MIT/LCS/TR-208

A MINICOMPUTER NETWORK SIMULATION SYSTEM

Brock C. Krizan

MIT/LCS/TR-208

A MINICOMPUTER NETWORK SIMULATION SYSTEM

Brock Collins Krizan

September 1977

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

A MINICOMPUTER NETWORK SIMULATION SYSTEM

by

Brock Collins Krizan

Submitted to the Department of Electrical Engineering on June 25, 1977 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science in Electrical Engineering.

ABSTRACT

The design, development and use of cost-effective computer networks require information about system behavior given a variety of network structures and operational policies. Because computer networks are complex systems whose behavior is generally not intuitively understood, there is a need for system analysis tools to provide a wide range of performance information.

This thesis describes a simulation system that generates behavioral information for a class of minicomputer network systems. This simulation system is modularly designed with modules for network modelling, specification of the network processing load, and simulation (a discrete event simulator). The network modelling done with the simulation system is based on a general purpose discrete modelling discipline. Flexible network model building blocks made from the basic modelling discipline structures are provided to the simulation system user. To prepare a simulation experiment the user assembles a network model from the building blocks and specifies a network processing load. To generate performance information the network model and load specification are input to the simulator along with simulation control parameters. On completion of the simulation experiment the generated performance information is output in a palatable form to the user. Overall this simulation system is a convenient and flexible system analysis tool for minicomputer networks.

Thesis Supervisor (Academic): Liba Svobodova
Title: Assistant Professor of Electrical Engineering
and Computer Science

Thesis Supervisor (VI-A Company): William Gimple
Title: Section Manager, Hewlett-Packard General
Systems Division

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

The author wishes to express his appreciation to the many people who have assisted in the creation of this thesis. First of all, thanks go to Hewlett-Packard and its employees; to Jim Cockrum, Bill Gimple, Eric Ha and John Hawkes for their managerial and technical support, and to John Williams for his help in producing a polished copy of this thesis. Special thanks go to Liba Svobodova for her invaluable advice and encouragement, provided from inception of the thesis project to completion of the thesis document. Appreciation is extended to John Tucker for his enlightened management of the 6A program which made possible the author's engineering internship at Hewlett-Packard. And finally, thanks go to Karen Doyle for her personal support during the many months of work which went into this thesis.

CHAPTER 1

A network is an interconnected or interrelated group or system. Network systems have been developed for railroads, telephone communications, mail services and electric power. In each case the development of a network from previously independent components has brought improved service at lower cost.

Based on this tradition of success in the development of network systems, it is not surprising to see a worldwide move towards the use of data processing-oriented communications networks (ie. computer networks). These networks are generally formed by the interconnection, via communications links, of computer systems. Among the goals in the move to develop computer networks are to provide information security, processing reliability and cost-effective processing services.

The physical disribution of processing services in a computer network can be exploited in providing the appropriate degree of information security to interacting processing tasks with different security requirements. One way to do this is to customize each node in a network to provide a different level of security (in terms of operating system safeguards, physical protection of the computer, etc.). Then processing tasks can be assigned to the computer system with the appropriate security and at the same time retain a communications capability to tasks with other security requirements. The network approach to security can be more effective than placing all tasks within the security structure of a single computer.

The network approach to reliability is based on the redundant processing capability inherent in a computer network. If one computer system in a network fails, the other systems in the network can be used to dynamically recover from the failure. Optimally, the processing load of the system that failed can be shifted without penalty to other systems and the globally observed operation of the network is unchanged. In most cases though, it must be expected that a degradation of network performance will occur after a single system failure. This gradual degradation of services, made possible through networking, is one way to avoid abrupt and total discontinuations of service (ie. when the processor fails in a computer system with no backup processor).

Perhaps the most important feature of computer networks is the potential to provide cost-effective processing services. There are a number of ways to realize this potential. Sharing of hardware, software and information resources can be done with a network of computers of different characteristics. A user can access costly resources without having those resources associated with the locally available computer system (eg. an expensive peripheral need be located on only one system in a network). Another way a network can be cost-effective is by allowing local data processing at the site of data acquisition and use, while at the same time maintaining data links to remote processing centers (eg. a bank with many branches would be a potential network application). This arrangement yields significant savings in communications costs over a completely centralized processing facility. Yet another cost-effective characteristic of networks is the allowance for a gradual and consistent expansion of the processing capabilities as they are required by an application. Networking can offer a range of processing capabilities more expansive than is offered by any single computer line.

While computer networks have great potential in providing a variety of data processing capabilities and services, they also pose difficult design and implementation problems. A network system is inherently more complex than the individual components that are linked together in the system. In so far as the behavior of a computer system is difficult to predict for a given user environment, prediction of the behavior of a computer network system can be mind boggling. But this behavior must be anticipated to some extent in order to develop effective data processing systems.

There are two primary phases in the development of computer networks. First, communications hardware and software must be developed to allow a computer system to function in a network environment. Modems, communications processors, programs to maintain communications protocols, and a host of other computer communications components must be designed and implemented. The second phase of network development involves selecting the appropriate computer and communications options in order to fill specific data processing needs. During both phases a network designer needs insight into the behavior of network systems.

System analysis tools provide information to supplement a designer's intuitive understanding of network behavior. These tools include hardware and software monitors for system measurement, analytic models, and system simulators. Most analysis tools are restricted in their application to particular aspects of network behavior. The complexity of network systems, coupled with the inherent limitations of the analysis tools cause this restriction. Therefore, in order to cope with new types of networks, there is a continuing need for new network analysis tools.

This thesis describes the development of a simulation system to be used for analysis of the behavior of a minicomputer network system. The following chapters trace the development of the Minicomputer Network Simulation System (MNSS). Chapters 2,3 and 4 review the research into, and the design of a flexible network model of a particular class of minicomputer system (designated Hewlett-Packard Computer systems or HPC systems). Chapters 5 and 6 describe the design and implementation of an effective interactive simulator that uses HPC network models. Chapter 7 provides a summary of some simulation experiments designed to demonstrate the capabilities of MNSS. Chapter 8 reviews the progress made by this thesis project and the potential for further work.

CHAPTER 2

BACKGROUND

There is a rich background of work concerned with computer system analysis. From this background are drawn the techniques that are used to develop computer network analysis tools. These tools include analytical models, performance monitors, and simulation systems. Analytical models are usually very efficient but are limited in application to relatively simple, well characterized systems. The use of performance monitors in network analysis is aimed at getting very accurate and specific information. A fundamental limitation of performance monitors is the necessity to have available a working version of the system to be analyzed. Simulation systems can be used to do system analysis at any stage in system development. In addition, a network system can be analyzed in as much detail as is necessary. The efficiency of a simulation system in analyzing network behavior is in general lower than that of an analytical model and the behavioral information derived with simulation is less accurate than that gathered by a performance monitor. Simulation systems, performance monitors, and analytic models, by having different operational characteristics, provide the capabilities for a range of computer network analysis at various stages in system development and use.

The following sections review the most pertinent background to the development of MNSS. This includes simulation modelling techniques and the implementation and use of computer network simulators. The principles behind the design of simulation systems in general, and MNSS in particular, can be distilled from the sea of work that has been reported. This work can be characterized into two groups corresponding to the principle stages in the MNSS development process. Initially models of minicomputer network systems had to be designed to describe all significant features. Then an effective simulator had to be designed and implemented that uses these network models.

2.1 Simulation Modelling

A representative model of a system is required to do simulation. Models used for simulation can be divided into two general classes: structural models and functional models [27]. Individual system components and their connections are represented in a structural model. This level of detail is appropriate for tasks such as logic simulation of digital systems [5]. A functional model provides a more abstract representaion of a system than a structural model. It describes how a system operates and can be used for mathematical or empirical system analysis. Extensive use of functional models, in the form of flowchart models [2], finite-state models [10], parallel nets [22] and queuing models [23], is made in simulating complex hardware and software computer system structures.

The effectiveness of a simulation model can be gauged by its ability to represent all significant system features and at the same time minimize the effort required to use it for system simulation. In the process of constructing a simulation model there are a number of factors that impact its effectiveness. These factors involve characteristics of the modelling language used to describe the model, and various qualities of the model. Specifically they include:

1. the flexibility of the modelling language (ie. the model description language) in abstracting real system structures,
2. the efficiency of the modelling language in a simulation context,
3. the ease with which a model can be adapted to changes in the structure of the system modelled,
4. the level of detail that is used in a model to represent significant system features (that may be at different levels of detail in different parts of the system structure), and
5. the effort required to verify and validate the model.

Work is continually being done on modelling languages and methods in order to deal with these factors.

A number of approaches have evolved for developing effective modelling languages. One approach is to have the modelling language be identical to the simulator programming language (ie. the programming language used to implement the mechanism for simulating the model) [11]. While this approach can yield efficient simulation processing, the process of abstracting a system in this type of modelling language may be very difficult. An ever expanding set of instructions may be needed to handle new and evolving system structures. A second approach to developing a modelling language is to have only a partial correspondence with the simulator programming language (eg. using a simulation language such as SIMULA or GASP to implement a modelling language) [16]. With this approach the simulator programming language performs simulation housekeeping chores and well defined activities such as table construction. System representaion in the modelling language can be done with more ease than in the first approach, but the conversion to the simulator language is more complex. Another approach is to use a general purpose language (FORTRAN, APL) [23] to implement the simulator, and have a separate modelling language which specifically deals with the structures of the system to be modelled. Representing system structures in the modelling language becomes increasingly straightforward at the cost of more effort to implement the simulator.

Whatever the relationship between the modelling language and the simulator language, it is advantageous to have the modelling language provide convenient model description building blocks [7]. The building blocks may vary from the 56 instructions in GPSS-5 to the two principle structures in DYNAMO. By having an effective set of building blocks a modelling language can be used to represent a range of different types of systems; a modeller can be spared the task of mastering several limited (special purpose) languages. In addition a model constructed using a well known set of building blocks can be understood by more people than one constructed using specialized, one-of-a-kind structures.

The complexity of a modelling language can be gauged by the number and types of the associated building blocks. In general a modelling language becomes more difficult to use and understand with larger numbers of building blocks. The degree of correspondence between building blocks and real system structures also influences the complexity of language use. A modelling language for a limited and well defined set of systems can incorporate just a few very specific building blocks and be very easy to use. But a modelling language used for general classes of systems must either have a great number of specific building blocks or a smaller number that are very general. In either case the generalized modelling language is harder to use than a specific modelling language.

Given a particular modelling language, the effectiveness of a model is affected by the technique of the modeller. Modelling techniques that promote the development of effective models include top-down development, modular design, and submodel organization.

Top-down development is extensively used in software development to facilitate debugging, functional verification, and maintenance. It is not surprising that top-down development has been constructively applied to simulation modelling [6]. Design and verification activities can be integrated together during top-down implementation of a simuation model. Since the conceptual model is usually derived in a top-down manner, this is a natural and efficient approach.

Modular design allows a modeller to adapt a model to changes in a system structure by making localized changes to the model. The model can be effectively used in simulations in which both parametric and structural system characteristics vary [24]. Parametric characteristics are subject to variation by changing single, usually numeric values; structural characteristics involve functional aspects of a system and are varied by changing the functional modules that make up the model description. For example, with a modularly designed network simulation model, not only can the line speeds and message frequencies be varied (parametric characteristics), but also the network configuration, communications protocols and processor scheduling algorithms (structural characteristics) can be changed. Modular design makes a model easier to develop and adaptable to structural variation for simulation studies.

Submodel organization is a hign level application of modular design (several modules may be included in a submodel) which is facilitated by the decomposition of a system into several system components or subsystems. The simulation of a system is restricted by time and cost limitations when a model is developed at a very fine level (low level) of detail. By using a submodel organization, different levels of detail can be used in individual submodels to ease simulation restrictions while maintaining the accuracy of the model representation [21]. Submodel representaions may range from a decision table to a highly detailed model (subject to simulation apart from the rest of the model). By using a decision table or a simple function to directly generate submodel outputs from the inputs, then analytical results, measurement data, and previous simulation results can be utilized to reduce needless simulation. Where a detailed representaion of a subsystem is needed for modelling accuracy, simulation on the desired low level of detail can be limited to the specific submodel. Excessive simulation is avoided by not having to model all subsystems at the same low level of detail.

