomes of successive tosses, then predictions of portions of the sequence not yet observed should be wrong half the time. Formally, a prediction method is a computer program which, given the first n digits of a sequence, will print out a prediction of the $n+1^{st}$ digit. A sequence is Von Mises random if for any computer program the fraction of the number of correct guesses among the first n guesses goes to the limit 1/2.

Martin-Lof's definition can be motivated by a bit of word play. Consider some property satisfied by almost all real numbers (a property of measure one). If one selects a real number from the unit interval at random, it will, with probability one, have the property. Hence a random real number should satisfy any property of measure one. This tentative definition turns out to be absurd because no number has all properties of measure one; but Martin-Lof shows that, if one restricts attention to measure one properties which are constructive in a suitable sense, then random real numbers exist. Random real numbers can be equated with their binary expansions to obtain random binary sequences.

The third definition due to Kolmogorov is based on the idea that a random sequence is one without a recurring pattern. The simplest precise description of a patternless sequence is simply a copy of the sequence. Thus a finite binary sequence is called random if the smallest program that prints out the sequence has essentially just as many bits as the sequence itself. An infinite sequence is Kolmogorov random if its finite prefixes are random.

Gerard Bruere-Dawson has investigated the relations among these definitions. He has proven that Von Mises's definition yields a strictly larger class of random sequences than does Martin-Lof's definition. Also, Martin-Lof's class of random sequences is at least as large as Kolmogorov's, but whether it is strictly larger is still an open question.

The natural goal of this work is to provide a theoretical framework for dealing with random number generators in computing. Any program that serves as a random number generator yields a nonrandom sequence by the definitions above, precisely because the sequence is generated by a program. However one can specialize the definitions of Von Mises, Martin-Lof and Kolmogorov so that mention of programs in these definitions is replaced by mention of fast or storage-efficient programs. Bruere-Dawson shows that there are computable sequences whose x^{th} digit can be generated in t(x) steps but which are Von Mises random with respect to all programs that run in time a little less than t, for all recursive functions t. This result is a small step in the direction of developing methods for constructing pseudo-random number generators which are known a priori to satisfy all computationally simple statistical tests for randomness.

Perceptrons

Two variant models of perceptrons have been considered by Bostjan Vilfan. As defined by Minsky and Papert, perceptrons represent one example of a device in which computations on "local" versus "global" information can be informally distinguished. Since connectedness is a paradigmatic global property, it should be the case that other perceptron-like models in which global calculations are restricted share the

inability of Minsky-Papert perceptrons to recognize connectivity. This result was verified, but required proofs that are quite different from those of Minsky and Papert.

Integer Programming

A system S of linear inequalities describes a convex polyhedron P in n-dimensional space. Embedded in P is another convex polyhedron P' which is the convex hull of the set of integer points contained in P. The relationship between the system S and the polyhedron P' is important to the solution of many combinatorial optimization problems. Murray Edelberg has investigated the following problem: Given S, find a system S' of linear inequalities which describes the integer polyhedron P' directly. He has developed a method for transforming S into S' for systems S of two-variable inequalities. Essentially, this method "rounds corners" of the polygon P by means of an integer division process based on a generalization of the familiar division theorem for integers. The properties of higher-dimensional integer polyhedra have also been studied, and the conditions under which a system S' describes an integer polyhedron P' have been determined.

Algorithms on Graphs

Professors Meyer and Fischer observed that Strassen's fast matrix multiplication algorithm can be applied to find the transitive closure of an n-node directed graph in $0(n^{2.9})$ steps, a considerable improvement over the best previously known algorithms which required $0(n^3)$ steps. The graph theoretic interpretation of Strassen's method is now being studied in the hope that still better algorithms can be discovered.

Complexity of Boolean Functions

Circuit diagrams and functional expressions are reasonably natural formalisms with which to describe Boolean functions. The complexity of a Boolean function is reflected by the size of a minimal Boolean expression for it, and the number of gates in, or the depth of, an optimal circuit for it. Asymptotic arguments imply that the majority of Boolean functions of n variables have size exponential in n, but thus far no particular function has been proved to have size exceeding n^2.

A summary of the few published arguments which enable one to accurately estimate these parameters for certain functions is now being prepared by Bostjan Vilfan. A class of functions whose size is conjectured to exceed any polynomial is being studied.

Algebraic Coding Theory

Work on algebraic coding theory was continued this year by Boon Ong and Gregory Ruth, under the supervision of Prof. C. L. Liu. This work is directed toward understanding the algebraic structure of certain

classes of codes and the construction of efficient error detection and correction codes.

Boon Ong devised a new scheme for constructing certain linear and nonlinear codes. The basic idea is to concatenate words from two given codes to yield a longer code with certain distance properties. This scheme has been used to construct a large class of linear codes, including the Hamming codes and the Golay (23,12) code, and a large class of nonlinear codes, including the Nordstrom-Robinson (15,8) optimal code and Preparata's $(2^n - 1, 2^n - 2n)$ optimal codes (n even). It is believed that this approach will not only aid in understanding the structure and properties of many known codes, but also lead to the discovery of new and useful codes.

Regular Languages

In their study of the counter-free languages, a subfamily of the regular languages, McNaughton and Papert introduce a binary operator "box" which preserves regularity and the counter-free property. They raised the question whether "box" was independent of the other familiar language operators of concatenation, union, complementation and star. The question was settled affirmatively by Fischer, Meyer, O'Neil and Paterson in a note which appeared in SICACT News (Dec. 1969). A stronger version of this result pertaining to language derivatives was subsequently obtained by Meyer and S. Ahy.

Probabilistic Automata

A synchronizing sequence for a probabilistic automaton A is a sequence that is guaranteed to leave A in a particular final state probability distribution, regardless of the initial state probability distribution. The automaton A is said to be synchronizable of order m if it has a synchronizing sequence of length m but no synchronizing sequence of length less than m. Denis Kfoury has shown that for each integer $n \geq 3$ and each positive integer m, there exists at least one n-state probabilistic automaton that is synchronizable of order m. However, for $n \geq 4$, Kfoury shows the problem of determining whether an arbitrary n-state probabilistic automaton is synchronizable to be recursively undecidable.

Publications 1969-1970

1. Bruere-Dawson, Gerard. "Pseudo-Random Sequences", M.S. Thesis, M.I.T., Dept. of Electrical Engineering, June 1970.

2. Edelberg, Murray. "Integral Convex Polyhedra and an Approach to Integralization", Sc.D. Thesis, M.I.T., Dept. of Electrical Engineering, August 1970.

3. Fischer, M., Meyer, A., O'Neil, P. and Paterson, M., "A Note on Independence of a Regularity-Preserving Operator", SICACT NEWS, Assoc. for Comp. Machinery, January 1970.

4. Kfoury, Denis. "Synchronizing Sequences for Probabilistic Automata", Studies in Applied Math. XLIX, No. 1, March 1970.

5. Meyer, A. and Fischer, M., "Relatively Complex Recursive Sets", to appear, Proceedings of the Logic Conference, Univ. of Manchester, August 1969.

6. Ong, Boon. "A New Construction Scheme for Linear and Non-Linear Codes", M.S. Thesis, M.I.T., Dept. of Electrical Engineering, June 1970.

7. Vilfan, Bostjan. "A Note on Cyclic Perceptions", Proc. of the Fourth Annual Princeton Symposium on Information Sciences and Systems, October 1969.

References

1. Blum, M., A Machine-Independent Theory of the Complexity of Recursive Functions, Journ. Assoc. Comp. Mach., 14, 322-336 (1967).

2. Kolmogorov, A., Three Approaches to the Quantitative Definition of Information, Int. Journ. of Comp. Math., 2, 157-168 (1968).

3. Martin-Lof, P. The Definition of Random Sequences, Inf. and Cont., 9, 602-619 (1969).

4. Minsky, M. and Papert, S., Perceptrons, M.I.T. Press, Cambridge, Mass., (1969).

5. McNaughton, R. and Papert, S., Counter-Free Automata, M.I.T. Press, to appear, 1971.

6. Strassen, V., Gaussian Elimination is Not Optimal, Num. Math., 13, 354-356 (1969).

7. Von Mises, R., Probability, Statistics, and Truth (trans. from German), Macmillan, New York, 1957.

MATHLAB

Prof. W. A. Martin
Prof. J. Moses

R. J. Fateman
S. Feldman
J. P. Golden
D. C. Hill
P. Loewe
S. Saunders
R. C. Schroeppel
P. S-H. Wang
T. Williams
L. Wilson
K. Young

MATHLAB

During the past year, we implemented a new algebraic manipulation system. This system, called MACSYMA, represents the culmination of seven years of research on algebraic manipulation in Project MAC and the Artificial Intelligence Group. This effort included the doctoral thesis research of Professors Martin and Moses, and the implementation of parts of Carl Engelman's MATHLAB system. MACSYMA incorporates some of the best algorithms and design features of the earlier systems along with ideas obtained from systems built outside MAC. The current version of the new system is probably the most general of all existing algebraic manipulators. With the extensions that are planned or are already under development, the system should perform as well as or better than other systems in most situations. The system is running on the AI Group's PDP-10. It is written entirely in LISP. The system currently occupies about 60,000 words, of which 27,000 represent compiled LISP programs.

The capabilities of MACSYMA are best understood via examples. What follows is a session with an imaginary user.

(C1) sin(2*x) + 3*sin(x)/(cos(x) + 1) + (x + 1)**2@

(D1) $$\sin(2\,x) + \frac{3\sin(x)}{\cos(x) + 1} + (x + 1)^2$$

Each request typed by the user is given a line label (in this case C1). The system's responses are also labeled. One inputs algebraic expressions in a FORTRAN-like notation. Algebraic expressions are displayed in a two-dimensional notation which approximates the notation found in mathematical textbooks. An @ signifies the end of a request and indicates that the result is to be displayed.

(C2) INTEGRATE (%,x)@

A % always represents the previous expression which, in this case, is D1.

(D2) $$-\frac{1}{2}\cos(2x) - 3\log(\cos(x) + 1) + \frac{1}{3}(x + 1)^3$$

Simple integrals such as the one above are obtained by the current version. It is to be expected that the extensive integration facilities of Refs. 1 and 2 will be available in the system in the coming year.

(C3) DPART(D1,2,2)@

(D3) $$\sin(2x) + \frac{3\sin(x)}{\boxed{\cos(x) + 1}} + (x + 1)^2$$

One may refer to previous expressions by name (e.g., D1 in line C3). For this purpose, old inputs and results are stored on disk files. At the end of a session, the expressions may be retrieved, edited, and stored away for future use.

DPART helps one in editing two-dimensional expressions. The arguments of DPART locate a subexpression by specifying in which terms, factors, or arguments of functions it is located. Thus cos(x) + 1 is the second argument (i.e., denominator) of the second term in the expression. The indicated subexpression is highlighted by enclosing it in a box as shown. One can now use this subexpression or replace it by some other expression.

```
(C4)  A = B@
(D4)        A = B
(C5)  % = C@
(D5)        A = B = C
(C6)  % - 1@
(D6)        A - 1 = B - 1 = C - 1
(C7)  EXPAND(2*%)@
(D7)        2 A - 2 = 2 B - 2 = 2 C - 2
```

EXPAND causes each term of the sums to be multiplied by 2.