A submodel organization can also be exploited in model validation [3]. Each submodel can be exercised and compared to the corresponding real subsystem. The submodels will be less complex than the complete model and easier to validate. Once each submodel is shown to operate correctly then submodel interactions can be proven correct (or redefined to become correct). This stepwise approach to validation avoids many of the problems of working with large and very complex models. It is especially valuable where there is a model comprised of many duplicate submodels (eg. a homogeneous network model with each node being a submodel).

2.2 Simulation System Design

A computer simulation system provides facilities to build a simulation model, takes as input a workload description, and uses a simulator mechanism to perform experiments. The simulation model and workload description determine the behavioral information that is generated and recorded for performance analysis by the simulator. Once a simulation model or the building blocks necessary to form a set of related simulation models have been developed, the next steps in the development of a simulation system are the design and implementaion of an efficient simulator and a comfortable user-simulator interface.

Computer network simulators can be classified according to the form of the workload description they use; among the most common classes are stochastic and trace driven [27]. For a stochastic simulator the workload is described by probabilistic distributions; resource demands are generated as random variables from these distributions. In contrast, a trace driven simulator operates with a workload represented as a deterministic sequence of resource demands. Trace driven simulation is very useful in tuning a system for well defined applications. It cannot be used for an application that cannot be represented accurately by a deterministic sequence of resource demands or where exact resource demand information is not available; in these cases stochastic simulation may be appropriate. Both types of simulation can be used to examine the performance of new system designs, alternative system configurations and resource management strategies.

An efficient performance analysis simulator generally simulates only those events that change the system state. Such simulators, called discrete event simulators, jump from event to event in simulated time. The length of the jumps (ie. the real time that would elapse between two events) does not effect the processing required to do a simulation; rather the total number of events is the key factor. Clearly the efficiency of the simulator is directly related to the level of detail of the simulation model. Increased detail is bought at the cost of more events being generated during simulation, and concequently more simulation processing.

MacDougall's BASYS simulator [19] for a disk-based multiprogrammed computer system is a prime example of a discrete event simulator. It does stochastic simulations and is based on a queuing model. Events correspond to the assignment of a job to the CPU, the release of the CPU to wait for the completion of an I/O request, etc. Handling of the events is implemented in the simulator as event routines. Each event routine performs the actions which correspond to the associated event and also predicts followup events (ie. schedules event routine executions). Event handling is maintained by the simulator through the use of an event list (that reflects the time sequence of events yet to occur), queue information structures, and a job table. Though the BASYS simulator is used to simulate a simple system, its operating principles can be adapted to do more complex simulations.

The character of the user-simulator interface often plays a large part in determining the effectiveness of a simulator. Given an efficient simulator and an accurate simulation model, the user should be able to easily set up, run, and get back the results of simulation experiments. This dictates a flexible interactive environment, not batch processing. The Computer Networks Simulation System developed at the University of Waterloo [14] provides an example of a comfortable user-simulator interface. Network topology and traffic characteristics are input via a conversational dialog. Simulation output is available with or without data analysis. In addition, message delay information can be displayed during the simulation run.

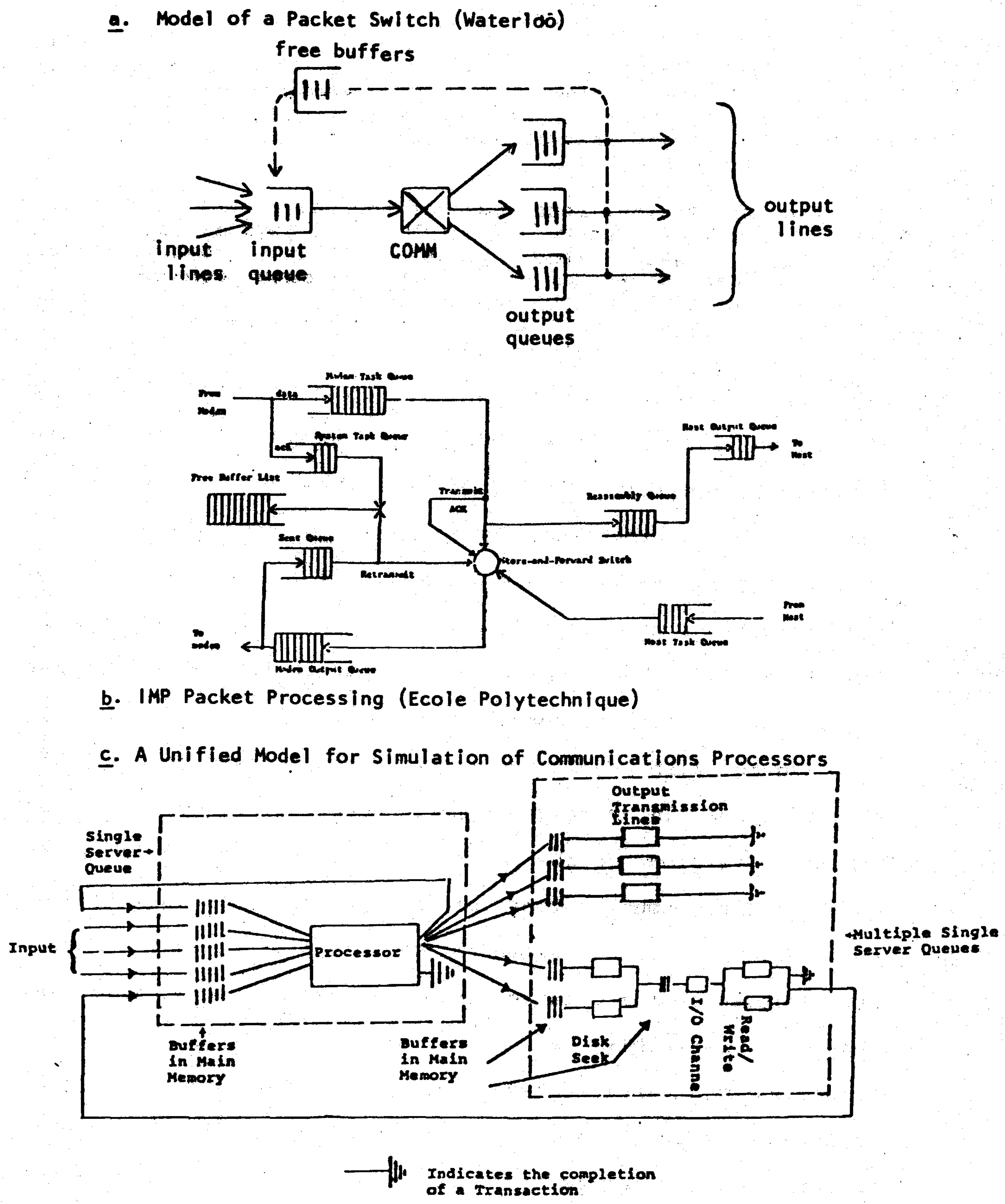
2.3 Computer Network Simulation System Examples

One way to review computer network simulation systems is in terms of the degree to which network components are represented in the corresponding simulation model. A simulation model may focus on a particular component of a computer network (eg. the communications links) and represent the rest of the network system at a very high level. A simulation system that incorporates such a simulation model is used to examine component behavior as opposed to overall network behvior. A simulation system that is used to examine overall network behavior requires a simulation model that represents all network components at some non-trivial level. An important simulation system design decision is to decide how completely the system is to be simulated.

Many simulation systems have been developed that focus on the communications subnetwork in a packet switching network [7,12,13,26]. The communications subnetwork includes a number of dedicated communications processors interconnected by communications links. Each communications processor receives messages from one or more attached host devices (computers, terminal concentrators, etc.), routes messages through the communications subnetwork, and delivers messages to attached host devices when appropriate. The simulation systems that simulate this activity use simulation models that range in detail in representing the structures (eg. communications links, communications processors) of the communications subnetwork. The contrast between several simulation models is shown in Figure 2.1.

The simulation model used at the University of Waterloo to study CIGALE (the packet-switching communications subnetwork of the CYCLADES computer network) is illustrated in Figure 2.1a. The objective of this study was to observe the behavior of CIGALE under various traffic conditions. To do this line speeds, line delays and buffer utilization were accurately modelled. The lines were assumed to be error free and the communications processor service time (to do packet routing and buffer handling) was modelled as a constant. These simplifications give an indication of how insignificant detail (in light of the modeller's objectives) is kept out of the simulation model representation.

Figure 2.1b shows a simulation model, used at the Ecole Polytechnique in Montreal [12], which can be considered an enhanced version of the Waterloo model. A more sophisticated host device-communications processor interface is included, representing message segmentation into packets (eg. messages are allowed that are larger than the maximum packet size) and message reassembly at the destination. Also packet transmission acknowledgement and retransmission in case of nonacknowledgement (eg. line errors or insufficient packet buffering at the destination) is represented in the simulation model. Overall, this simulation model can be used by a



a. Model of a Packet Switch (Waterloö)

b. IMP Packet Processing (Ecole Polytechnique)

c. A Unified Model for Simulation of Communications Processors

Figure 2.1 Communications Submetwork Models

simulation system to generate more detailed packet (message) traffic information than the Waterloo system.

A significant variation of the previously discussed simulation models is shown in Figure 2.1c. This simulation model, developed at the Network Analysis Corporation [7], is intended to be used in simulations of communications processors in general. The interesting feature of this simulation model is the detail used to represent packet handling (routing, buffer management). This detail can be contrasted with the constant service time for packet handling in the Waterloo model. In focusing attention on the operations of the communications processor as opposed to network message traffic, a detailed computer system representation for the communications processor was required.

A fundamentally different computer network simulation system was developed by Linsenmayer and Ligomenides [17]. Their simulation model combines a communications subnetwork model (such as those in Figure 2.1) and models of the computer systems attached to the communications subnetwork. In representing in some detail all major components in a computer network, the Linsenmayer model can be used to simulate computer networks as interdependent combinations of hardware/software elements and user job environments. In particular, studies are planned to examine various aspects of global job allocation in a computer network. The ultimate justification of a complete network simulation model, in light of its inefficiency relative to limited models (eg. those in Figure 2.1), is its usefullness in generating overall network behavior.

MNSS is used for studying the overall behavior of a class of network systems. As such, it is closest in design (functions and limitations) to the Linsenmayer and Ligomenides system.

CHAPTER 3

DEVELOPMENT OF A SIMULATION SYSTEM

The development of another simulation system is justified in light of the inadequacies of existing simulation systems in generating the behavior of constantly evolving network structures. Even where simulation systems accurately generate network behavior, there are often limitations in the user-simulation system interface. The success of MNSS as a network system analysis tool rests on its capability to provide a friendly user environment from which interesting network behavior can be studied. Each step in the MNSS design process was directed at achieving these goals.

This chapter outlines the development of MNSS and identifies those features that make the effort worthwhile. The primary reasons for MNSS development will be presented, followed by a description of the network system to be studied and a general overview of the design of the corresponding simulation system.

3.1 Motivation For The Development Of MNSS

The motivation for the development of MNSS is based on two primary desires: the desire to study the behavior of HPC network systems and the desire to build a ′better′ simulation system.

The design of a computer system (composed of one or more computers) is based to some extent on the anticipated behavior of the system in selected environments. The system design may be optimized for throughput, response time, security or some special requirement. In all but simple environments, the behavior of most computer systems is too complex to be derived by intuition and mental wizardry. HPC network systems are under development and there is a need for an analysis tool to study the complex network behavior generated by alternative designs. In the initial stages of development some of the areas of interest are protocol design and resource management. Once network structures have been developed, the performance of HPC networks of various configurations under realistic user loads has to be examined. This must be done to optimally customize HPC networks to user specifications.

Simulation was chosen as the system analysis technique because of its capabilities in generating behavioral information of complex systems at various stages in system development. The fundamental choice of simulation over measurement was based on practical considerations. While measurement is more accurate than simulation in deriving behavioral information, it cannot be used during early development when HPC networks do not exist in measureable form. In addition, there are non-trival logistical problems in trying to measure the performance of large network systems that operate with a variety of configurations and user loads.

The advantage of simulation versus analytical modelling is based on the limitations of mathematical models of complex sytems. A workable mathematical model of an HPC network system could not be developed to derive the scope of behavioral information that is required. In general, simulation is flexible, accurate, and efficient enough in generating HPC network system behavior, to be effective as an analysis tool.

As was indicated in Chapter 2 many simulation systems have been developed, with a wide range of capabilities, for the study of computer network systems. Despite this abundance, there do not seem to be any available simulation systems that could be used to analyze HPC network systems. This is due, in part, to the uniqueness of the HPC network architecture (described in Section 3.2.1). Another reason to develop a new simulation system is the inpracticality of adapting an existing simulation system to a new operating environment (potentially different programming languages, processing capabilities, and i/o facilities). A third compelling reason to develop MNSS is the need for a simulation system that is useable in a commercial environment. There is a need for a better simulation system than has been previously offered.

A ′better′ simulation system is one that is developed according to the principles reviewed in Chapter 2. Using those principles a simulation system can be made easy to use, flexible enough to perform a wide variety of simulation experiments, accurate in generating network system behavior, and cost/effective as an analysis tool. Any one of these features can be found in currently available simulation systems, but few systems integrate them all into an effective package. A goal of the MNSS development effort is to provide an integrated system in which all facets of the system reflect general design goals of flexibility, accuracy, ease of use, and efficiency. This applies to the simulation model, simulator, and user interface.

3.2 Development Of MNSS

MNSS was developed to incorporate features that are advantageous for a general purpose simulation system. The modelling discipline, simulator, and user interface are not fundamentally restricted to the simulation of any one class of computer systems. The primary use of MNSS though, is to simulate the behavior of HPC network systems. The HPC network architecture that is particularly suited to the formation of minicomputer networks. Therefore, ′Minicomputer′ Network Simulation System (MNSS) is an appropriate designation for the simulation system described in this thesis report. The following two sections present the general characteristics of the HPC network architecture and an overview of the MNSS implementation and use.

3.2.1 The HPC Network Architecture

There are two basic aspects to the HPC network architecture: 1. the architecture of the individual HPC systems, and 2. the way in which network communications is handled by an HPC system.

An HPC system supports a multiprogramming environment in which several independent programs can be executed concurrently. A program is made up of one or more processes; a process is composed of the software control structure for a particular execution of code. Code and data may be shared between processes within a program. All contention for system resources (eg. CPU, memory, etc.) occurs at the process level. The process control software uses a priority-ordered preemption scheme to arbitrate resource contention. Time-slicing is used to delegate usage of resources such as the CPU among processes of equal priority.

There are two basic types of processes: user processes and system processes. User processes are the principal agents of a user in getting work done on an HPC system. The priority level of user processes may vary, but typically it is fixed at a particular level. User processes can be associated with several forms of activity, including code execution (processing), short wait operations, long wait operations, and remote processing operations. The distinction between short wait operations (eg. disk i/o operations) and long wait operations (eg. terminal i/o) is made because long waits force a process to lose control of contested resources such as the CPU, whereas short waits do not. Remote processing operations relate to remote process activity that is initiated by a remote request. A process will make a remote request in order to do work on a remote system in an HPC network. This may include accessing a network database or running a program that works only on a particular system in a network. A user process 'waits' for a remote request to complete before it resumes local activities (ie. processing, wait activities).

A system process is generated by the HPC operating system to support user acitvities. Through system processes, users can utilize HPC hardware and software resources that cannot be accessed by user processes directly. The reasons for using system process intermediaries are to maintain the independent activities of concurrently existing processes, to act as agents for user process communication activities, and to relieve users of the complications of low level HPC operations. System processes are at higher priority levels than user processes and consequently preempt user processes in cases of resource contention.

In a network environment the system processes most often seen are associated with communications activities. These system processes handle communications request/response initiation and network store-and-forward operations. Once communications system processes gain CPU control (by waiting in line behind other system processes or by preempting a user process), they do processing, short waits, and link control operations to perform a communications function. There is no long wait activity and, due to the brief time it takes to perform a communications function, the communications system process is not preempted (by another system process) from CPU control.

HPC networks can be arbitrarily connected in locally or geographically distributed configurations. Store-and-forward facilities are used whenever information must be transferred between HPC systems that are not directly connected. The communications links in an HPC network may be half or full duplex, and can transfer information at a variety of baud rates (1200, 2400, ...). System processes maintain the protocols (eg. SDLC, X.25) that are used to control the links.

There are three types of information transferred through an HPC network: request messages, response messages, and link control messages. Request and response messages are generated by a system process for a user process. The transmission and routing of these messages is also handled by a system process. A request message is generated when a user process seeks to do work on a remote HPC system. When it arrives at the appropriate destination, a user process is started to do the requested work. When the work is completed, a response message is generated. This message is then sent back to restart the waiting initiator user process. Link control messages are used primarily to acknowledge the transmission of request/response messages and to aid recovery from link errors.

All communications overhead (eg. use of CPU and memory resources) is handled by the 'host' systems in an HPC network; there are no communications front end processors or network store-and-forward machines. System processes, which share resources with user processes, are used to do network communications. The performance of an HPC system is degraded for local activities in order to provide network communications capabilities. This is a reasonable tradeoff in a minicomputer network where an investment in dedicated communications hardware is not cost-effective. The HPC network architecture is, therefore, primarily an architecture for providing low cost communications services for a network composed of minicomputers.

3.2.2 MNSS Design And Use

The two key steps in the MNSS development effort were, first to formulate flexible models to represent HPC network systems, and then to build a simulation system using these models. The guidelines for the development effort were dictated by the goals set forth in Section 3.1, and reflected many of the principles discussed in Chapter 2.

MNSS models, which are representations of HPC network systems, are built up from a basic model of an HPC system.

Inherent in the HPC system model is an interactive relationship of the representations of processing and communications mechanisms. This relationship is essential to the HPC network architecture. Overall, MNSS models can be characterized as complete network models. This is true because all major network components (ie. local processing and network communications) and their interactions are represented non-trivally. MNSS models are described in a functional modelling language that has a small number of building blocks to represent HPC network system features.

MNSS is implementated on an HP3000 computer in SPL, an Algol-like programming language. MNSS incorporates a discrete-event simulator and an interactive interface. The simulator uses an SPL version of an MNSS model to generate HPC network behavior. The MNSS user interface is implemented as an interactive dialog. In many cases the user picks from a menu of alternatives in order to run a simulation experiment and get back results.

There is a wide range of behavioral information that can be generated by the simulator (due to a large extent to the completeness of MNSS models). The user can request the following results to be displayed by MNSS:

1. processing statistics broken down by node and process type (eg. system process, locally or remotely initiated system processes),

2. cumulative remote processing initiated by local processes at each node,
3. queuing statistics (eg. maximum length, mean length, maximum wait, mean wait) for user processes in the CPU wait queue,
4. queuing statistics for messages in the link wait queue,
5. CPU utilization statistics broken down by node and processor state (eg. system or user process control, idle, process handling overhead),
6. link utilization statistics broken down by link state (eg. idle, transmitting),
7. the number of user process launchings (intiate CPU control) per node, and
8. the number and average size of message transmissions per link.

This information gives a user the ability to do sophisticated studies of HPC network systems.

Chapter 4

MNSS MODELLING

The simulation of a system initially requires that a model of the system be conceived. MNSS utilizes a set of related models that are abstract representations of HPC network systems. The development of these models (refered to here as ′MNSS models′) was based on two general design criteria. First, the models had to be applicable for use in a simulation system. Essentially the cost, in terms of human and processing resources, to develop the simulation system and to run simulation experiments had to be minimized. This required reasonably high level models of HPC network systems and a straightforward procedure for the implementation of these models in a simulation framework. The second design criterion was that the models be made to contain the variables and relations that are significant in representing the behavior of HPC network systems. This design criterion was balanced against the first design criterion, insuring that important detail was not purged from the model in order to simplify simulation system development and use.

The effort to design and implement MNSS models ran within the framework established by the design criteria. The first step was to develop a modelling discipline. This involved the development of a modelling language and a technique to conveniently map ′real life′ system structures into the abstractions of the modelling language. The next step was to construct high level building blocks from modelling language structures that could then be used to build MNSS models. The structural similarities of HPC networks facilitates the use of these building blocks; this avoids starting from scratch in modelling each different network. The final step in the modelling effort was to develop a systematic procedure to map modelling structures into structures in the simulation system programming language.

4.1 The MNSS Modelling Discipline

The MNSS modelling discipline facilitates the development of functional models of systems. It represents a generalization of techniques used in finite-state modelling and queueing network modelling. MNSS modelling structures have close analogs in the structures of these modelling disciplines. The five basic MNSS model structures are entities, groups, group transitions, entity sources and entity sinks.

An ′entity′, in the context of the MNSS modelling discipline, is an object whose behavior is of interest. It could be a shopper in a supermarket model or a query in a data base model (note that the MNSS modelling discipline is general enough to be used to describe models of systems other than HPC computer networks). Associated with an entity is information about those entity characteristics that have a bearing on its behavior in the system to be modelled. Entities can be classified according to common characteristics into 'entity classes'. Entity classes are used as a convenient way to deal with entities (particularly the information associated with entities) durning the construction of a model.

A 'group' is a collection of entities showing a particular form of behavior; the behavior of an entity over time can be described by the sequence of groups it was in. A group can incorporate a variety of orderings on the entities (of one or more entity classes) that occupy it at any point in time, ranging from no ordering to a sophisticated queue ordering (eg. FIFO, according to entity class, based on priority information associated with an entity, etc.). For the supermarket model a group could be defined for the shoppers waiting to be checked out or the shoppers being checked out. The waiting line (a FIFO queue) is the ordering in the 'waiting' group.

Entities can come into existence in a model of a system in a number of ways. They may be defined to exist in a particular group when the model is initialized (in general as permanent entities in a closed system model); alternatively an entity may be created and injected into a group in the model from an 'entity source'. Associated with an entity source is information that identifies the class of entity to be created; this information that is used to generate entity characteristics. An entity can cease to exist in a model by being absorbed by an ´entity sink´. By using entity sources and entity sinks, models of open systems (ie. systems that interact with their environment) can be formulated.

A ´group transition´ is a path from one group to another group, from an entity source to a group, or from a group to an entity sink. Each group transition has associated with it an event routine. The event routine identifies the consequences of an entity leaving a group or entity source via the group transition. The consequences of a group transition may include provocation of other entities to undergo group transitions (immediately or at some future time), changes to group entity orderings, or modifications of entity information. For example, in the supermarket model a shopper leaving the ´being checked out´ group will cause: another shopper to do a group transition to that group from the ´waiting´ group, a group transition to be scheduled for the new occupant of the ´being checked out´ group (when all that shopper´s groceries have been checked out), and the ´waiting´ group to be reordered.