In addition to handling equations, MACSYMA has facilities for manipulating summations, derivatives, integrals and factorials.

```
(C8)  DERIVATIVE(X,1,Y,2,F(X + H,Y + K)) + SUM(I,0,N,I!)
      + INTEGRAL(X,A,B,G(X))@
```

$$(D8)\quad \frac{D^3}{DX\,DY^2}(F(X+H,Y+K)) + \sum_{I=0}^{N} I! + \int_A^B G(X)\,DX$$

The display program makes use of the limited character set of a GE Datanet to generate summation and integral signs.

A facility exists that allows one to define new functions. The language is close to ALGOL (actually it is closer to MLISP). For example, the factorial function can be defined as follows:

```
(C9)  FAC(I): = IF I = 0 THEN 1 ELSE I*FAC(I - 1)$
```

A $ inhibits the display of the result which is essentially a rehash of the definition.

```
(C10) FAC(5)@
(D10)       120
```

The system also has the capability of manipulating variable-length arrays. Arrays need not have their dimensions declared and may have their entries defined by some function. Only entries that are needed in computation will be evaluated by the function. To define an array A_I with entries $X^{I!}$, we can use the following definition:

```
(C11) A[I]: = IF I = 0 THEN X ELSE A[I - 1]**I$
```

Brackets denote arrays.

(C12) A[2]: X**3@
(D12) X^3

A : signifies assignment. One can assign values to array elements, thus overriding the general definition.

(C13) A[3]@
(D13) X^9

Another useful facility in the system is polynomial factorization.

(C14) FACTOR(X**6 - 1)@
(D14) $(X + 1)(X - 1)(X^2 + X + 1)(X^2 - X + 1)$

A powerful simplification algorithm which performs a cancellation of common factors in quotients is embodied in RATSIMP.

(C15) (X**2 + X - 6)/(X**3 + 6*X**2 + 9*X)@

(D15) $$\frac{X^2 + X - 6}{X^3 + 6X^2 + 9X}$$

(C16) RATSIMP(%)@

(D16) $$\frac{X - 2}{X^2 + 3X}$$

A powerful substitution mechanism is available in MACSYMA. It can be used to simplify expressions containing sines and cosines, for example. Let s stand for sin(x), and c for cos(x), then by substituting 1 for $s^2 + c^2$ in $s^4 + 2s^2c^2 + c^4$ we get 1, and by substituting 1 for $s^2 + c^2$ in $(s^3 - s)/c^2$ we get $-s$.

A substitution for anything but a literal is inherently ambiguous. Suppose we wanted to substitute c for xy^2 in x^2y^3, then reasonable answers are: cxy, c^2/y, and x^2y^3. By varying the calls to the substitution function, we can get all three of these responses.

MACSYMA has an extra-hairy simplifier in addition to the normal simplifier and RATSIMP, called RADCAN (RADical CANonical). RADCAN knows a great deal about exponentials, logarithms and algebraic expressions. For example, RADCAN will simplify

$$\log(e^{2x} + 2e^x + 1) - 2\log(e^x + 1)$$

to 0, and it will also recognize that

$$(2^{1/3} + 4^{1/3})^3 - 6(2^{1/3} + 4^{1/3}) - 6 = 0$$

RADCAN does not yet recognize

$$\log \tan(\frac{x}{2} + \frac{\pi}{4}) - \sinh^{-1} \tan x = 0$$

but we are working on that defect.

Extensive facilities exist in MACSYMA for letting a user define patterns and add new simplification rules to the system. For example, we can inform the system that cos(0) should become 1, and that cos(nπ) should be transformed to $(-1)^n$, when n is a non-zero integer. Given such rules cos(5π) simplifies to −1, and cos(6π) to +1.

Our final example is the solution of a classical problem in algebraic manipulation, the so-called F & G series problem of dynamical astronomy. The series F_i and G_i are given by the relations:

$$F_i = -\mu G_i + \frac{d}{dt} F_{i-1}$$

$$G_i = F_{i-1} + \frac{d}{dt} G_{i-1}$$

where $F_0 = 1$, $G_0 = 0$, and $\frac{d}{dt}\mu = -3\mu\sigma$, $\frac{d}{dt}\sigma = \epsilon - 2\sigma^2$, $\frac{d}{dt}\epsilon = -\sigma(\mu + 2\epsilon)$

The results are polynomials in ϵ, μ, σ.

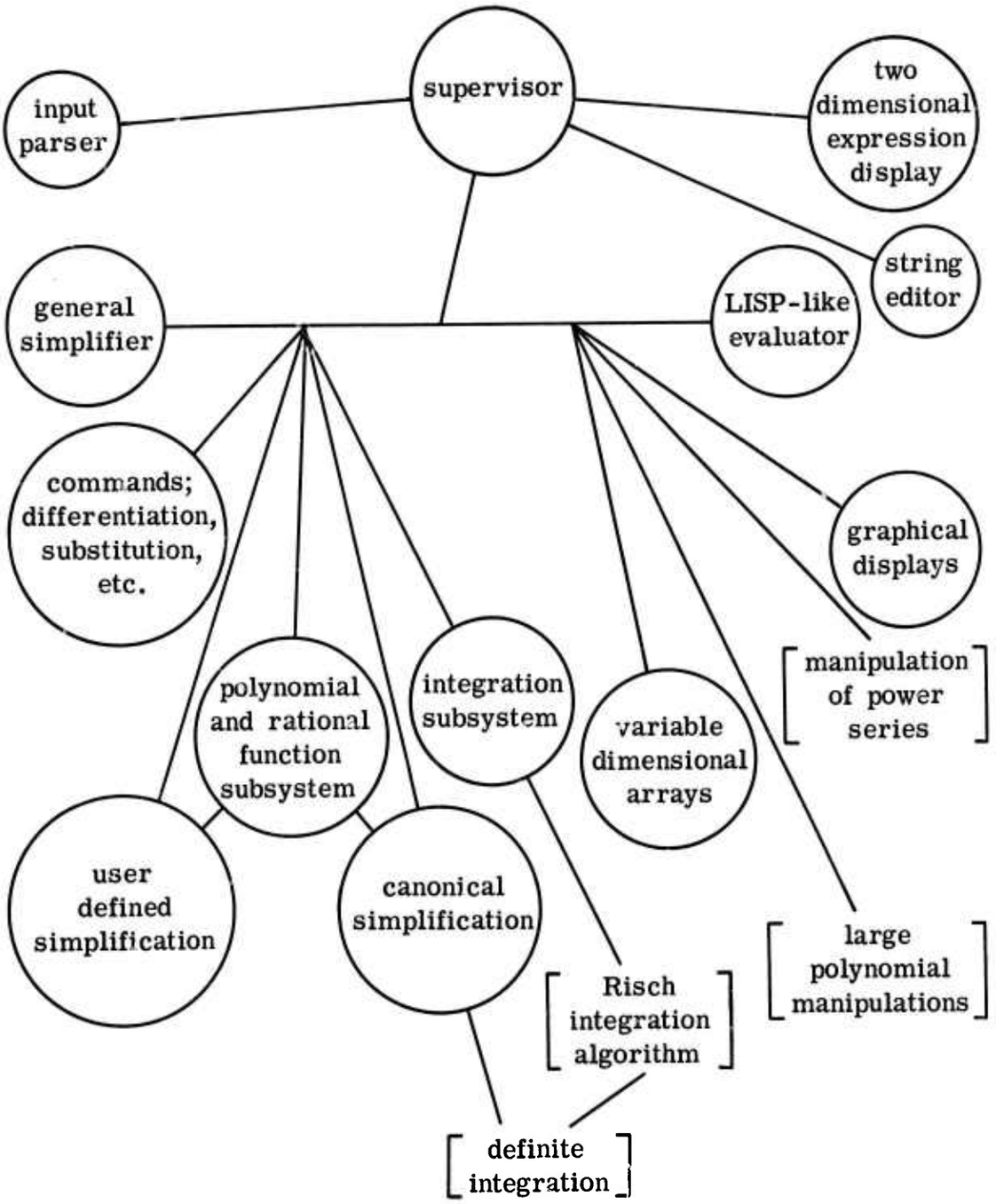
The representation of the problem of finding F_i and G_i is fairly natural in MACSYMA.

```
(C17) DERIVATIVE(T,1,EPS): -SIG*(MU + 2*EPS)$
(C18) DERIVATIVE(T,1,MU): -3*MU*SIG$
(C19) DERIVATIVE(T,1,SIG): EPS - 2*SIG**2$
(C20) F[0]: 1$
      The F_i and G_i will be placed on an array
(C21) G[0]: 0$
(C22) FANDG(N): = FOR I: 1 STEP 1 UNTIL N DO
      (F[I]: EXPAND(-MU*G[I-1] + DIFF(T,1,F[I-1])),
      G[I] : EXPAND(F[I-1] + DIFF(T,1,G[I-1])))$
```

The ALGOL FOR statement is part of the language as well as are the IF . . . THEN . . . ELSE construction, GO's and RETURN's. Parentheses define a block. Statements in a block are separated by commas. Local variables in a block are identified by a DUMMY statement. Variables in FOR statements are automatically made local.

```
(C23) F[5]@
                          3              2
(D23) 105 MU SIG  - 45 EPS MU SIG - 15 MU  SIG
```

The above is just a sampling of the facilities of the current system. The system is partitioned into major sections. The diagram below indicates the sections that currently exist. The sections in brackets are under development.



References

1. J. Moses, "Symbolic Integration", MAC-TR-47, December 1967.

2. J. Moses, "The Integration of a Class of Special Functions with the Risch Algorithm", MAC Memo M-421, September 1969.

Prof. R. R. Fenichel
Prof. J. Weizenbaum
J. C. Yochelson

UNCL

The Project MAC Progress Report V (July 1967-July 1968, p. 98 et seq.) describes the language at the heart of the TEACH system, which then was called PL/2 but which since has been designated UNCL (UNcommonly Clean Language):

> "It is an interactive language that somewhat resembles JOSS and other JOSS-like languages in several major respects: for example, the presence of block structure, a context editor, and a function-tracing feature".

The year ending June 1970 was spent in system design of the UNCL interpreter for Multics. A result of this effort was design of a novel list-processing system allowing multiple cell-types. No other separable results could be seen.

Publications 1969-1970

1. Fenichel, Robert R., and Jerome C. Yochelson. "A LISP Garbage Collector for Virtual-Memory Computer Systems", Communications of the ACM, XIII, 11 (November 1969), pp. 611-612.
2. Fenichel, Robert R., Joseph Weizenbaum, and Jerome C. Yochelson. "A Program to Teach Programming", Communications of the ACM, XIII, 3 (March 1970), pp. 141-146.

DYNAMIC MODELING, COMPUTER GRAPHICS AND COMPUTER NETWORKS

Prof. J. C. R. Licklider

A. C. Adams
B. J. Bailin
W. F. Bauer
A. K. Bhushan
G. R. S. Bingham
E. H. Black
M. F. Brescia
R. D. Bressler
H. R. Brodie
M. S. Broos
A. L. Brown
K. M. Brown
R. Bryan
M. T. Cheney
M. A. Cohen
S. E. Cutler
B. K. Daniels
Prof. G. A. Gorry, Jr.
M. J. Grano
F. Guertin
J. H. Harris
R. F. Hill
P. W. Hughett
W. F. Hui
J. P. Jarvis, III
R. Johnston
E. I. Katz
R. M. Katz
R. Lindsay
R. M. Metcalfe
R. E. Neubauer
P. A. Pangaro
C. L. Reeve
A. Vezza
J. E. Ward
R. T. Wong
F. L. Yost

DYNAMIC MODELING, COMPUTER GRAPHICS AND COMPUTER NETWORKS

I. INTRODUCTION

During the year, three new research groups were formed in Project MAC: Dynamic Modeling, Computer Graphics, and Computer Networks. The first two and about half of the third are strongly interrelated, and their work is reported upon in this one section. The other part of the work in computer networks, also interrelated but mainly involving members of the Computer System Research Group, is reported upon in that group's section.

The aim of the Dynamic Modeling Group is to develop methods, techniques, and a hardware-software system that will facilitate the formulation, representation and exploration of complex ideas and processes in the form of interactive computer-program models. The term "dynamic" connotes time-varying force and thus motion of or within the model; and one of the aspirations of the Dynamic Modeling Group is to adapt, extend and exploit the great capability of computer graphics to present complex interrelations in a comprehensible way. Another of the group's aspirations is to assemble a computer-based modeling system that will make available to the modeler a more extensive collection of resources than the group could itself possibly create. The interrelations with the Computer Graphics and Computer Networks Groups are therefore essential to the effort in modeling.

The aim of the Computer Graphics Group is to advance computer graphics as a medium of man-computer interaction and to improve the contribution of computer graphics to "machine-aided cognition". Within that general field, the group is concentrating mainly on software tools and techniques that will facilitate communication nd understanding of complex ideas and relationships.

The aim of the Computer Networks Group is twofold: (1) to participate with groups in other laboratories in pioneering the first nationwide network of general-purpose time-shared computers and (2) to explore and develop uses of the network that will facilitate resource-sharing and teamwork among geographically distributed people and computers. The group has undertaken to connect the Multics GE-645 computer system and the Dynamic Modeling/Computer Graphics PDP-10 computer system into the ARPA Network -- a network, sponsored by the Advanced Research Projects Agency, that is planned to embrace 15 or more time-shared computers within the next two years. The group is especially interested in uses of the network that will involve the operation of computer programs with subprograms in two or more computers and interaction with remote -- and in some instances even widely distributed -- sets of data.

II. THE DYNAMIC MODELING/COMPUTER GRAPHICS COMPUTER SYSTEM

Most of the effort of the three groups has been, and during the next two years will continue to be, devoted to the development of a computer system and its incorporation into the network. The development is essentially the experimental and iterative specification, implementation and testing of a philosophy of interactive computing that emphasizes three things: graphical display-and-control, software coherence, and the sharing of resources. In the next few paragraphs, the philosophy will be briefly set forth.

We want to create a computer system that will significantly facilitate and augment the efforts of users engaged in the kinds of intellectual activity that involve formulating, clarifying, exploring, testing and revising ideas. The concept of modeling that is basic to the group's philosophy involves (1) creating in the memory of the computer a model that represents the idea by specifying all its parts and all their interrelations; (2) executing the model and thereby revealing, through graphic display, the behavior implicit in its specification; (3) observing the behavior and comparing it with expectations, either intuitive or based on quantitative data; and (4) revising the representation in the memory of the computer, executing the revised model, and so on.

III. PLANS FOR RESEARCH IN MODELING, GRAPHICS, AND NETWORKS

For two reasons, the report of research in the three areas will be mainly a report of plans. The first reason is that, because the groups are new, their main effort has been planning. The second is that we want others to understand what we aspire to create and do.

The plans have been influenced greatly by the fact that, in order to conduct research in man-computer interaction -- in any of the three areas -- one must have, or be involved in creating, a computer system that will support the computer end of the interaction. In the present case, that means a rapidly and powerfully responsive local computer system -- an interactive time-shared system -- connected into a network of interactive time-shared systems. We need a computer system similar to that of the M.I.T. Artificial Intelligence laboratory or to that of Douglas Englebart's "On-Line System" group at the Stanford Research Institute, but with stronger emphasis on kinematic graphic display, and we need to incorporate the system into the ARPA Network.

Basic Schema

The plans call, therefore, for a protracted effort in system-building. The building will by no means be merely the implementation of an existing design. It will proceed, of course, within the guidelines of a

general schema, but it will involve the continuing, iterative interplay of design, experimental implementation, testing, evaluation and modification. Thus, in style, the research of the three groups will probably be more closely akin to that of the Computer System Research Group than, for example, to that of the Theory of Automata Group. It will seek understanding through the interaction of physical synthesis and experimental analysis much more, especially during the system-building period of about three years, than through theorem-oriented theoretical work.

The general schema that is guiding the work embodies the following features:

1) Vigorous development and exploitation of graphical interaction.

2) Heavy reliance upon "core residency" of procedures and data sets to permit fast interaction and kinematic graphical display.

3) Deferral of linking and binding operations (linking among procedures and binding of procedures to data) until late in the over-all process -- usually until the time of initial execution.

4) Use of a greatly extended (and further extensible) "run-time package" -- a library of procedures and data sets that can be retrieved and linked or bound into the currently operating program or model during its testing or execution.

5) Mastery of a considerable diversity of data types and emphasis on type-checking and (when necessary) automatic type transformation and reformatting.

6) Integration of the processes of formulation, programming, testing, debugging, program modification, and data-base updating insofar as possible into one coherent activity.

7) Integration of the main programming language and the interaction languages of the system insofar as possible into one coherent language.

8) Heavy reliance (tempered somewhat by realism) upon using already programmed and available procedures in the (remote, network) computers in which they operate.

9) On-line aids to users, including much on-line documentation and a descriptor-based retrieval system to facilitate the users' learning to operate the system and their finding and application of procedures, models, and data.

10) Significant augmentation of the capabilities of serious, long-term users at the expense, if and as necessary, of facilitation of the work of casual, one-time or sporadic users.

Hardware System

The physical computer system on the basis of which we hope to realize an approximation to the schema outlined is shown diagrammatically in Fig. 1. The foundation of the planned system consists of a Digital Equipment Corporation (DEC) PDP-10 (KA10) main processor and an Evans and Sutherland (E&S) LDS-1 display subsystem, 256K words of 36-bit memory (most of it of 1.6 to 2.0-microsecond cycle time), and three DEC disk-pack drives capable of storing 15 million words. There are two sets of consoles: four major consoles, each with a display driven by the E&S subsystem, a stylus and tablet (ST), and a Computer Displays (now Adage) Advanced Remote Display Station (ARDS), and four or more programming consoles, each consisting of an Imlac PDS-1, which is a minicomputer plus display and keyboard. The system includes, also, eight microtape units, an operator's Teletype, paper-tape reader and punch, a line printer, and an interface to the Interface Message Processor (IMP) that connects M.I.T. computers into the ARPA Network. The system may be able to use the network connection to Multics in lieu of standard magnetic tape. If not, it will need a tape interface and controller and a magnetic-tape unit. Card input will be handled via the network.

The main ones of the foregoing items have been or are to be purchased. In the interest of holding down the over-all cost, however, the two peripheral-device interfaces and scanners (shown as one long rectangle in the figure), most of the memory ports and interfaces, a device to adapt the DEC input-output bus to transistor-transistor logic, and the interface to the IMP are to be constructed in the laboratory. In addition, the E&S (Kratos) displays, the stylus tablets, and the ARDS units will be integrated into consoles in the laboratory.

Software System

The main components of the planned software system are listed in Table I. The table expresses something of the extent of the undertaking but probably not much about its organization. The following paragraphs will deal briefly with its organization.

The plan calls for borrowing much of the basic system software from the Artificial Intelligence laboratory. Indeed, during the first year, we adapted, or in a few cases simply adopted, from the PDP-10 computer system of that laboratory, an amazingly fast and powerful supervisor (ITS), an excellent assembler (MIDAS), and several very useful utility routines. The borrowed software has gotten us off to a rapid start, though admittedly it is a start not precisely in the planned direction. Our concerns with graphics, coherence, and sharing will require that we modify some of the underpinning even while we are building upon it. We recognized the inherent danger in doing that, but the advantage of getting under way at once compelled us to accept the risk.