In order to provide a convenient mapping between a ´real life´ system and a model of that system, a symbolic notation has been defined for the MNSS modelling structures; this notation is illustrated in Figure 4.1. The presence of an entity class symbol within the symbol for a group, entity source or entity sink indicates that entities of that particular class can occupy the group, source or sink. One or

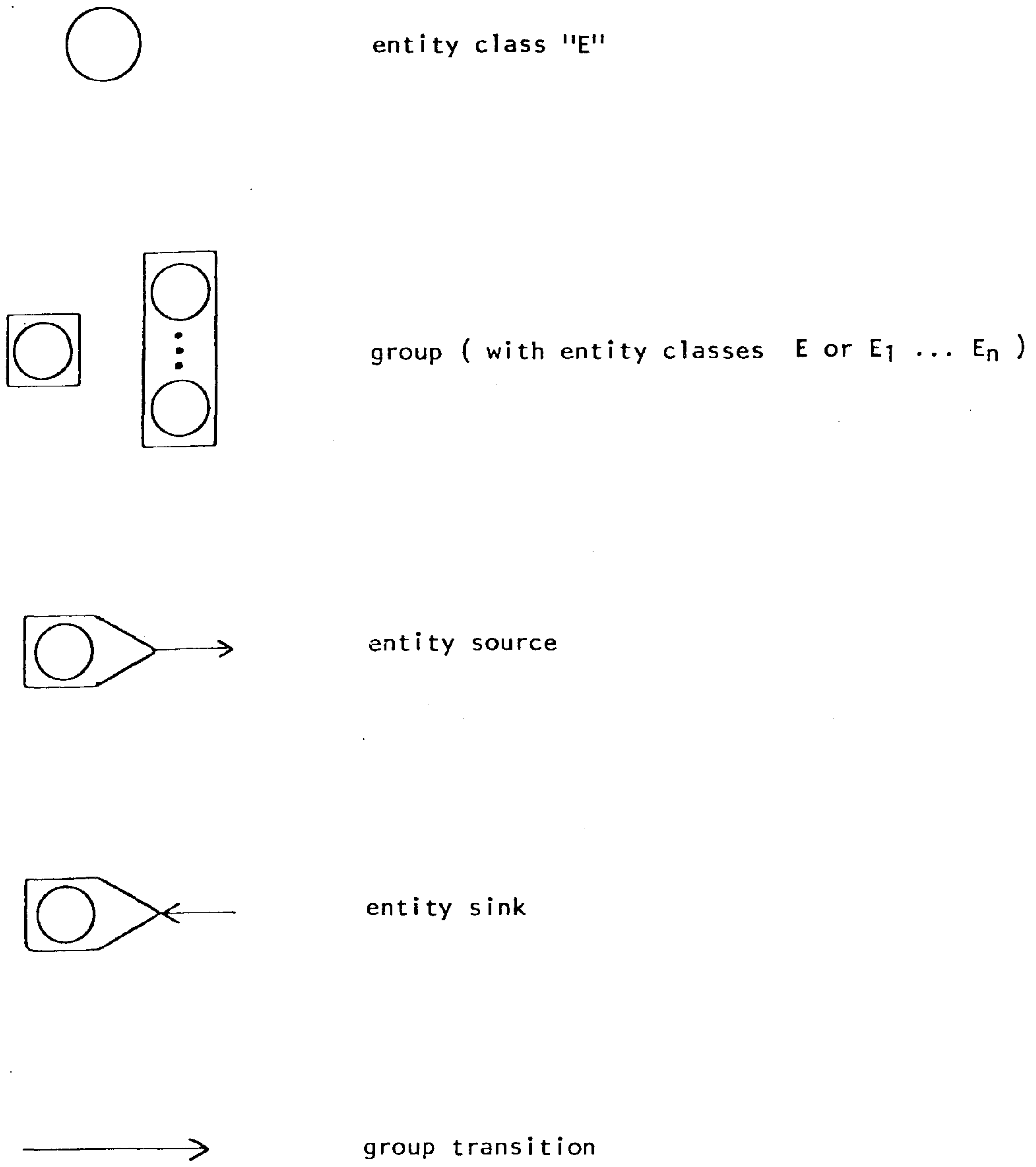
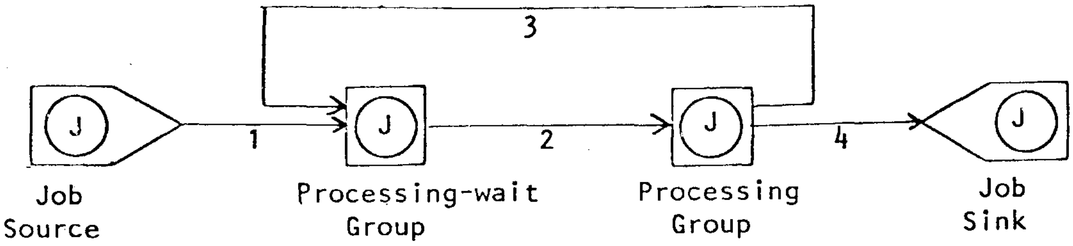


Figure 4.1 MNSS Modelling Structures

more entity classes can be associated with a group, source or sink. MNSS modelling structures (and the symbols shown in Figure 4.1) are connected together according to the following rules:

1. Groups can be connected together by a group transition as long as they have at least one entity class in common;
2. An entity source and a group can be connected by a group transition, leading from the source to the group, as long as they have at least one entity class in common;
3. A group and an entity sink can be connected by a group transition, leading from the group to the sink, as long as they have at least one entity class in common;
4. One or more transitions can lead to and from a group, from an entity source, or to an entity sink;
5. A path in a model established by the connection of groups associated with a particular entity class must be either a closed path or a path from an entity source to an entity sink.

A job processing model created using the MNSS notation is shown in Figure 4.2. A job (an entity in the model) comes from an external job source and goes into a FIFO queue ordered processing-wait group. A job is transferred to the processing



Group Transitions

1. job introduction
2. job processing begins
3. job processing suspended
4. job processing completes

Entity Class

Ⓙ Jobs

Figure 4.2 A Job Processing Model

group when it is at the head of the FIFO queue and the processing group is unoccupied by any other job. A job stays in the processing group until it either completes the processing it has to do, in which case it is absorbed by the job sink, or it uses a system allocated amount of processing time (ie. occupies the processing group for that amount of time), in which case it returns to the processing-wait group. This model is very simple and not very useful (it assumes among other things that there is no overhead associated with moving a job in and out of the processing group). In order to get a more detailed model, additional groups could be added, more descriptive information could be associated with jobs (ie. the amount of memory required), and the group transition event routines could be made more sophisticated (eg. taking into account job memory requirements, job handling overhead, etc.).

In general the following procedure should be used in modelling a system:

1. Isolate the system of interest and identify all interactions between the system and its environment;
2. Identify those objects in the system whose behavior is of interest and set up entity classes (with entity characteristic information);
3. For each entity class identify the types of behavior that are possible and establish a network of groups

for the significant aspects of this behavior;

4. Consolidate the groups that are essentially the same but that have different entity classes;
5. Define group transition event routines for each group transition in the model.

4.2 The Structure Of MNSS Models

MNSS models are designed to accomodate the behavior of user processes, system processes and messages in HPC network systems. The composite behavior of these objects encompasses all interesting system behavior as identified in Chapter 3. Accordingly user process, system process and message entity classes have been defined. For an MNSS model, entities from these classes exist in a network of groups, entity sources and entity sinks structured to represent all significant entity behavior.

MNSS models can be built using an HPC system model as a basic component. As such, the HPC system model has to be adaptable to arbitrary communications configurations (ie. network environments). A submodel structure has been developed to enhance the flexibility of the HPC system model. The submodels that have been constructed are essentially another (lower) level of MNSS model components.

Two submodels make up an HPC system model; they are the processing submodel and the communications submodel. The processing submodel represents the behavior of user and system processes. The behavior of these processes is affected by, and affects, the flow of messages in the communications submodel. The interactions between the two submodels emanate from each submodel's group transition event routines. There are no group transitions between the processing submodel and the communications submodel, but a group transition in one submodel may trigger (by way of an event routine) a group transition in the other.

To form a complete MNSS model, the communications submodels of several HPC system models are connected together by group transitions. If two HPC systems are tied together by a communications link in the network to be modelled, then the corresponding communications submodels are connected.

The overall structure of an HPC system model model is shown in Figure 4.3. The details of submodel organization and interaction are discussed in the following sections on the processing submodel and the communications submodel.

4.2.1 The Processing Submodel

The processing submodel is illustrated in Figure 4.4. This submodel consists of four groups, two entity sources, two entity sinks and eight group transitions. These structures are arranged to provide a representation of an HPC system which describes the significant behavior of user and system