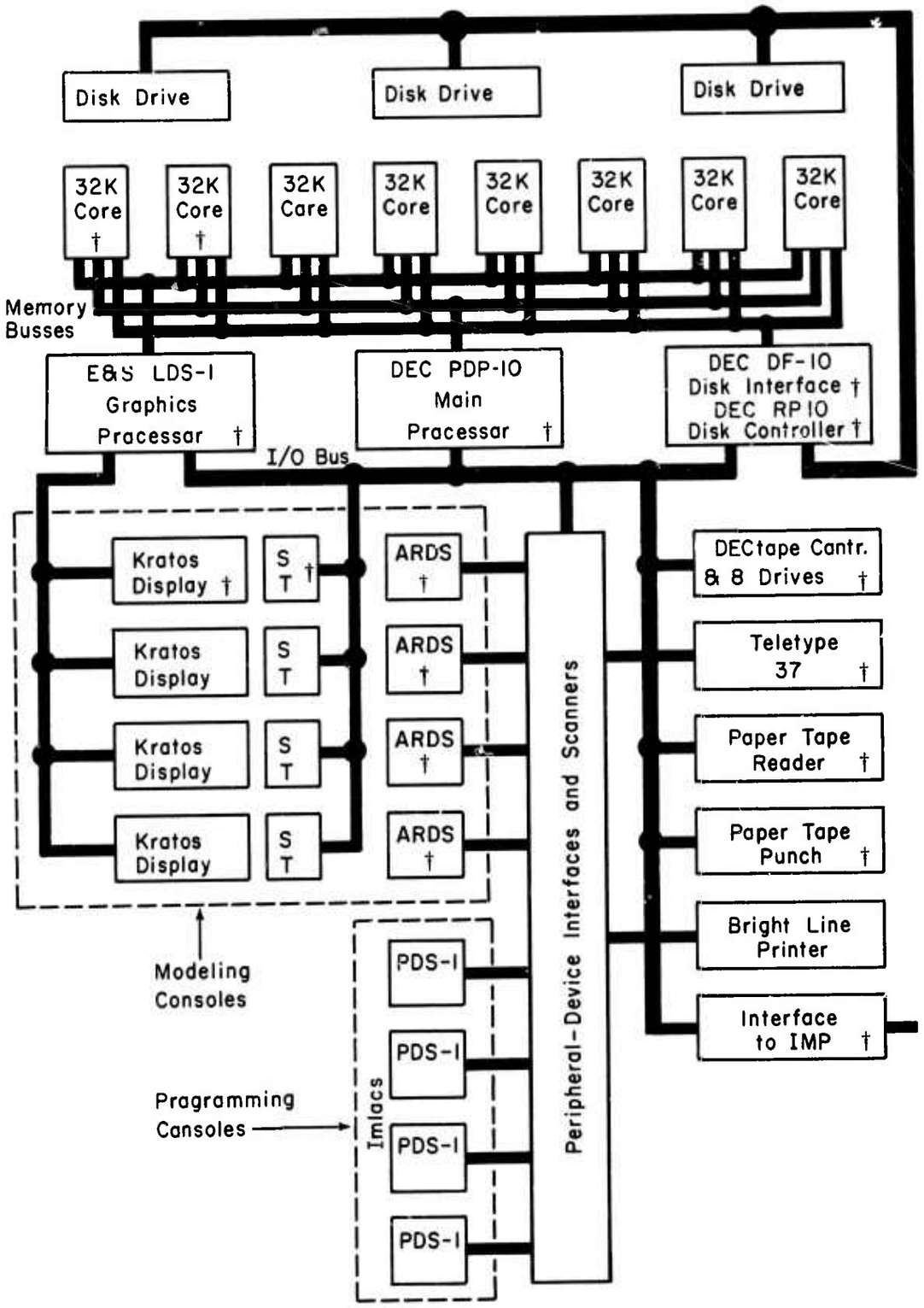


Fig. 1.
Proposed hardware plan for the Modeling/Graphics/Networks computer system. (The † symbol indicates hardware items operating at the end of the year.)

TABLE I

Main Software Components of Planned System for Research
in Dynamic Modeling, Computer Graphics and Computer Networks

IDENTIFICATION		*FUNCTION*	*WORK REQUIRED*
A.	SYSTEM SOFTWARE		
1.	ITS*†	Time-sharing supervisor	Adapt
2.	Δ	High-level programming language(s)	Select, implement or adapt for PDP-10
3.	MIDAS*†	Assembly language	Adapt
4.	Δ	Dynamic loader	Design and implement or adapt from "STINK"*
5.	DDT*†	Debugging aid	Adapt
6.	Δ	Graphical debugging aid	Design and implement with some use of DDT
7.	MONIT	Interim command interpreter	Design and implement
8.	Δ	Graphical command interpreter	Design and implement
9.	TECO*†	Editor	Adopt, then adapt
10.	NCP	Network control program	Design and implement
11.	INFO*†	On-line documentation of utility routines	Adopt, then adapt
12.	Δ†	Interface to Imlac PDS-1, incl. loader, editor, and assembler	Design and implement some; adapt some
13.	OLIVER	Interface with individual user; agent for individual user	Design and implement
14.	CARE†	Mediator -- handle calls and returns, facilitate user intervention	Design and implement
15.	MACDMP*†	Dumper and loader of system routines	Adopt
16.	Δ	Interface to programs that operate under DEC time-sharing supervisor	Design and implement
17.	LOCK*†	Utility associated with ITS	Adopt
18.	Δ	Library construction and maintenance	Design and implement

TABLE I (cont.)

IDENTIFICATION		*FUNCTION*	*WORK REQUIRED*
19.	Δ	Flow-chart interpreter	Design and implement, adapting some from GRAIL
20.	Δ	Core-resident shared subroutine and data-set address table	Design and implement
21.	Δ	Core-resident (individual) user's subroutine and data-set address table	Design and implement
22.	Δ	Inter-user communication	Design and implement
23.	Δ	Disk salvager and other disk utilities	Design and implement
24.	SATAN	Chief demon (starts up other demons as needed)	Design and implement
25.	SPOOL*	Background printing	Design and implement
26.	Δ	Network logger, network test routines, and other network utilities	Design and implement
B.	GRAPHICS SOFTWARE		
1.	Δ†	Stylus-tablet input	Design and implement
2.	Δ†	Character recognition	Design and implement
3.	Δ†	Two-dimensional graphing	Design and implement
4.	Δ†	Three-dimensional graphing	Design and implement
5.	Δ	Complex-plane display	Design and implement
6.	Δ	"Visual statistics"; visual data analysis and estimation of relationships	Design and implement
7.	Δ	Graphical display of program states, actions and relationships	Design and implement
8.	POLYVISION	Control of composition of multi-picture displays	Design and implement
9.	Δ	Transformer from E&S to ARDS and Imlac displays; preprocessor for transmission via network	Design and implement

TABLE I (cont.)

IDENTIFICATION		*FUNCTION*	*WORK REQUIRED*
10.	Δ	"Library of shapes"	Design and implement
11.	Δ	Graph and diagram editor	Design and implement
12.	Δ	Graphical network command language	Design and implement
C.	PROCESSING ROUTINES		
1.	Δ	Two-dimensional parser	Design and implement
2.	Δ	Fast Fourier transform	Implement
3.	Δ	Other integral transforms	Design and implement
4.	LEXICON-TEXT	Lexicon-based text handling	Design and implement
5.	Δ†	Algebraic compiler for use with stylus-tablet, character recognizer, and two-dimensional parser	Design and implement
6.	Δ	Extension of high-level language to encompass diverse data types	Design and implement
D.	INTEGRATIONS		
1.	Δ	Extension of high-level programming language to encompass graphics and diverse data types	Design and implement
2.	Δ	Introduction of coherence into the interaction languages of the main system and utility routines	Design and implement
3.	Δ	Unification, insofar as possible, of main high-level programming language and the main interaction languages	Design and implement
4.	Δ	Introduction of coherence into library of procedures and data	Design and implement
5.	Δ	On-line documentation system	Design and implement

Δ *Not yet named.*
* *Adopted or adapted from AI Group routines.*
† *Preliminary version operating July 1970.*

The initial organization of the system has been determined to a large degree by the nature of the adopted and adapted system and utility software. It is an organization in which a user interacts with one routine at a time, with each routine mainly in its own language. For example, one may log in to MONIT, ask MONIT for the editor TECO, prepare an assembly-language program (naming it, for example, PROG MIDAS) and file it, return to MONIT, call the assembler MIDAS, have MIDAS translate PROG MIDAS into its machine-language form PROG BINARY, return to MONIT, call the debugging routine DDT and through it call the loader STINK, have STINK load PROG BINARY and transfer the symbol table(s) to DDT, tell DDT to execute PROG BINARY, observe that PROG BINARY does not function properly, return to MONIT, call TECO, retrieve PROG MIDAS, edit it, return to MONIT, call MIDAS, and so on and on.

The foregoing description is not a burlesque. Except for leaving out some steps -- for example, the first time MIDAS translates PROG MIDAS, MIDAS finds syntactic errors that force the user to go back at once through MONIT to TECO -- it is a fair account of the sequence of transactions required by most present-day time-sharing systems of a programmer who simply wants to prepare and execute a program. If he wants to do anything more complex than that, he has to turn into a veritable switch engine. We want to get far away from all that. The initial organization will therefore have to be changed. We want it to be more like that of LISP systems in which program writing, filing, translating, debugging, and almost all the other operations are carried out within the coherent framework of LISP. Why not simply adopt LISP? We could do a lot worse. The main reasons for not simply adopting it are that current LISP systems, while highly coherent or even unified internally, are almost impossible to put into effective communication with any other systems; that they recognize only one data type or very few data types; and that they are not well developed into the dimension of graphical interaction.

One major step in effecting the new organization of software will be to create for the user the effect of communicating with one inclusive program, an editor-assembler-loader-debugger-informer, instead of a multiplicity of routines that converse in diverse and conflicting languages. That step is perhaps not crucial for the experienced programmer, and so it will not be taken first, but we think it will be crucial for substantive modelers.

Another major step is so to arrange matters that a user can monitor the operation of his program or model, intervene at any appropriate time, interrupt the operation, make changes, backtrack as far as necessary, and resume operation. The mediator, the dynamic loader, the core-resident subroutine and data-set address tables, the documentation-

retrieval subsystem, and the unified programming-interaction language (see Table I) are all part of the required mechanism.

A third major step is to provide for a large amount of sharing of pure, core-resident procedures. Residency in primary memory is of course the key, in the present state of computer technology, to very high responsiveness, very fast interaction. But primary memory is almost always at a premium. It is necessary, therefore, to determine carefully just which procedures and data to hold in primary memory -- and even more essential to prepare procedures in such a way that the most frequently used components (which are likely to be basic building blocks used in many different programs) can be separated out and held continually in primary memory. It will be necessary to experiment in order to find the optimum in this complex area, but it is obvious that the system must be designed in such a way as to permit experimentation. The third step is intended to make it so.

Other steps of organization, which we shall for the sake of brevity, just mention, are:

1) Substituting interaction through diagrammatic graphics for much of the alphanumeric interaction that characterizes most time-sharing systems.

2) Moving much of the initiative, in the interaction between the user and his running program, from the program to the user. This involves fundamentally separating many of the interactive elements from the program proper and replacing them with "event identifiers" that provide handles for, but do not themselves initiate, interaction.

3) Developing a large and coherent library of procedures and data sets that can be linked or bound into users' programs at run time. We have set the semirealistic goal of 1000 such procedures and 100 such data sets. We fear, however, that as many as 10,000 basic procedures may be needed to provide a facility in which a typical "new" program or model consists of 10% new and 90% library material.

Planned Techniques for Modeling and Graphical Interaction

At present, the field of computer-based modeling is divided into two areas, one concerned with sequences of discrete events, the other concerned with dynamic interactions of variables that, for the most part, are continuous in the real world and, only because the nature of the computer forces discreteness, discrete in the computer. Each area has its modeling or simulation languages -- for example, GPSS and SIMULA for discrete events and DYNAMO for (qualifiedly) continuous variables. The languages make it convenient to express the operations in which they specialize but difficult to "follow the lead of the problem" if the

problem crosses language boundaries. We are interested, therefore, in a technique of modeling analogous to that made popular in childhood by Tinker Toys, Erector Sets, and Mechano Sets: Start with an idea (perhaps a bit nebulous but interesting), a lot of modular parts, and a lot of connectors. Assemble the parts by hand under visual monitoring and control. (Doubtless a kind of language is involved in the process, but it does not intrude or sharply delimit.) Get new ideas and insights as you go. Take some of what you have built apart and rebuild it to incorporate new features. As soon as you have something that works, test it, see how to make it better, make it better, test it again, and so on. We want to develop that technique in the context of the computer. In the process, we hope to avoid the childhood frustration of running out of parts.

In modeling complex processes or systems, one tends to represent their parts in terms of abstract symbols, which are usually characters or words or word-like strings of characters. Such symbols have proven very helpful to modern man, and he has become accustomed to thinking of them as advances over the picture-like glyphs used in earlier times. No doubt they are. But it is not clear, and should not be tacitly assumed, that words are on the best evolutionary path. The advent of the computer opens up new possibilities for dynamic representation in which words, being static, may not play the major role. We plan to explore the field of what might be called "dynamic glyphs": quasipictorial signs that can move about and change shape as the things they represent act or are acted upon. Perhaps the conveyors of meaning in future languages may evolve, under the influence of the computer, from the animated cartoon. Whereas present-day natural languages are strong in nouns and weak in verbs, it may be that future languages may have as many verbs as future computers have subroutines. In any event, this appears to be a potentially important area for technique development.

When one works at a computer console, even a sophisticated and expensive one, he soon feels severely constrained by the size of its display screen, which is not nearly so large as the array of desktops and tabletops, covered with papers and reprints, to which he is accustomed. It is frustrating to lose what is on the screen in order to get to see something else, for often one wants to compare the one with the other and has to call the old up again to check it -- and thereby loses the new. In short, time-multiplexing a 10-inch-square area seems to be a retrograde step from "space-multiplexing" in a large workspace. We plan, therefore, to explore a number of techniques, ranging from very fast time-multiplexing a single screen through splitting screens and using several screens in parallel, to simulate an office workspace, with the aid of head-mounted cathode-ray tubes (à la Ivan Sutherland) and eye-fixation sensing (à la Minsky, Papert, and Geffner). The exploration will include distorted displays in which the area of present

interest is presented in larger scale than the other areas -- as in the "New-Englander's Map of the United States".

Light pens and styli are not well developed, even yet, and therefore call for further technique development. We hope to develop a manual-input system that will "understand" a repertoire of metalinguistic signs analogous to proofreaders' marks as well as the elements relating to substantive input. The stylus language should be a part of a coherent interaction language and not, as it is now, a sequence of decomposed and nonballistic movements back and forth between a drawing and a set of light buttons.

We hope in due course to incorporate a limited speech vocabulary into the coherent system.

Graphical definition and control of processing and composition and control of display will be a focus of our effort. We hope to develop techniques that will let a programmer or a modeler formulate and specify a program or model primarily through graphic means, then graphically control its testing, modification and execution, choosing at each stage from a library of display and control modes and formats. The interrelation of language and metalanguage is a main theme in this area.

Plans Relating to the PDP-10 Computer System and the ARPA Network

The initial effort in networking must be, of course, to receive an IMP and make the necessary interfaces, hardware and software, between it and the PDP-10. Then programs must be developed to mediate communication between the PDP-10 and other host computers. Development of those programs, which include the Network Control Program (NCP) listed in Table I and a Logger not listed there, will involve much interaction between the Computer Networks Group of Project MAC and corresponding groups in other ARPA Network organizations. We plan first to develop interim interface and communications programs, then full-fledged versions. There may be a series of the latter as the Network evolves.

As soon as communication is established with other host computers, network research will open up to include several parallel undertakings. These will include:

1) Systematization of access from other hosts to resources available in the PDP-10 system.

2) Systematization of access through the PDP-10 to other host systems -- as though the user were logged directly into them. (This way of using the network is only of minor interest to us, but it is a natural step.)

3) Transfer of files between other hosts and the PDP-10.

4) Use of remote resources by programs operating in the PDP-10; integration of certain remote resources into the PDP-10 system in such a way as to make them appear to be local.

5) Establishment of procedures that will make it possible for other hosts to use the graphical-processing facilities (e.g., picture "trimming and framing") of the E&S processor.

6) Study of data types and data translation within the network from a taxonomic point of view.

7) Study of models and modeling languages and facilities available within the network.

In executing those plans, we expect to take advantage of the fact that the Multics computer will also be connected into the ARPA Network. In taking the early steps, it will be helpful to work with Multics and PDP-10 consoles side-by-side. We expect also, however, to enter into joint research efforts with workers in other host organizations.

IV. PROGRESS AND THE PRESENT STATE

During the past year, the three groups were brought up approximately to size (see listings at beginning of section), main parts of the computer system were received or constructed and assembled into a working facility, and beginnings were made on several of the software-development and technique-development projects. In order to check the system planning and design against demands posed by actual modeling, a neuronal-network model and an air-traffic-control model were constructed. The Interface Message Processor arrived and was installed, and the physical interface between the GE-645 and the IMP and the one between the PDP-10 and the IMP were designed and constructed. At the end of the year, enough of the parts of the envisioned system were in place and operating to constitute a strong bootstrap, and an intensive summer of programming -- manned in part by 25 student programmers -- was under way. The software library count stood approximately at 50 useful programs, but most were not yet well documented. An on-line documentation system was being created -- initially in the Multics GE-645 rather than in the PDP-10 because the latter lacked disk files.

The "†" marks in Table I identify the software items of which preliminary versions were operating at the end of the year. Work was under way on about half the items not marked. The prevailing feeling in the three groups was that too many things were going on at once, that it was difficult to see the woods for the trees, that there was very much work indeed to be done, but that progress was accelerating as new components and subsystems came into operation -- and that one could already sense, through the one E&S console, some of the power of highly responsive graphical interaction.

ON-LINE CIRCUIT DESIGN AND HYBRID COMPUTING STRUCTURES

Prof. M. L. Dertouzos

F. G. Abramson
D. L. Isaman
G. P. Jessel
M. E. Kaliski
P. A. King
M. P. Lum
C. Lynn
J. R. Stinger
H. C. VanSteveninck (Guest)

Electronic Systems Laboratory

ON-LINE CIRCUIT DESIGN AND HYBRID COMPUTING STRUCTURES

For our research group, this was a year of transition from On-Line Circuit Design to Hybrid Computing Structures. The former research, initiated five years ago, shortly after the inception of Project MAC, was concerned with the use of on-line computation in the design of electronic circuits. Major results of this work have been published in the open literature and are incorporated in CIRCAL-2, a general-purpose, on-line, circuit-design program.* This program, currently operational on CTSS, is undergoing conversion by a company which will make it commercially available for IBM 360-67 use by the end of the year. The new research in hybrid structures was motivated by the long computing times and inefficiencies we encountered in solving very large systems of algebraic and differential equations by using conventional machines. Our current effort is concerned with the study and development of novel hybrid computing techniques and structures, which possess the accuracy of digital systems and the speed of analog systems.

I. CIRCAL-2: A GENERAL-PURPOSE CIRCUIT-DESIGN PROGRAM

The main objective of this program is the effective use of on-line computer utilities in the design of electronic circuits. It summarizes in its structure the conclusions of several years of research as well as "lessons" learned from the predecessors, CIRCAL-0 and CIRCAL-1, of that program.

In developing CIRCAL-2, we have tried to make good use of on-line computation in both the interactive and more formal optimization types of design. Accordingly, the program can be used for the repeated analysis of lumped networks, with several features and provisions for efficient interaction; or it can be commanded to automatically optimize a circuit, in accordance with a prescribed rule. The main features of the program are as follows:

1) Multiple Analysis Capability: -- The program accepts any number of different circuit-analysis techniques, residing on disk. One of the consequences of this feature is the development of a sufficiently general common set of network elements. Another consequence is the use of time or frequency as the computing dimension.

2) Homogeneity of Information Structures: -- The on-line circuit-design process involves the input, modification, output and definition of information that describes elements, networks, or

* (See J. R. Stinger and M. L. Dertouzos "CIRCAL-2: A General-Purpose On-Line Circuit Design Program User's Manual", M.I.T. Electronic Systems Laboratory, Report ESL-R-381, May 1969.)

sources. It involves, also, the output of informational and diagnostic messages and the specification by the user of automatic circuit-optimization procedures. All information, regardless of meaning, is handled uniformly by a text editor. The information necessary for analysis, as well as the results of analysis, is handled in a uniform way through the so-called input and output data structures which are sufficiently general to permit the use of a large number of analysis techniques. In addition, commonly used objects are isolated for economy; for example, the CIRCAL-2 operator which permits the definition of functions, is applicable to nonlinear characteristics of network elements, waveforms of sources, output of a function of a computed variable, and the computations performed in an optimization process.

3) Efficient On-Line Interaction: -- An express and a slow mode of operation are provided, enabling (a) the experienced user to stack commands with minimum interaction and (b) the inexperienced user to proceed on a question/answer basis. In addition, all information that is changed in the course of ordinary design, such as resistor values, analysis time increments, etc., is organized in such a way that, once initialized, it need only be changed incrementally, thereby minimizing significantly the interaction cycle.

4) Ability to Define Optimization Procedures: -- In formal design, a common procedure is the adjustment of certain parameters until some performance index is minimized. CIRCAL-2 makes possible the definition by the user, in a circuit-oriented language, of instruction sequences that behave as "pseudo-users" -- that is, they automatically modify/analyze a circuit until and if the desired index is minimized. Of course, these automatic optimization techniques are used when the design process is fully understood and algorithmically expressible. In the more common design cases where such intangibles as experience and intuition are necessary, the users resort to the on-line interactive features of the program.

II. STATUS AND USE OF CIRCAL-2

CIRCAL-2 is currently operating on CTSS (modified IBM 7094). During the Summer of 1970 it is being converted by SofTech, Inc. (Waltham, Mass.) for 360-67 and 360-75 use, with additional plans for conversion to other popular machines. Several electronics and integrated circuit companies are expecting to use CIRCAL-2 after its conversion (toward the end of this year).

The analysis techniques residing in the present version of CIRCAL-2 are transient analysis, frequency analysis, and frequency analysis of

sparse networks. In addition, we have used CIRCAL-2 as an experimental forum in developing new analysis techniques. One such technique developed by our group deals with the algebraic/recursive representation and analysis of nonlinear networks[4]. In this technique, circuits are treated as recursive constructs. That is, specific rules are postulated for the interconnection of two electronic networks so as to result in a new network. Recursive use of these relatively few and simple rules gives rise to any one of a large class of complex circuits. We have found that complex-looking circuits, if designed by humans, have a very simple recursive structure, since the designer has created them with some fixed "rules" -- such as "left-to-right information flow", and at any given stage, the processing of at most two or three signals in the generation of a new signal.

In addition, we have studied the case of highly repetitive circuit structures, such as those encountered in integrated digital circuits and have suggested means for their analysis.

Our group plans to phase out the research on on-line circuit design toward the end of this year coincident to the commercial availability of CIRCAL-2.

III. CONTINUITY AND COMPUTATION

Our new research borders the classical areas of continuous (analog) and discrete (digital) systems. We have several approaches and objectives in mind: First, by viewing continuous systems from a computational point of view, we can ask several questions of a theoretical nature:

> What is the logical capability of continuous systems? Are there input-output tasks that cannot be handled by any continuous system? Is there a computational hierarchy of continuous machines with ever-increasing computational power? Is there an inherent upper limit to the tasks conducted by any continuous machine (analogous to Turing's thesis for discrete machines)?