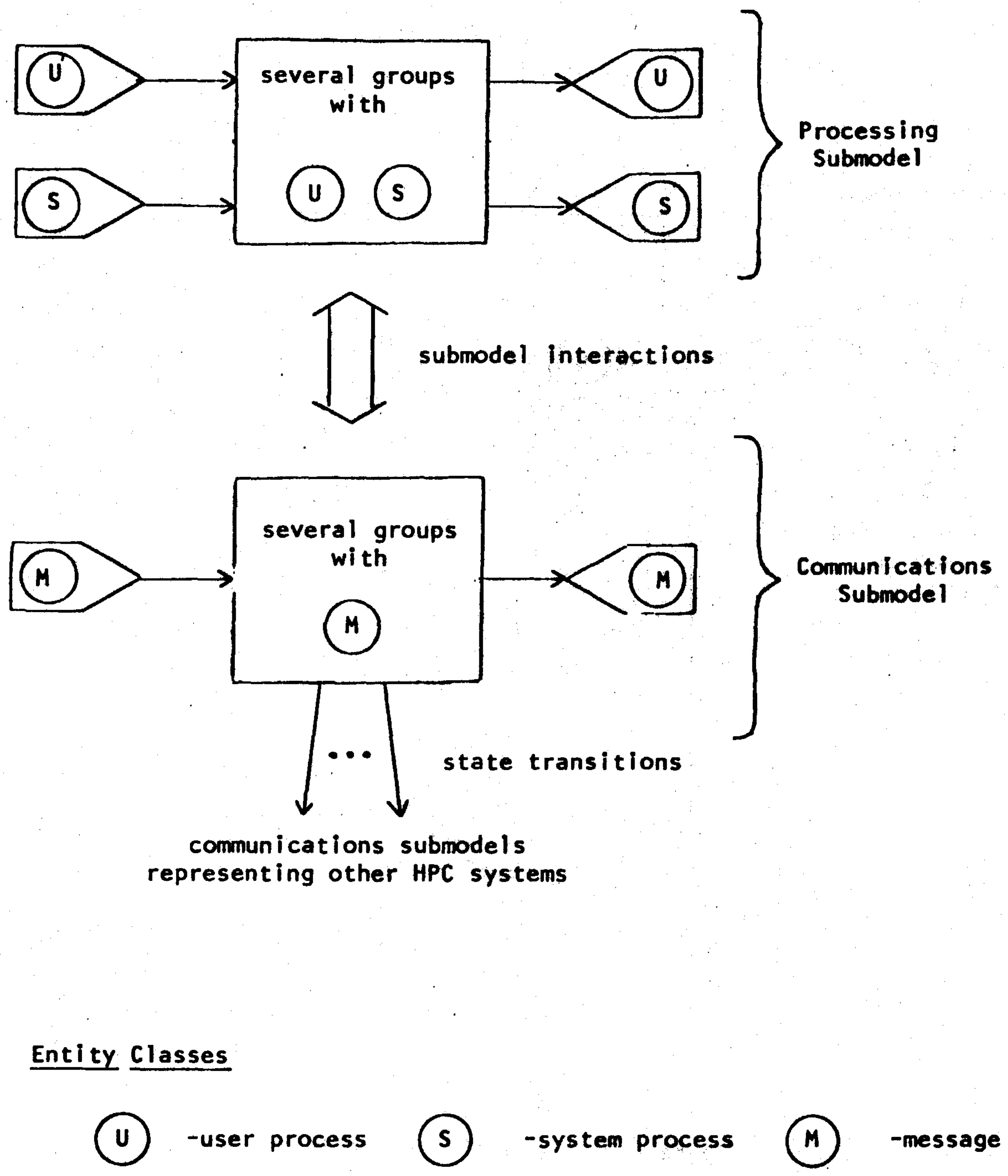


Figure 4.3 General HPC System Model Structure

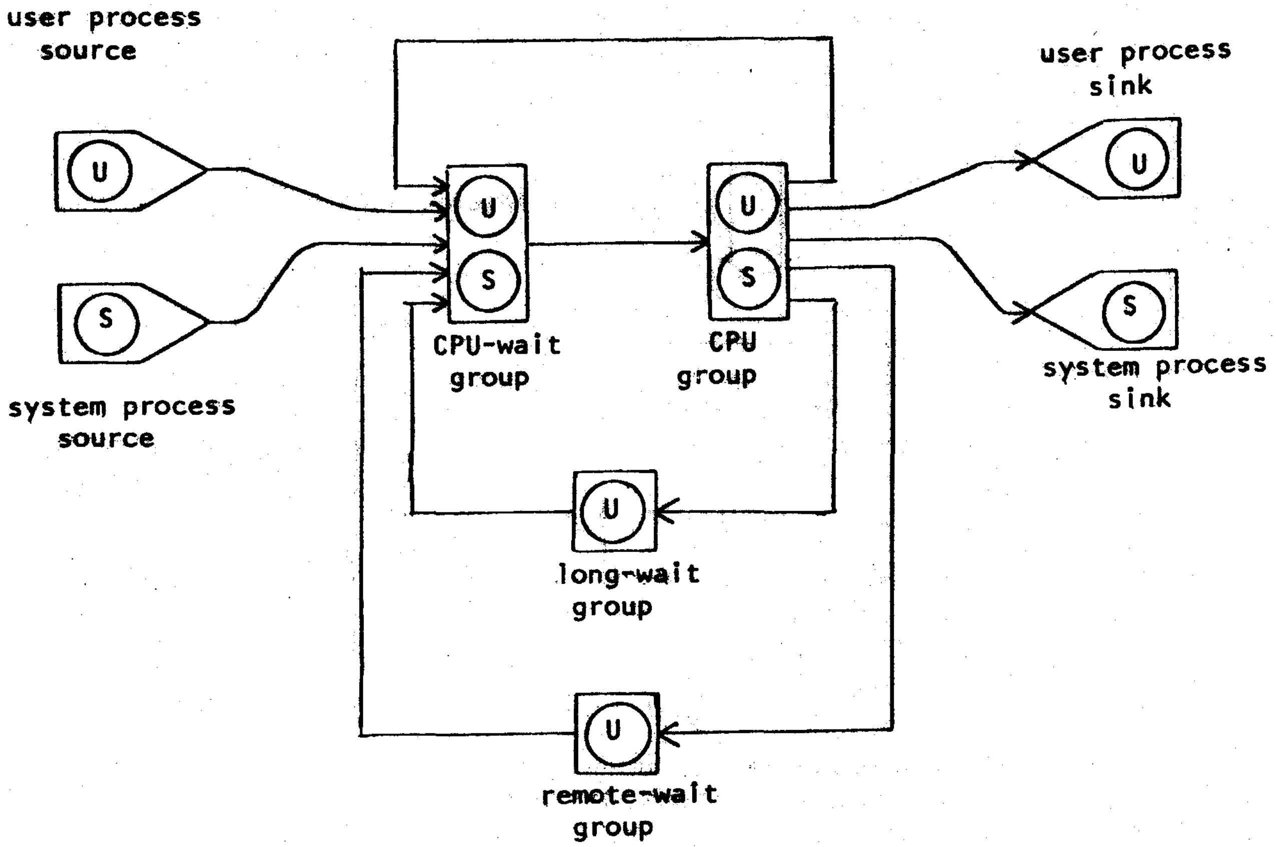


Figure 4.4 Processing Submodel

processes. The entities that flow through the submodel are, naturally enough, user processes and system processes. Each process entity has information associated with it which, along with group ordering and transition information, determines when group transitions are to occur. The nature of the information associated with a particular process entity is based on it′s entity class.

The following items of information are associated with each user process entity:

1. The ′user process type′ identifies the user process as being either locally or remotely initiated.
2. The ′complete time′ is the processing time (ie. residence in the CPU group) required for user process completion.
3. The ′long wait time′ is the processing time until the user process initiates a long wait or, if the user process is doing a long wait, the time when the long wait completes.
4. The ′remote wait time′ is the processing time until the user process initiates a remote request (for a locally initiated process) or a completion response (for a remotely initiated process). For a remotely initiated process this item is equivalent to the complete time.
5. The ′process time distributions′ are the time

distributions from which information items 3 and 4 are generated (eg. when a user process terminates a long wait the processing time until the next long wait is generated using the appropriate process time distribution).

6. The ´memory requirement´ specifies the amount of primary memory needed to hold the user process code and data (ie. the information that needs to be brought into primary memory to allow the process to run).
7. The ´remote destination distribution´ is the distribution that is used to generate the destination of a remote request for a locally initiated user process, or the response destination for a remotely initiated user process (in the latter case the distribution deterministically generates the source of the remote request).
8. The ´message size distribution´ is used to generate the size of a remote request message or a completion response message.
9. The ´remote process distributions´ are defined for locally initiated processes only. They are used in the generation of a remote user process that is initiated as the result of a remote request. These distributions include the process time distributions and the message size distribution (for the reply generated by the remote process).

A system process exhibits much simpler behavior than a user process; a system process does not do long waits, and also does not initiate remote requests. Consequently, the information associated with a system process entity is a subset of that associated with a user process entity. The following information items are defined for each system process entity:

1. The ′system process type′ identifies the task of the system process as either initiating a remote request communication or handling a message from a remote source (ie. routing it through the communications system).
2. The ′complete time′ is the processing time necessary to complete the appropriate system process task.

The four groups in the processing submodel are designated the CPU group, the CPU-wait group, the long-wait group, and the remote-wait group. The CPU group can be occupied by a single entity from the system process entity class or the user process entity class. A process in the CPU group has control of the HPC CPU. The process may be doing either processing or a short wait, which does not result in a switch of CPU control to another process (as discussed in Section 3.2.1). When the process is preempted from CPU control, completes, initiates a long wait, or initiates a remote request, an appropriate group transition from the CPU group will occur. Preemption will cause an entity transition to the CPU-wait group; a long wait initiation results in a transition to the long-wait group; the initiation of a remote request by a process brings about a transition to the remote-wait group; and when a process completes it is swept away to an entity sink (user processes to the user process sink, system processes to the system process sink).

The three 'wait' groups characterize the behavior of a process entity when it is not in control of the CPU. If a process entity is in the CPU-wait group then the process is waiting to gain control of the CPU. The CPU-wait group has a dual FIFO queue ordering for system and user processes. System processes and user processes are kept in seperate queues. When the CPU group is unoccupied and the CPU-wait group is occupied then a process will make a group transition from the CPU-wait group to the CPU group. The process that makes the transition is taken from the system queue if it is nonempty, else it is taken from the user queue. The presence of a system process in the CPU-wait group causes a user process in the CPU group to be preempted. The user process undergoes a group transition to the CPU-wait group and is positioned at the head of the user process queue, becoming the next user process to be served. In addition, a user process can be preempted from the CPU group if it has done a 'user slice' of processing and there are other user processes in the

CPU-wait group. In this case the preempted user process is placed at the tail end of the user process queue in the CPU-wait group.

The long-wait group and the remote-wait group can only be occupied by user process entities (system process entities do not exhibit the behavior these groups represent). The long-wait group is occupied by a user process entity when the entity is doing a long wait. This group has an ordering of user processes based on the length of time each process must remain in the group. When the long wait of a process has completed the process undergos a group transition to the CPU-wait group and is placed in the user process queue. The remote-wait group is occupied by a user process when the user process has initiated a remote request. The remote-wait group does not impose an ordering on the entities that occupy it. When a remote request has completed (ie. a response message is received at the HPC node), the user process that initiated the request is removed from the remote-wait group, and placed in the CPU-wait group.

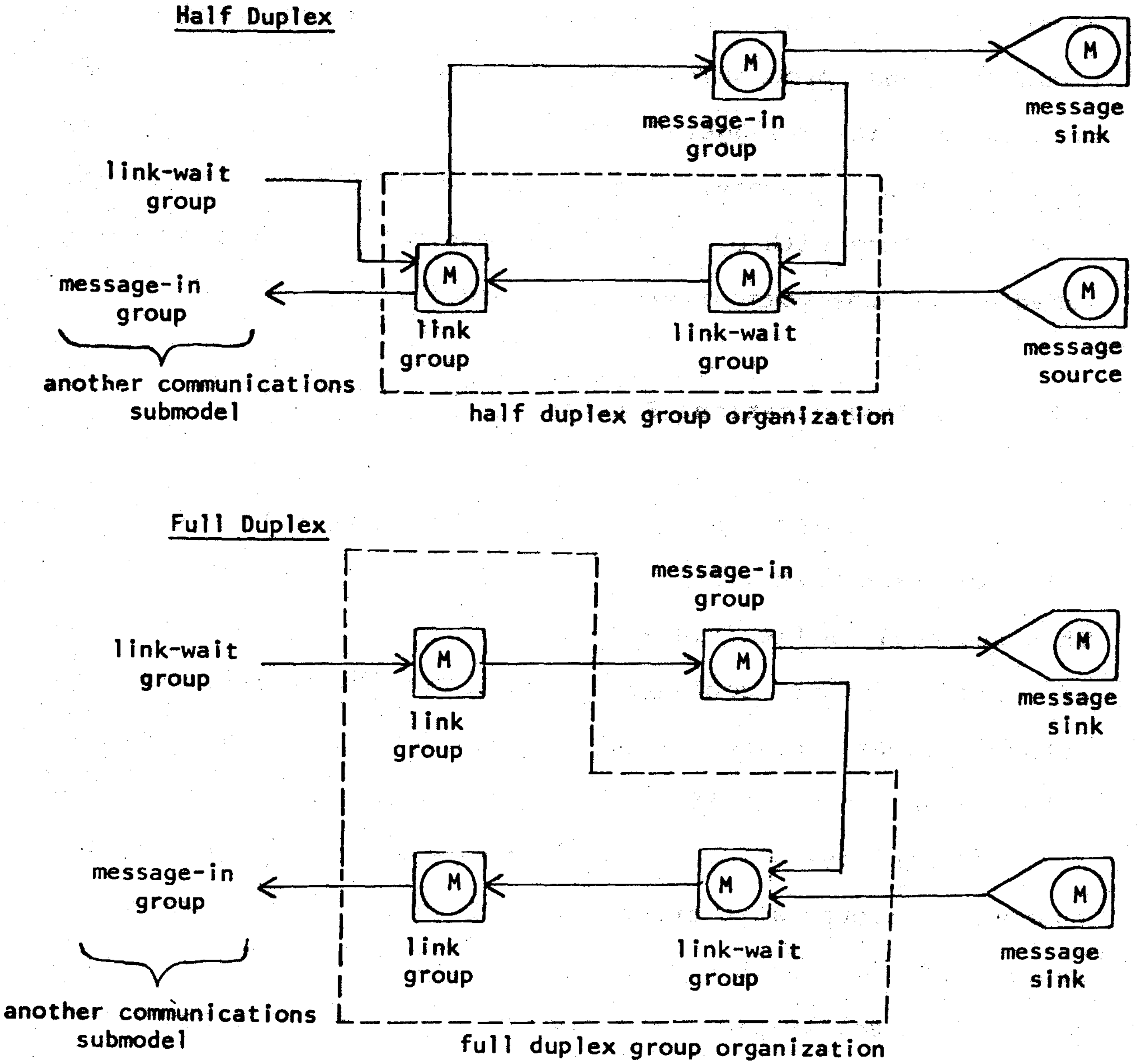
The process entities that flow through the processing submodel can be generated at MNSS model initiation to represent a permanent user load, or they can be generated dynamically to represent a user load that changes over time. The processes generated as part of a permanent user load are always user processes and are placed initially in the CPU-wait group (when the model is used for simulation, care should be taken to avoid biasing CPU-wait queue statistics as a result of this initial condition); system processes are generated as a result of the actions of user processes and as such cannot be in existence at model initiation when no action has taken place. Processes are generated dynamically at the system process source and the user process source, and are injected (via group transitions) into the CPU-wait group. The user process source generates user processes with all the associated information (complete time, etc.). The user process generation capacity of the user process source, coupled with the group transition event routine (for the transition from the user process source to the CPU-wait group) that dictates the frequency of entity generation, can completely represent a dynamic user load on an HPC system. The user process source also generates user processes to satisfy remote requests. The transition of this type of user process into the CPU-wait group is triggered by the completion of the system process that handled the incoming remote request.