Questions such as these may, at first, appear fatalistic, since continuous computing systems in their known form (analog computers) have a limited accuracy and are difficult to program. We have, however, two major reasons for studying these questions: First, in contrast to digital systems, our ability to design continuous systems that perform a given input-output task is severely limited. Consider, for example, the computational hierarchy of discrete machines (left half of Fig. 1) and our suggested continuous counterparts on the right. While we can express input-output tasks for the discrete machines, e.g., using Boolean Expressions for (a), regular expressions for (b), and recursive functions and predicates for (c), we have no known ways of expressing continuous I/O (input-output) tasks for (e) and (f). Even more important,

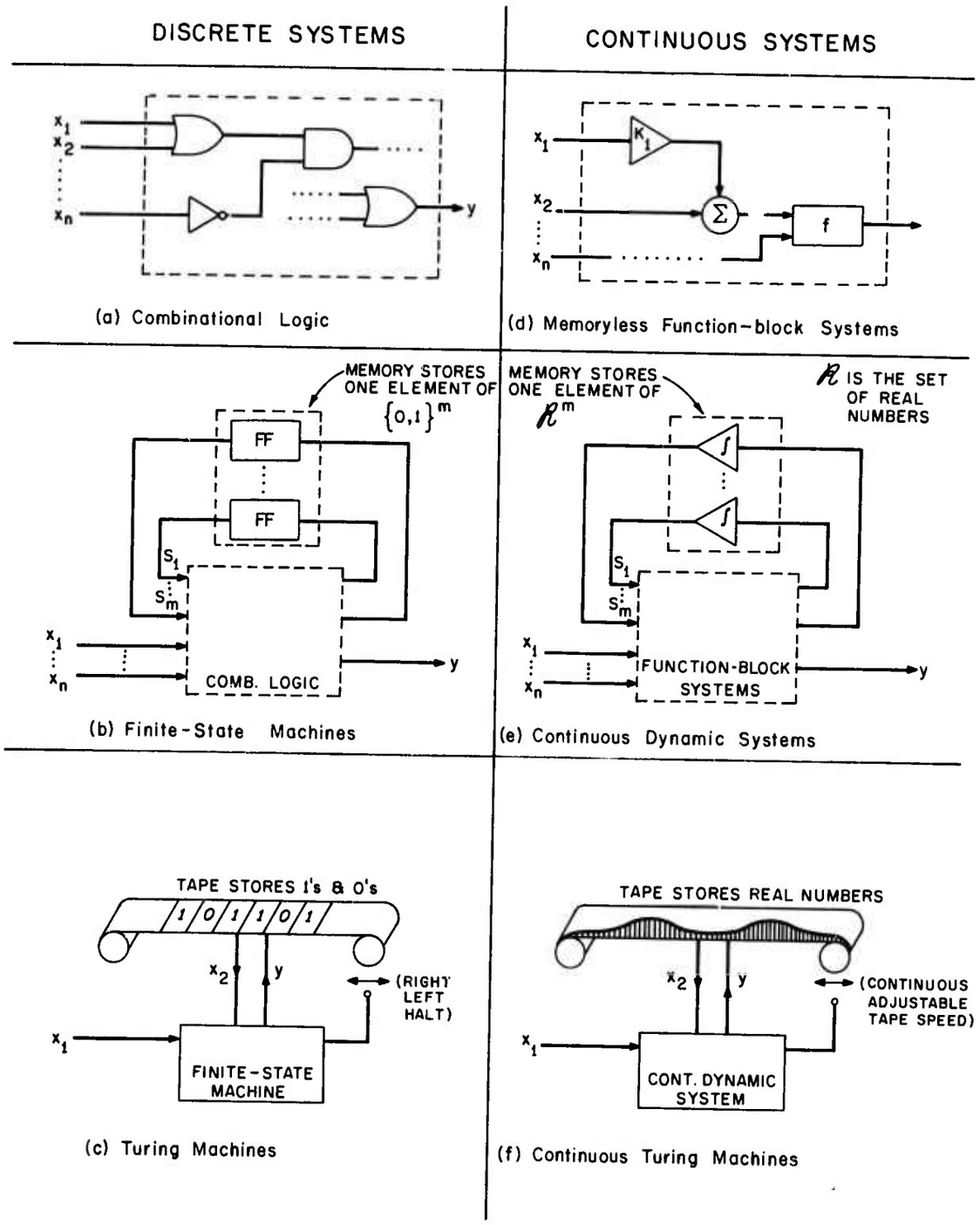


Fig. 1.
Computing-Machine Hierarchies.

while we have procedures for designing discrete machines, e.g., Quine-McCluskey for (a) and Huffman-Moore for (b), we lack any such procedures for the case of continuous machines. We believe that both the representation of I/O tasks as well as the synthesis of dynamic systems such as (e) are possible. We have obtained some early results to substantiate our claim. Specifically, we have developed procedures which, given practical constraints on the nonlinear function blocks that can be used, return the number of integrators needed and provide the specification for the function blocks to be used in (e) so as to accomplish the desired task.

Our first reason, then, is the practical synthesis of continuous dynamic systems. The second reason concerns the use of continuous and discrete components together to provide effective procedures, as discussed in the following section. In order to combine discrete and continuous machines we must have a considerably better understanding of the latter, especially in regard to logical and computational issues.

In this area, our activities to date have been as follows:

1) Study of autonomous systems of the type of Fig. 1(e), with the integrators replaced by sample and holds (sh), i.e., objects operating discretely in time, yet capable of storing a real number. This study has given a large number of results as to the way in which function-block constraints such as boundedness and Lipschitzness* affect the number of memory variables and the way in which the system is interconnected. Given specified sequences of real numbers, we have developed procedures for constructing autonomous sh machines that realize these sequences under any additional constraints on boundedness and Lipschitzness of the constituent function blocks.

2) The results of part (1), above have been extended to autonomous integrator systems. We have established that no one-integrator system can ever oscillate, and that a three-integrator system can realize arbitrary I/O tasks with no constraints on the function blocks. We have further established that the number of integrators grows in inverse proportion to the Lipschitz constant and to the bound or the functions used in the system.

3) A language similar to regular expressions has been developed for the expression of autonomous tasks of systems of the type of Fig. 1(e).

4) We have investigated Real Turing machines of the type of

* These latter two conditions are practically necessary. The former, for instance, insures a finite voltage range while the latter restricts the "wiggliness" of every function in the system.

Fig. 1(f), trying to establish a model that describes naturally any continuous task. Although we are not yet certain as to whether we have found such a model, we have developed some powerful models of Real Turing machines that have many interesting properties. One such model can solve precisely all those halting problems that a conventional Turing machine cannot solve. We have also identified problems unsolvable by the Real Turing machines.

The study of Real Turing machines, besides its theoretical interest, has also a practical appeal, in that it can lead to universal analog machines, and, for our purpose, to universal and interpretive hybrid machines, discussed below. The fascinating question of a "Real Turing Thesis" is still open -- that is, is there an inherent upper limit to all the "continuous computational processes"? Alternatively, is an analog computer at that logical limit? Can it solve any problem that a human can solve with, say, graphic constructions on paper, or any problem that can be handled by any other special analog system? Throughout these questions, it is assumed that our machines are capable of infinite precision, the primary concern being to study how continuity affects computation.

IV. HYBRID COMPUTATION: LINEAR EQUATIONS

One of the reasons for our study of continuous computation is our discovery of certain mixed or hybrid structures -- consisting of continuous and discrete subsystems -- which can be so arranged as to solve large and relatively complex problems with the speed of analog systems and the accuracy of digital systems.

The basic idea behind these structures is computation with two classes of variables, which we have called pressures and flows. The flow variables include the unknowns and are digitally expressed as binary numbers. The pressure variables are analog quantities which are subject to accuracy limitations. Execution involves the adjustment of flows on the basis of the inexact pressures in accordance with some prescribed rules, and the adjustment of the pressures on the basis of exact computations on the flows. These adjustments may be conducted synchronously or asynchronously in time.

The first structure that we have studied is shown in Fig. 2. It is intended for the solution of large systems of linear equations. Here the equation to be solved is $\underline{A}\underline{x} = \underline{y}$ where vector $\underline{y}$ and matrix $\underline{A}$ are given, and vector $\underline{x}$ is the unknown. The computing structure consists of two parts: (1) a digital or exact substructure which checks to see if a suggested vector (a flow variable) $\underline{x}_i$ does satisfy the above equation; this structure computes the error $\underline{y} - \underline{A}\underline{x}_i$; and (2) an analog substructure which computes inexactly the pressure variable

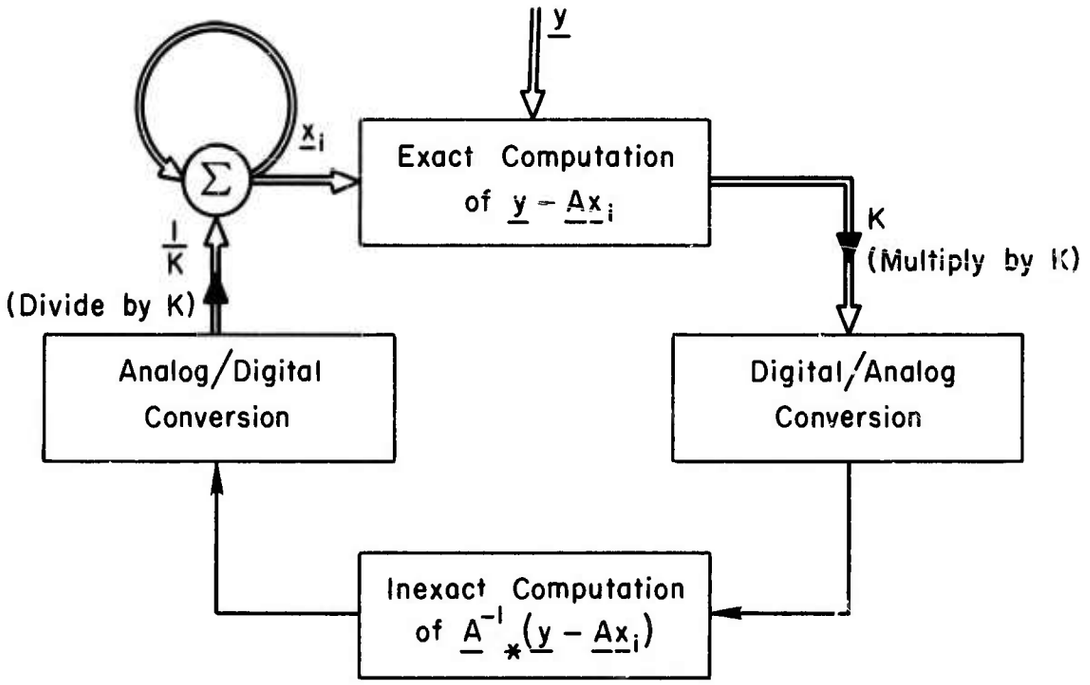


Fig. 2.
A Hybrid Computing Structure.