The system process source generates system processes to handle communications processing. The processing may be required for the initiation of a remote request or for handling the arrival of a response. A remote request is made by a locally initiated user process entering the remote-wait group. The completion of a remotely initiated process (ie. the process is absorbed by the user process sink) signals the initiation of a remote response. Both the remote request and response cause identical communications processing and hence the same type of system process is generated (ie. with the same completion times). A different type of system process is generated by the system process source to handle the communications processing for network message routing. This processing is triggered by a group transition in the communications submodel (to be discussed in the next section in detail along with other interactions between the processing submodel and the communications submodel).

4.2.2 The Communications Submodel

The communications submodel provides a representation of the important behavior of the communications message flow. This submodel is flexible enough to be adapted to a wide range of communications configurations. Two variations on the communications submodel are shown in Figure 4.5. This Figure illustrates how half and full duplex communications capabilities are integrated into the submodel. In general the submodel incorporates three types of groups (two of which are duplicated for each communications link modelled), an entity source, an entity sink, and five principle group transitions. The entities that flow through the submodel belong to the message entity class.

The information associated with a message entity controls


Half Duplex

Full Duplex

Figure 4.5 Communications Submodel

the message flow through the communications submodel. This information is used by group transition event routines to direct the message to its local or remote destination. In addition, once a message reaches its destination the entity information is used to determine the correct group transition action to be taken (eg. restart a local user process or generate a remotelly initiated user process). The message entity information includes the following items:

1. The 'message type' classifies the message as either a remote request message or a response message.
2. The 'message source' identifies the user process which initiated the remote request and caused the message to be generated (for a response message the message source is the user process that originally initiated the remote request).
3. The 'message destination' is an identification of the HPC system to which the message is directed.
4. The 'message length' is the length of the message in bytes.

The three types of groups in the communications submodel are designated the message-in group, the link-wait group and the link group. There is only one message-in group per communications submodel, but there can be duplicates of the link-wait group and the link group. Essentially the 'link section', blocked off in the submodel illustrations of Figure 4.5, is duplicated for each communications link (half or full duplex) attached to the HPC system being modelled. The flexible structure of the communications submodel lends itself to representation of a wide variety of HPC system communications configurations.

The message-in group can be occupied by multiple messages with no ordering enforced on the messages. Messages occupy the message-in group when they have been received by the HPC system and are being processed. This means that a for each message in the message-in group a system process, generated when the message entered the message-in group, is active in the processing submodel. The system processing to be done upon arrival of a message into the message-in group is for message routing activities. This includes determining the message destination; if the destination is that particular HPC system then the appropriate user processing action is taken, else the message is started on the way to its remote destination.

The action taken on a message, and the accompanying group transition from the message-in group, occurs when the corresponding system process in the processing submodel completes and is absorbed by the system process sink. If the message is directed to a remote destination, the message will make a group transition to an appropriate link-wait group. The particular link-wait group chosen is determined by a routing algorithm incorporated into the group transition event routine (the link-wait group is selected such that the message will be routed correctly to its destination over the link associated with the link-wait group). If the message is directed to the local HPC system then a transition will occur to the message sink where the message will be absorbed. If the message is a response to a remote request, this transition will trigger a transition in the processing submodel from the remote-wait group to the CPU-wait group for the user process that originally initiated the remote request. Otherwise the message is a remote request and a remotely initiated user process will be generated by the user process source and injected into the CPU-wait group.

A message will be launched from the message source to a link-wait group when the system process initiated to handle a remote request completes (ie. the process is absorbed by the system sink). The message generated by the message source will be assigned descriptive information (source, destination, length) based on information associated with the user process making the remote request.

In general the interactions between the processing submodel and the communications submodel occur in conjunction with transitions by messages to and from the message-in group and transitions from the message source. These interactions, which are designed into the event routines, are the key to modelling communications overhead in the HPC system.

The key to modelling a variety of communications configurations is the flexibility to customize the organization of the link and link-wait groups in the communications model.

The link group can be occupied at any time by at most a single message. A message in the link group is in the process of being transmitted (physically) from one HPC system to another. Consequently each link group is part of two communications submodels; it is the bridge that ties together the models of individual HPC systems to form MNSS models. A message leaves the link group and enters the message-in group in the communications submodel of the HPC system to which it was transmitted when the transmission has completed. The time of transmissiion is based on the length of the message (a message entity information item) and the speed of the link (taken into account in the link to message-in group transition event routines). Corresponding to every link group in the communications submodel there is a link-wait group. Messages in the link-wait group are ordered in a FIFO queue. When the link group becomes free for entry of a message (ie. there is no message in it and the delays due to physical link control have elapsed) the head message in the link-wait message queue makes the transiton to the link group.

The event routine that is associated with the group transition from the link-wait group to the link group models the link control protocol used on the corresponding link in the real network system. For every variety of control protocol there is a distinct event routine. For example, a half duplex link control event routine would be concerned with delays in turning the link around and in switching a link from an idle to a transmit-ready disposition. On the other hand, a hard-wired full duplex link would have an event routine that is only concerned with whether the link is free or not (the link is always transmit-ready and oriented in the right direction).

Once the appropriate event routine for a link control protocol has been defined, the next step in modelling a link in the communications submodel is to make the appropriate link and link-wait group connections. The differences in the conections for a half and full duplex link can be seen in Figure 4.5.

4.3 Simulation Using The HPC System Model

Several features of the HPC system model enhance its potential as a simulation model. Among these are the types of structures used in the model, the model's levels of detail in representing significant HPC behavior, and the flexibility of the model as a building block for MNSS models.

The structures defined by the MNSS modelling discipline and used in the HPC system model can be conveniently manipulated in the context of a simulation system. The information associated with MNSS modelling structures can be implemented as data structures in the simulation programming language. For example, each entity class can have an information table, with each entry in the table corresponding to an active entity in the model. The group transition event routines that control the action in the model can be implemented as procedures (pieces of code that are callable on demand by the main simulator program). A group transition occuring in the simulation model then results in execution of the appropriate transition procedure. In general, MNSS model structures can be systematically converted to the programming language structures of the simulation system.

Another factor that affects the conversion of a model for use in simulation is the level to which the model represents a system's behavior. A model can fail to be an effective simulation model if it represents a system with too much or too little detail. Too much detail (ie, the model represents some system behavior at too low a level) can make the model's use in simulation a torturous exercise in overkill. On the other hand, a model which does not represent certain aspects of a system in enough detail would not be of use in simulating all the significant system behavior.

The HPC system model utilizes two primary levels of detail to produce a valid and effective model of an HPC system. The processing submodel represents one level of abstraction of system structures. The communications submodel represents structures at another, higher level. The message in the communications submodel is an entity that is created, manipulated and destroyed by the processes of the processing submodel.

By modelling messages separately from processes, easy correspondences can be made between HPC network communications structures and structures in the HPC system model (eg. communications links and protocols). This multilevel modelling technique also avoids the complexity and inflexibility of representing communications in a processing model (in particular the group transition event routines would be very hard to define).

Perhaps the key feature of the HPC system model is the way it can be used as a building block for MNSS models. This capability was shown in Section 4.2.2 in the discussion of the use of the link group in connecting HPC system models together. Basically a network of communications submodels is formed, with each communications submodel having a corresponding processing submodel. The simulation system implementation of this network model structure is facilitated by adding node and link qualifiers to the MNSS model group names. These additional qualifications are necessary since there are several copies of each group in a network model made up of several submodels. With each group in a network unambiguously identified, simulation system data structures can be easily implemented and managed.

CHAPTER 5

MNSS IMPLEMENTATION AND USE

There is a great deal of latitude in implementing MNSS. The details of a particular implementation may be shaped by the specifiations of the computer system on which MNSS is implemented, or by the needs of the network analyst who is to use MNSS. On the other hand there are a number of essential features that should be common to all effective implementations. These features involve the overall structure of an MNSS implementation, the representation of a simulation experiment, and the basic characteristics of the simulator. By nature they are both useful and feasible within the restrictions of any particular implementation. Therefore an accounting of these features should be made to guide MNSS implementors.

The following sections provide a discussion of important MNSS-implementation independent features. Appendix A supplements this discussion with a brief description of some interesting aspects of a particular MNSS implementation.

5.1 Overall Structure Of A MNSS Implementation

The overall structure of a MNSS implementation should reflect an emphasis on top-down and modular organization. One possible structure that does this is modularized based on MNSS functions. A graphic view of this structure is given in Figure 5.1.

Five major MNSS functions have been identified:

1. create a simulation experiment specification,
2. recall a simulation experiment specification,
3. modify a simulation experiment specification,
4. save a simulation experiment specification, and
5. run a simulation experiment.

These functions represent effective tools for performing simulation experiments. A simulation experiment specification is created by first initializing all relevent information tables and then, through user interaction, building a basic network specification (ie. the number of nodes in a network and how they are connected). Once an experiment specification has been created it can be modified interactively to produce a variety of related specifications. In particular network configuration information (eg. link speeds) and user load information can be manipulated. It is possible to create non-volatile copies of an experiment specification, that is, copies that do not disappear when MNSS use is terminated; experiment specifications can be saved and recalled later when they are needed. A user can choose to run a simulation experiment at any time with the available experiment specifications.

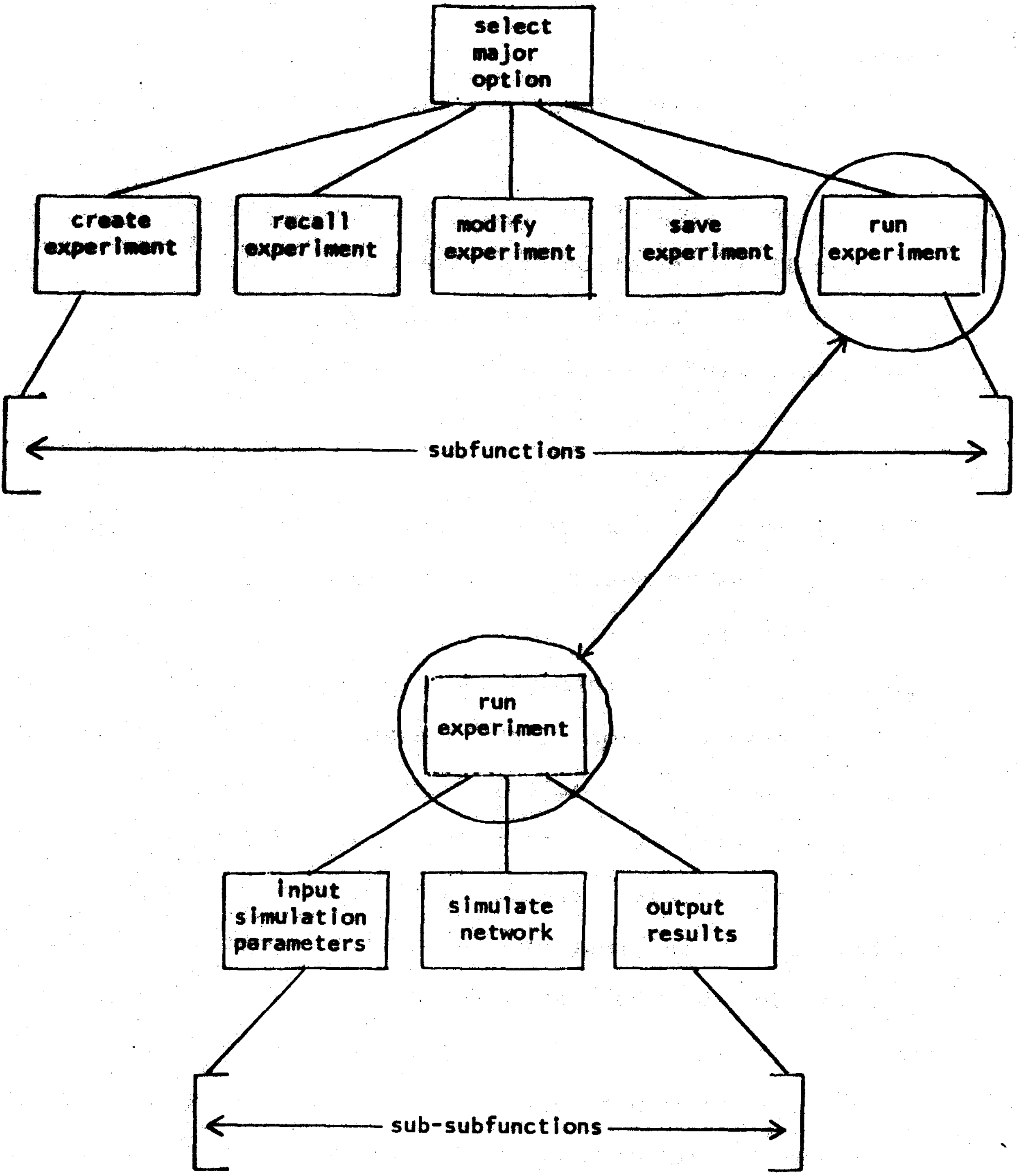


Figure 5.1 MNSS Implementation Organization

Each of the major MNSS functions can be implemented as a super module containing numerous subfunction modules (each of which may contain sub-subfunction modules, etc.). This modularization according to subfunction is shown in Figure 5.1 for the major function ′run experiment′.

At the top of the top-down organization shown in Figure 5.1 is a module which acts as a switch between the major functions. In particular this module is implemented to switch control, at the request of the user, between the five major functional modules; the only exception is when MNSS is first started up, in which case the user only has the freedom to create or recall an experiment specification. Any other function would be invalid because of the lack of an experiment specification to operate on.

5.2 Representation Of A Simulation Experiment

The items of information that are needed in the representation of a MNSS experiment are dictated by the characteristics of the HPC network models. For example, the network representation must include the number of nodes and links, the connectivity of the network, and link descriptive information (eg. half or full duplex, link speed, etc.). The management of these information items is to a large extent MNSS-implementation dependent, but there are some information management policies that are generally useful. Two such policies deal with providing alternative representations for individual information items and minimization of the information redundancy in an experiment representation.

In most cases a MNSS experiment representations will contain a large amount of redundant information if the user load for each node in the network is represented seperately. This redundancy often results from the use of a standard user load that is specified for more than one node in the network. To reduce redundancy, information can be kept in ´common´ areas where it can be accessed by (ie. linked to) higher level information structures in the experiment representation. Information items or structures that are potentially used more than once in an experiment are candidates for ´common´ status. This applies to everything from distribution specifications to complete process specifications.

Figure 5.2 illustrates an experiment representation breakdown with common process characteristics and common distributions. The common process characteristics, that might be grouped together and shared by processes, include process time distributions, memory requirements and message size distributions. A separate table(s) could be used to store distribution specifications. Entries in this table(s) would be pointed at by entries in the process table and the common process characteristics table.

The success of the common information item representation scheme rests on three factors. One is the storage requirements

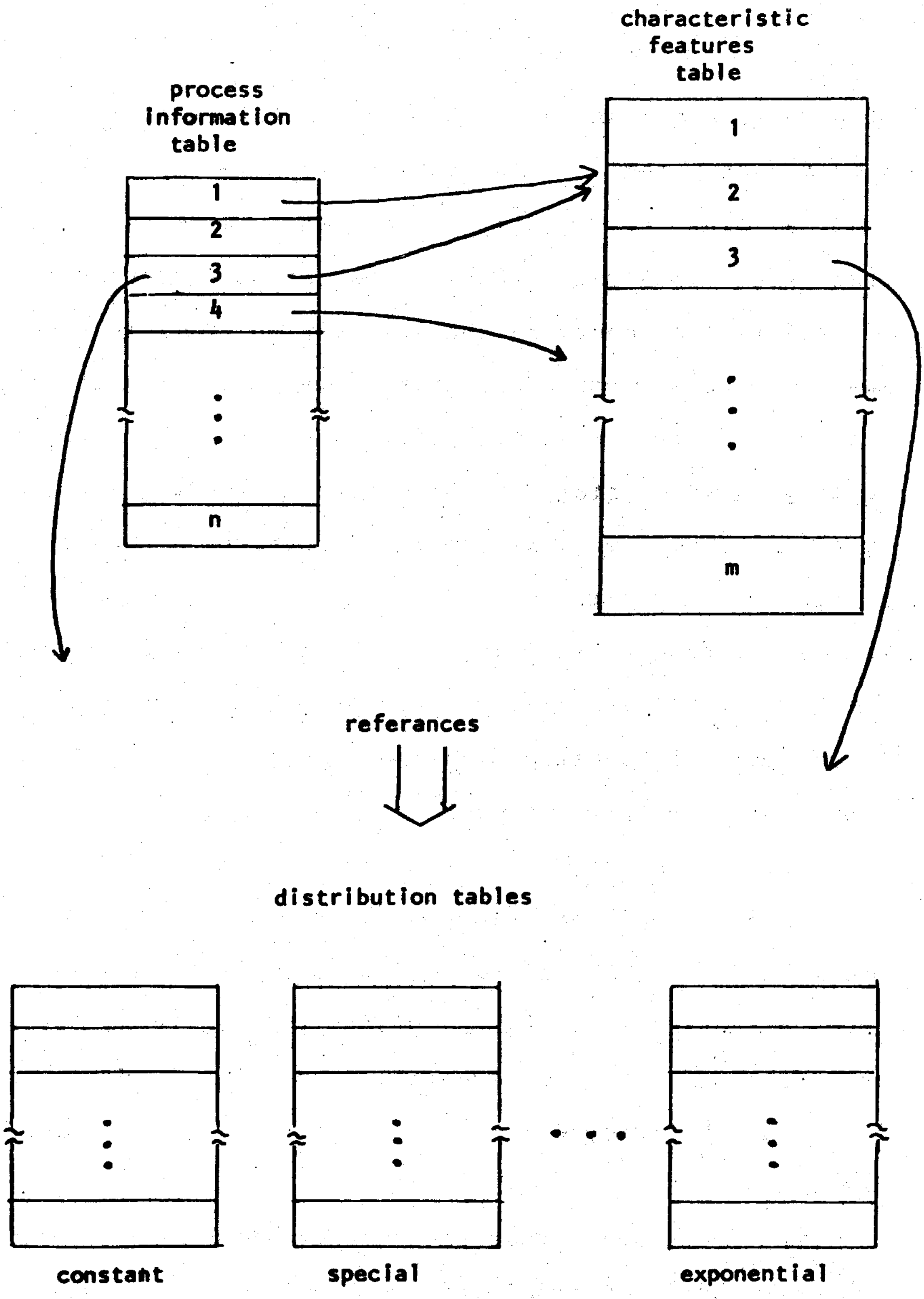
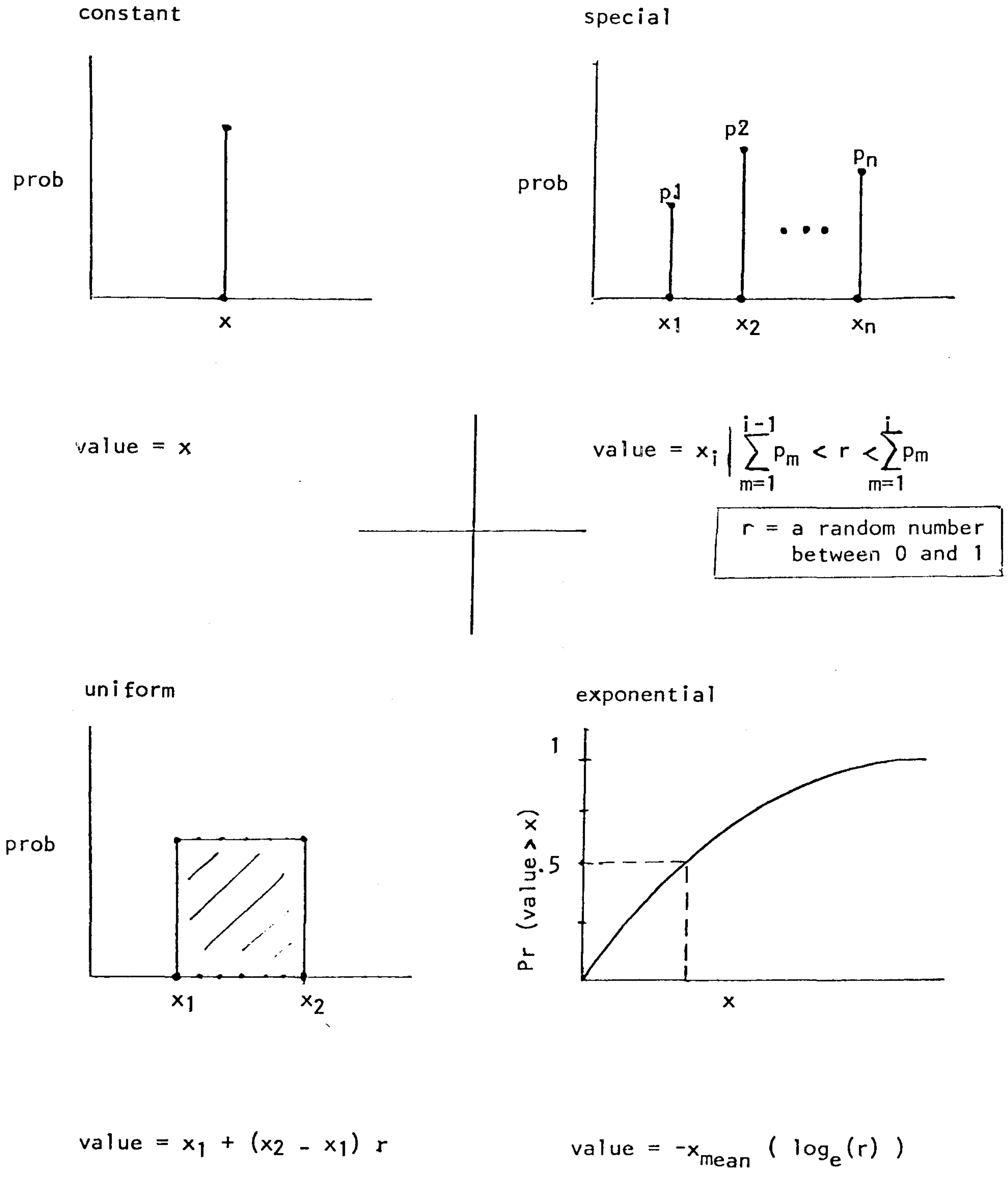


Figure 5.2 Experiment Representation Breakdown

of a reference link relative to the requirements of the common information item. If an information item needs more storage than a link to that item, then the information item may be best kept in a common table. The second factor is the number of references made to an information item. In general, the smaller the storage difference between the item and a link to it, the larger the number of references that are needed to justify common status. The third factor for success is the specification time that can be saved by a user in creating simulation experiments with standard process specifications. Modifications of existing simulation experiments can also be greatly simplified. For most foreseeable MNSS implementations a common information item organization for experiment representations is attractive in terms of system efficiency and user convenience.

Alternative representations for individual items can be provided in a MNSS implementation by allowing a user to select from a variety of distribution types. These types could include constant, special, uniform and exponential distributions. Figure 5.3 illustrates the various distribution functions.

The constant, uniform and exponential distributions are commonly used in modelling and simulation. The function used for the exponential distribution is taken from MacDougall's tutorial paper on simulation [19]. This function can be adapted to generate a finite range of values by specifying a



value = x

value = $x_i \mid \sum_{m=1}^{i-1} p_m < r < \sum_{m=1}^{i} p_m$

r = a random number between 0 and 1

value = $x_1 + (x_2 - x_1)\ r$

value = $-x_{mean}$ ($\log_e(r)$)

Figure 5.3 MNSS Probability Distributions

maximum and minimum value. Values generated outside this range are either discarded or replaced by the closest valid value. This modification of the exponential distribution function is useful because information items do not always range in value from zero to infinity (note that constant and uniform distributions have limits).