$\Delta x_i = A^{-1}_*(y - Ax_i)$, where A^{-1}_* is an inexact approximation to A^{-1}, discussed below. Execution is iterative, each iteration consisting of first the computation of the error, then of the pressure Δx_i and then of x_{i+1} as $x_{i+1} = x_i + \Delta x_i$. If A^{-1}_* were exactly A^{-1}, then the exact solution would be obtained in one iteration. Since A^{-1}_* is inexact, each iteration brings x_{i+1} closer than x_i to the solution, with contraction depending on the error between A^{-1} and A^{-1}_*. The implementation of A^{-1}_* that we envision does the inversion implicitly -- that is, an analog system is set up (under program control), with parameters that are the known coefficients of A; the manner in which the analog components are interconnected, however, simulates to within analog accuracy, the equation $A_*x = y$. There are many ways of structuring an analog system to behave in such a way -- roughly as many as there are iterative algorithms for linear equations. One way, which we have chosen, is based on the following approach:

Let D be a matrix consisting of the diagonal elements of A and of 0's everywhere else; and let N be the nondiagonal elements, i.e., $N = A - D$. Thus, the equation to be solved is

$$(N + D)x = y$$

or

$$Dx = y - Nx$$

or

$$x = D^{-1}(y - Nx) \quad .$$

Note that $\underline{D}^{-1}$ is known, by inspection of $\underline{A}$. The analog system is accordingly set up to simulate the last (implicit) set of equations shown in Fig. 3 (for a system of two equations). Naturally we have taken advantage here of the fact that, in linear systems, the solution errors are linearly related by the same equation, i.e.,

$$\underline{A}\,\underline{\Delta x} = \underline{\Delta y} \quad .$$

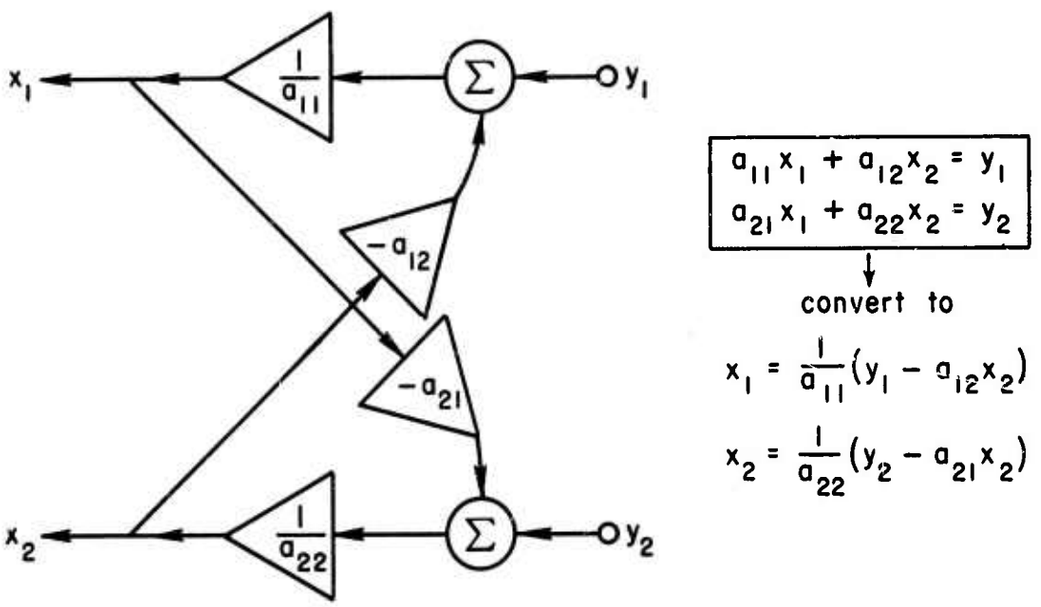


Fig. 3.
Inexact Analog Substructure.

As $\underline{x}_i$ approaches the solution, the error $\underline{y} - \underline{A}\underline{x}_i$ is computed exactly at a computing cost which grows as N^2 (for an $N \times N$ matrix $\underline{A}$). It is then converted (and amplified to a reasonable level) by a digital-to-analog converter. The computation of the pressure $\underline{\Delta x}_i$ is executed at a speed which depends, on one hand, on the response time of the scalers and summers of Fig. 3 and, on the other hand, on the eigenvalues of $\underline{A}$. Because of the continuity and asynchronousness of analog systems, this solution is reached very rapidly in a matter of, at most, a few milliseconds, for the typical case of several thousand equations. Thus, the over-all computing effort grows practically as N^2 per iteration, since the digital computation dominates. The number of iterations necessary for solution of the equation depends on the norm of the error $\underline{A} - \underline{A}_*$. For typical analog systems with errors in the vicinity of 1% to 2%, each iteration increases the accuracy of the solution by six bits ($\frac{1}{64} = 1.5\%$). This is because, as the digitally computed error becomes smaller, when it is converted to an analog quantity it is amplified so as to remain large with respect to the offset errors of the analog substructure. Thus, in five iterations, the solution is computed

to 30-bit accuracy. Since complete digital computer solution of the above equation grows as N^3, it follows that the savings in computing time are of the order of N, i.e., the above structure solves a 1000-equation system 1000 times faster than a strictly digital system with the same parameters.

During the reporting period, we have examined in considerable detail the attainable speed and accuracy of $\underline{A}_*$ under several different implementations.

V. HYBRID COMPUTATION: EXTENSIONS

The above structure can be extended to handle large systems of nonlinear equations $\underline{f}(\underline{x}) = \underline{y}$. In fact, any conventional iterative algorithm that uses a matrix correction, such as an inverse Jacobian gradient or a Newton-Raphson method, can be realized in this form. In this case, however, the coefficients of $\underline{A}^{-1}{}_*$ will vary from iteration to iteration as $\underline{x}_i$ varies.

Other extensions involve the solution of large systems of differential and integral equations. We have identified several such structures and are currently investigating them. In the course of our work, it has become apparent that, if a problem can be decomposed into two parts -- an "exact" subproblem which checks the accuracy of a proposed solution and an "inexact" subproblem which improves every proposed solution, and if the "checking" part requires less computation than the "improving" part -- then the over-all problem can be effectively solved by a hybrid computing structure.

It is not essential, however, that a problem be decomposed in two such parts. We are currently investigating decomposition into a larger number of analog and digital parts, each of which handles a relatively simple task such as addition, or evaluation of a function. These structures seem promising in that they "simulate" one-for-one the algebraic and differential operations of equations by corresponding electronic mechanisms.

Several additional questions have been raised in the course of our research, and these are currently under investigation. They involve the inherent limitations on computing speed among analog and digital systems, based on a small number of physics-oriented axioms such as limits on a maximum energy that can be stored per unit volume, a minimum energy that can be detected, and a maximum power density per unit area. It appears that the time required for addition of n quantities grows at least as log n, regardless of whether it is conducted by digital or analog devices; however, we have not yet been able to prove this from the basic axioms. Other questions concern the coding of information so as to maximize processing speed. For example, if the computation to be performed is simply arithmetical negation, that is, $y = -x$,

then a signed binary coding of x results in the fastest computation of y, since the energy that must be changed is the minimal detectable energy; an analog coding of x would require a larger energy change, hence a slower computing time. We have been able to prove, along these lines, that to any finite function $y = f(x)$, with x, y integers, there corresponds a coding of x and y (typically a highly redundant one) that changes only the minimal detectable energy, hence gives rise to "instant" computation.

In a more practical vein, we are currently initiating a study of the applicability of our hybrid structures to meteorological, electromagnetic-field and space-navigational problems which are very time-consuming or nearly impossible by conventional machines. We are also starting to look at the possibility of using inexact digital rather than analog processes in our computing structures. Roughly speaking, we are trying to establish whether continuity or inexactness is the significant property that makes the hybrid structures compute so rapidly.

Published Material

1. M. L. Dertouzos, "Educational Uses of On-Line Circuit Design", IEEE Transactions on Education, Vol. E-12, No. 3, September 1969, pp. 197-198.

2. M. L. Dertouzos, "Computer Science Education in the Soviet Union", IEEE Spectrum, Vol. 6, No. 12, December 1969, p. 74.

3. M. L. Dertouzos, "Character Generation from Resistive Storage of Time Derivatives", AFIPS Conference Proceedings (Fall Joint Computer Conference) Vol. 35, November 1969, pp. 561-568.

4. M. L. Dertouzos, "Computer Analysis of Nonlinear Networks by Recursive Decomposition", Proceedings, Second Biennial Cornell Electrical Engineering Conference on Computerized Electronics, Vol. 2, August 1969, pp. 57-67.

5. M. L. Dertouzos, "A Visit to the Soviet Union", Aug. 1969, Report to the National Academy of Sciences, National Science Foundation and IEEE.

6. G. P. Jessel and J. R. Stinger, "CIRCAL-2: A General Approach to On-Line Circuit Analysis", 1969 NEREM Record, Vol. 11, pp. 30-31.

to 30-bit accuracy. Since complete digital computer solution of the above equation grows as N^3, it follows that the savings in computing time are of the order of N, i.e., the above structure solves a 1000-equation system 1000 times faster than a strictly digital system with the same parameters.

During the reporting period, we have examined in considerable detail the attainable speed and accuracy of $\underline{A}_*$ under several different implementations.

V. HYBRID COMPUTATION: EXTENSIONS

The above structure can be extended to handle large systems of nonlinear equations $\underline{f}(\underline{x}) = \underline{y}$. In fact, any conventional iterative algorithm that uses a matrix correction, such as an inverse Jacobian gradient or a Newton-Raphson method, can be realized in this form. In this case, however, the coefficients of $\underline{A}^{-1}{}_*$ will vary from iteration to iteration as $\underline{x}_i$ varies.

Other extensions involve the solution of large systems of differential and integral equations. We have identified several such structures and are currently investigating them. In the course of our work, it has become apparent that, if a problem can be decomposed into two parts -- an "exact" subproblem which checks the accuracy of a proposed solution and an "inexact" subproblem which improves every proposed solution, and if the "checking" part requires less computation than the "improving" part -- then the over-all problem can be effectively solved by a hybrid computing structure.

It is not essential, however, that a problem be decomposed in two such parts. We are currently investigating decomposition into a larger number of analog and digital parts, each of which handles a relatively simple task such as addition, or evaluation of a function. These structures seem promising in that they "simulate" one-for-one the algebraic and differential operations of equations by corresponding electronic mechanisms.

Several additional questions have been raised in the course of our research, and these are currently under investigation. They involve the inherent limitations on computing speed among analog and digital systems, based on a small number of physics-oriented axioms such as limits on a maximum energy that can be stored per unit volume, a minimum energy that can be detected, and a maximum power density per unit area. It appears that the time required for addition of n quantities grows at least as log n, regardless of whether it is conducted by digital or analog devices; however, we have not yet been able to prove this from the basic axioms. Other questions concern the coding of information so as to maximize processing speed. For example, if the computation to be performed is simply arithmetical negation, that is, $y = -x$,

APPENDIX A

PROJECT MAC TECHNICAL REPORTS*

† TR-1 Bobrow, Daniel G.
Natural Language Input for a Computer Problem Solving System
September 1964 AD-604-730

† TR-2 Raphael, Bertram
SIR: A Computer Program for Semantic Information Retrieval
June 1964 AD-608-499

TR-3 Corbató, Fernando J.
System Requirements for Multiple-Access, Time-Shared Computers
May 1964 AD-608-501

† TR-4 Ross, Douglas T. and Clarence G. Feldmann
Verbal and Graphical Language for the AED System: A Progress Report
May 6, 1964 AD-604-678

† TR-6 Biggs, John M. and Robert D. Logcher
STRESS: A Problem-Oriented Language for Structural Engineering
May 6, 1964 AD-604-679

† TR-7 Weizenbaum, Joseph
OPL-1: An Open Ended Programming System Within CTSS
April 30, 1964 AD-604-680

† TR-8 Greenberger, Martin
The OPS-1 Manual
May 1964 AD-604-681

† TR-11 Dennis, Jack B.
Program Structure in a Multi-Access Computer
May 1964 AD-608-500

TR-12 Fano, Robert M.
The MAC System: A Progress Report
October 9, 1964 AD-609-296

† TR-13 Greenberger, Martin
A New Methodology for Computer Simulation
October 19, 1964 AD-609-288

TR-14 Roos, Daniel
Use of CTSS in a Teaching Environment
November 1964 AD-661-807

TR-16 Saltzer, Jerome H.
CTSS Technical Notes
March 1965
AD-612-702

TR-17 Samuel, Arthur L.
Time-Sharing on a Multiconsole Computer
March 1965
AD-462-158

† TR-18 Scherr, Allan Lee (Thesis)
An Analysis of Time-Shared Computer Systems
June 1965
AD-470-715

TR-19 Russo, Francis John (Thesis)
A Heuristic Approach to Alternate Routing in a Job Shop
June 1965
AD-474-018

TR-20 Wantman, Mayer Elihu (Thesis)
CALCULAID: An On-Line System for Algebraic Computation and Analysis
September 15, 1965
AD-474-019

TR-21 Denning, Peter James (Thesis)
Queueing Models for File Memory Operation
October 1965
AD-624-943

† TR-22 Greenberger, Martin
The Priority Problem
November 1965
AD-625-728

TR-23 Dennis, Jack B. and Earl C. Van Horn
Programming Semantics for Multiprogrammed Computations
December 1965
AD-627-537

† TR-24 Kaplow, Roy, Stephen Strong and John Brackett
MAP: A System for On-Line Mathematical Analysis
January 1966
AD-476-443

TR-25 Stratton, William David (Thesis)
Investigation of an Analog Technique to Decrease Pen-Tracking Time in Computer Displays
March 7, 1966
AD-631-386

TR-26 Cheek, Thomas Burrell (Thesis)
Design of a Low-Cost Character Generator for Remote Computer Displays
March 8, 1966
AD-631-269

TR-27 Edwards, Daniel James
OCAS - On-Line Cryptanalytic Aid System
May 1966
AD-633-678

TR-28 Smith, Arthur Anshel (Thesis)
Input/Output in Time-Shared, Segmented, Multi-processor Systems
June 1966
AD-637-215

TR-29 Ivie, Evan Leon (Thesis)
Search Procedures Based on Measures of Relatedness Between Documents
June 1966
AD-636-275

TR-30 Saltzer, Jerome Howard (Thesis)
Traffic Control in a Multiplexed Computer System
July 1966
AD-635-966

TR-31 Smith, Donald L. (Thesis)
Models and Data Structures for Digital Logic Simulation
August 1966
AD-637-192

TR-32 Teitelman, Warren (Thesis)
PILOT: A Step Toward Man-Computer Symbiosis
September 1966
AD-638-446

TR-33 Norton, Lewis M. (Thesis)
ADEPT - A Heuristic Program for Proving Theorems of Group Theory
October 1966
AD-645-660

TR-34 Van Horn, Earl C. (Thesis)
Computer Design for Asynchronously Reproducible Multiprocessing
November 1966
AD-650-407

TR-35 Fenichel, Robert R. (Thesis)
An On-Line System for Algebraic Manipulation
December 1966
AD-657-282

† TR-36 Martin, William A. (Thesis)
Symbolic Mathematical Laboratory
January 1967
AD-657-283

TR-37 Guzman-Arenas, Adolfo (Thesis)
Some Aspects of Pattern Recognition by Computer
February 1967
AD-656-041

APPENDIX A

TR-38 Rosenberg, Ronald C., Daniel W. Kennedy and Roger A. Humphrey
A Low-Cost Output Terminal for Time-Shared Computers
March 1967 AD-662-027

TR-39 Forte, Allen
Syntax-Based Analytic Reading of Musical Scores
April 1967 AD-661-806

TR-40 Miller, James R.
On-Line Analysis for Social Scientists
May 1967 AD-668-009

TR-41 Coons, Steven A.
Surfaces for Computer-Aided Design of Space Forms
June 1967 AD-663-504

TR-42 Liu, Chung L., Gabriel D. Chang and Richard E. Marks
Design and Implementation of a Table-Driven Compiler System
July 1967 AD-668-960

TR-43 Wilde, Daniel U. (Thesis)
Program Analysis by Digital Computer
August 1967 AD-662-224

TR-44 Gorry, G. Anthony (Thesis)
A System for Computer-Aided Diagnosis
September 1967 AD-662-665

TR-45 Leal-Cantu, Nestor (Thesis)
On the Simulation of Dynamic Systems with Lumped Parameters and Time Displays
October 1967 AD-663-502

TR-46 Alsop, Joseph W. (Thesis)
A Canonic Translator
November 1967 AD-663-503

† TR-47 Moses, Joel (Thesis)
Symbolic Integration
December 1967 AD-662-666

TR-48 Jones, Malcolm M. (Thesis)
Incremental Simulation on a Time-Shared Computer
January 1968 AD-662-225

TR-49 Luconi, Fred L. (Thesis)
Asynchronous Computational Structures
February 1968 AD-677-602

† TR-50 Denning, Peter J. (Thesis)
Resource Allocation in Multiprocess Computer Systems
May 1968 AD-675-554

† TR-51 Charniak, Eugene (Thesis)
CARPS, a Program which Solves Calculus Word Problems
July 1968 AD-673-670

TR-52 Deitel, Harvey M. (Thesis)
Absentee Computations in a Multiple-Access Computer System
August 1968 AD-684-738

TR-53 Slutz, Donald R. (Thesis)
The Flow Graph Schemata Model of Parallel Computation
September 1968 AD-683-393

TR-54 Grochow, Jerrold M. (Thesis)
The Graphic Display as an Aid in the Monitoring of a Time-Shared Computer System
October 1968 AD-689-468

TR-55 Rappaport, Robert L. (Thesis)
Implementing Multi-Process Primitives in a Multiplexed Computer System
November 1968 AD-689-469

† TR-56 Thornhill, D. E., R. H. Stotz, D. T. Ross and J. E. Ward (ESL-R-356)
An Integrated Hardware-Software System for Computer Graphics in Time-Sharing
December 1968 AD-685-202

TR-57 Morris, James H. (Thesis)
Lambda-Calculus Models of Programming Languages
December 1968 AD-683-394

TR-58 Greenbaum, Howard J. (Thesis)
A Simulator of Multiple Interactive Users to Drive a Time-Shared Computer System
January 1969 AD-686-988

TR-59 Guzman, Adolfo (Thesis)
Computer Recognition of Three-Dimensional Objects in a Visual Scene
December 1968
AD-692-200

† TR-60 Ledgard, Henry F. (Thesis)
A Formal System for Defining the Syntax and Semantics of Computer Languages
April 1969
AD-689-305

TR-61 Baecker, Ronald M. (Thesis)
Interactive Computer-Mediated Animation
June 1969
AD-690-887

† TR-62 Tillman, Coyt C. (ESL-R-395)
EPS: An Interactive System for Solving Elliptic Boundary-Value Problems with Facilities for Data Manipulation and General-Purpose Computation
June 1969
AD-692-462

TR-63 Brackett, John W., Michael Hammer, and Daniel E. Thornhill
Case Study in Interactive Graphics Programming: A Circuit Drawing and Editing Program for Use with a Storage-Tube Display Terminal
October 1969
AD-699-930

† TR-64 Rodriguez, Jorge E. (Thesis) (ESL-R-398)
A Graph Model for Parallel Computations
September 1969
AD-697-759

† TR-65 DeRemer, Franklin L. (Thesis)
Practical Translators for LR(k) Languages
October 1969
AD-699-501

TR-66 Beyer, Wendell T. (Thesis)
Recognition of Topological Invariants by Iterative Arrays
October 1969
AD-699-502

† TR-67 Vanderbilt, Dean H. (Thesis)
Controlled Information Sharing in a Computer Utility
October 1969
AD-699-503

† TR-68 Selwyn, Lee L. (Thesis)
Economies of Scale in Computer Use: Initial Tests and Implications for the Computer Utility
June 1970
AD-710-011

† TR-69 Gertz, Jeffrey L. (Thesis)
Hierarchical Associative Memories for Parallel Computation
June 1970 AD-711-091

† TR-70 Fillat, Andrew I. and Leslie A. Kraning (Thesis)
Generalized Organization of Large Data-Bases: A Set-Theoretic Approach to Relations
June 1970 AD-711-060

† TR-71 Fiasconaro, James G. (Thesis)
A Computer-Controlled Graphical Display Processor
June 1970 AD-710-479

† TR-72 Patil, Suhas S. (Thesis)
Coordination of Asynchronous Events
June 1970 AD-711-763

* * * * * *

TECHNICAL MEMORANDA ‡

† TM-10 Jackson, James N.
Interactive Design Coordination for the Building Industry
June 1970 AD-708-400

† TM-11 Ward, Philip W.
Description and Flow Chart of the PDP-7/9 Communication Package
July 1970 AD-711-379

* * * * * *

† Project MAC Progress Report I
to July 1964 AD-465-088

Project MAC Progress Report II
July 1964-July 1965 AD-629-494

† Project MAC Progress Report III
July 1965-July 1966 AD-648-346

Project MAC Progress Report IV
July 1966-July 1967 AD-681-342

APPENDIX A

Project MAC Progress Report V July 1967-July 1968	AD-687-770
Project MAC Progress Report VI July 1968-July 1969	AD-705-434

---

* Copies of all MAC reports listed in Appendix A, as well as earlier Progress Reports, have been deposited with DDC; using the appended AD number, a report may be secured from the National Technical Information Service, Operations Division, Springfield, Virginia, 22151. The prices from NTIS are: microfilm $0.95; hard copies: reports more than two years old $6.00, all others are $3.00 except TR-83 which is also $6.00.

† Out-of-print, may be obtained from NTIS (see above).

‡ All TMs have been deposited with DDC and are available only from NTIS, using the AD number appended; the cost is $0.95 for microfilm and $3.00 for hard copy.