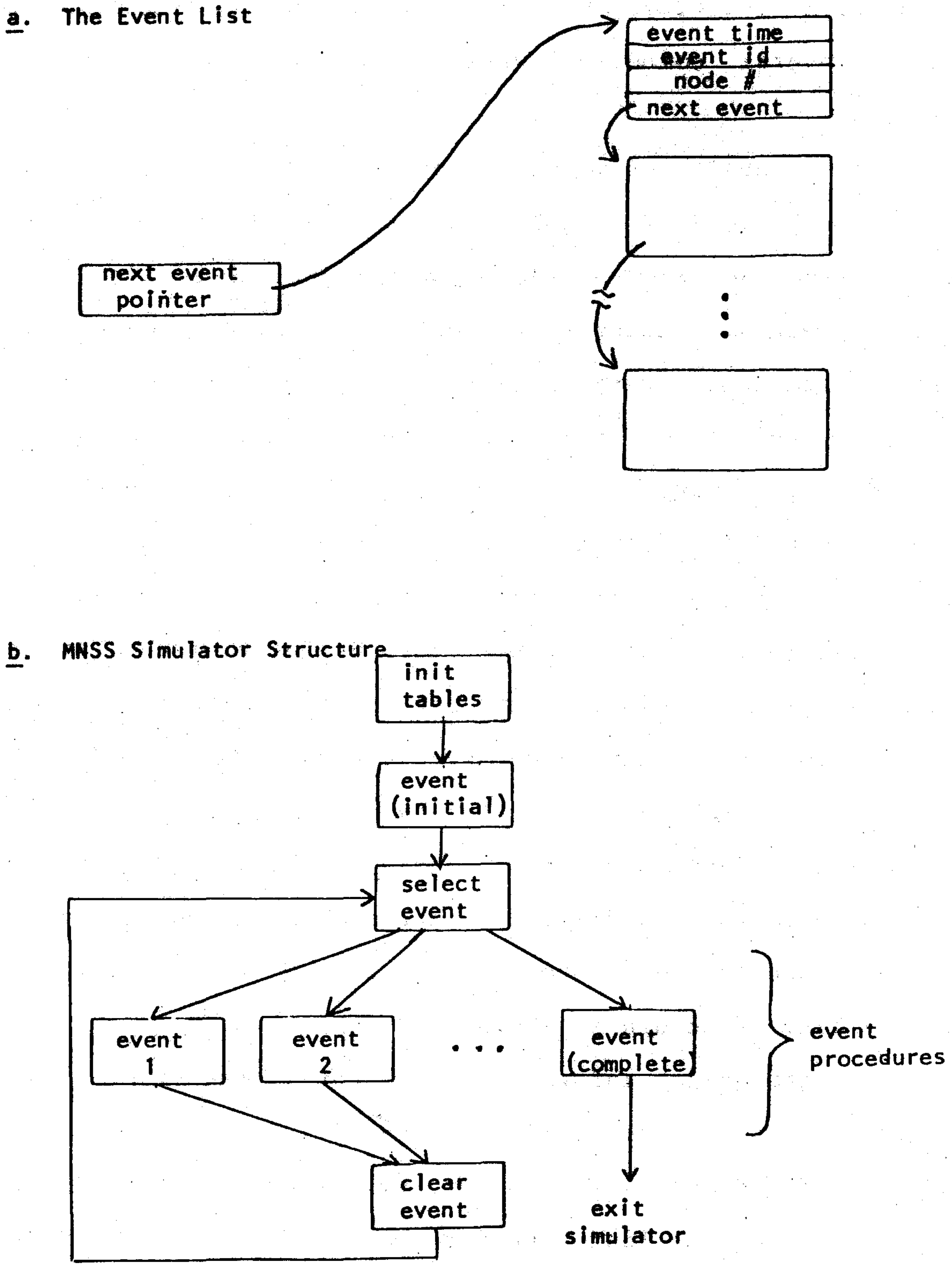
The special distribution is useful for information items with values that can only be generated from an unusual distribution (ie. not constant, uniform or exponential). A special distribution consists of a finite number of values, each of which has a probability. The sum of the value probabilities is one. The number of values in a special distribution is dictated by the experiment specification requirements of the particular simulation study in which the special distribution is used. In general a special distribution can be used to approximate any distribution and is the alternative when constant, uniform or exponential distributions are inadequate.

5.3 MNSS Simulator Characteristics

An event-driven simulator is especially suited to the simulation of systems characterized using the MNSS modelling discipline. Events in the MNSS modelling discipline describe the effects of the state changes of entities in a system model. These events can be easily converted to the events of an event-driven simulator. Another valuable feature of an event-driven simulator is the capability for pseudo-coincident simulation activity. This is required for network simulation where several computer system nodes may be operating in parallel.

An event-driven simulator can easily be implemented with a clearly defined modular structure. This structure is illustrated in Figure 5.4a. The modules can be divided into two groups: those that maintain the event list (initialization, event selection, event scheduling and event removal) and those that do event activities (the event procedures). The event list maintenance modules are independent of the type of system being simulated. The event procedures on the other hand are specific to a particular type of system (eg. HPC network systems). By insuring the isolation of system dependent features, the simulator can be straightforwardly adapted whenever the system is modified.

The structure of the event list is the key to pseudo-coincident simulation activity. It is the place where the simultaneous activities occuring throughout a simulated network are merged into a serial stream of events. The MNSS event list structure is shown in Figure 5.4b. Each event list item carries enough information to identify the event type and where in the network system the event is to occur. The event list is linked together with the next event to occur placed at the head of the list. The foremost event is pointed to by a



a. The Event List

b. MNSS Simulator Structure

Figure 5.4 MNSS Simulator Characteristics

'next event pointer'. Event list items that identify coincident events are arranged with the first-scheduled events being foremost in the event list (ie. first to happen by way of the event procedures). This procedure yields an adequate ordering of events for MNSS studies of HPC network systems (where events are frequent and limited in scope).

Appendix A discusses the MNSS/3000 implementation which incorporates the ideas presented in this chapter. There is also additional generally applicable information on the implementation of MNSS. The MNSS/3000 implementation is used to provide the simulation results that are used in the following chapters on verification and experimentation.

CHAPTER 6

VERIFICATION OF THE MNSS IMPLEMENTATION

In order for simulation results to be useful they must represent a verifiable picture of the behavior of the system being simulated. Clearly it is not realistic to compare the results of every simulation experiment to measurements of the corresponding real system. This would defeat the purpose of the simulation system to be more adaptable for study then a real system. In addition a simulation system may be used to anticipate the behavior of a system yet to be developed. In this case no measurements are possible, hence other verification techniques must be applied.

Effective verification procedures utilize a limited amount of information about the predicted or measured behavior of a system in order to prove that simulation results (in general) are valid. Among the procedures that are often used are the identification and verification of simulation model parameters and the functional verification of simulation system operations.

Simulation model parameters represent real system constants that can be isolated from complex system interrelationships. They may be anything ranging from the speed of a communications link to the time it takes to do a particular task (eg. start an execution of a process once it′s in main memory). The verification (where it is necessary) of parameters used in the simulation model can be done through measurements or the prediction of system behavior.

The functional verification of the simulator requires matching behavior in the simulated system to behavioral data produced by simulation. This can be done by first insuring that basic behavior patterns in the simulated system are represented in the simulation model; if action ′X′ always follows action ′Y′ in the simulated system then the same sequence must be represented in the model. Then the behavior pattern results produced by the simulator are verified in view of corresponding results generated by the simulated system.

Functional verification can be done at two levels. First, fundamental behavior pattern results can be predicted for a particular system and then confirmed for the simulation system. This verification can be done to the extent that fundamental behavior patterns occur in a system and produce predictable results. Another level of functional verification can be done through the use of system measurements. Unpredictable behavior pattern results can be identified and compared to simulation experiment results. This form of functional verification must be done with discrimination because there are potentially an infinite number of behavior patterns.

The following two sections of this chapter will describe the procedure by which MNSS is verified in light of HPC system behavior. Because HPC network systems are in a development stage, no measured results are available, but measurement experiments are specified for later use. For now verification of critical model parameters and functional behavior are dependent on predicted system behavior.

6.1 Simulation Model Parameters

For the HPC system model three groups of parameters have been identified:

1. system process completion times,
2. process switching overhead,
3. operating system constants.

The values of the parameters in these three groups are either constants or determined by a known function (of environmental factors).

There are two types of communications system processes represented in the HPC system model. In an HPC system these system processes have associated modules of code that do link control and message handling. The generation of a system process results in a reasonably predictable execution path through the code modules. By taking into account the code execution times of an HPC system and the nature of the execution paths, a prediction can be made for the system task completion times. Based on this procedure the following completion times have been derived; 200ms for the message generation system process, and 100ms for the message routing system process.

In order to get more accurate values for system process completion times, measurements could be done on an HPC network system when HPC systems and measurement tools become available. The measurements would require a message source, a software monitor, and at least a three node HPC network (to have message routing). The software monitor would take timings at the end points of the execution paths through system process code modules. These measurements (repeated many times) would yield representative values for system process completion times.

Process switching overhead can, in general, be represented by the following expression:

$$Tov = C1 + C2*Min + C3*Mout$$

In this expression, process switching overhead (Tov in units of time) is a function of the memory requirements of the processes involved in the switch (Min and Mout in units of space), and the system code that must be executed to do memory management (C2 and C3 in units of time/space) and CPU control transfer (C1 in units of time). System code execution times can be predicted based on an examination of the execution paths associated with process switching. For the first implementaion of the HPC system model (ie. the implementaion referred to in this thesis) a standard memory requirement is assumed for all user processes so the expression for process switching overhead reduces to a constant (100ms is the derived value of the constant). This simplification of the overhead function is necessary because of the difficulty to predict its value in general.

Measurements on an HPC system will be needed to accurately define the parameters in the overhead function. These measurements will require an HPC system, a software monitor, and a controlled user process load on the system. The measurement procedure would involve generating two user processes with a known memory requirement and then allowing a specified number of process switches to occur. By measuring the overhead associated with a variety of process memory requirements enough values could be generated to identify the overhead function.

Operating system constants are set to a specific value when the operating system is implemented. Therefore no measurements of HPC system operations are needed. Some operating system constants are fixed for all HPC system implementations. For example, the minimum CPU control time for a user process before it can be preempted by another user process (ie. the 'user slice') is set to 500ms. On the other hand some constants are particular to an installation and must be redefined for each different HPC network. The window size for a half duplex communications line is an example.

6.2 MNSS Functional Verification

There are a number of fundamental behavior patterns that are inherent in an HPC network system. These include:

1. continuous CPU bound execution with a standalone user process (ie. there is no process switching overhead),
2. long wait initiation by a process, removing the process from contention for CPU control,
3. process switching and the consequent overhead,
4. system processing resulting from network message flow,
5. the startup of a remote user process after receipt of a remote request,
6. the completion of a remote user process resulting in the generation of a remote response message,
7. local user process restart on receipt of a remote response,
8. store-and-forward message traffic in networks that are not fully connected.

Each of these behavior patterns should be duplicated in the behavior of the simulated HPC network system. To show that this is indeed true specialized simulation experiments can be defined that focus on a particular pattern of behavior. The experiment specifications must describe a system whose behavior can be predicted. If the simulation experiment results are as predicted then the fundamental behavior pattern that produced the results is verified. The experiment analysis that follows is part of the process to functionally verify MNSS.

In order to verify the functioning of process switching, remote request operations and store-and-forward mechanisms, a standard simulation experiment has been specified. For each MNSS function to be verified, appropriate user process definitions are added to this specification. Appendix B gives a summary of the standard experiment in the MNSS descriptive notation.

6.2.1 Process Switching

The simulation experiment used to verify process switching has two CPU bound (no long waits) user processes competing for CPU control. Neither of the user processes ever makes a remote request so all activity in the simulated system takes place at one HPC network node. The activity in the system can be described by the following state-time sequence:

```
   time >  * 100ms * 500ms * 100ms * 500ms *  actions repeated
Process 1  *  wait *  CPU  *  wait *  wait *     .  .  .
Process 2  *  wait *  wait *  wait *  CPU  *     .  .  .
```

The time intervals that occur when both processes are in the CPU-wait state represent the times when there is process switching overhead. Given this simple behavior pattern and knowledge of the length of time the simulation spans, it is easy to predict CPU utilization and CPU-wait statistics. For example, durning a one minute period the process switching overhead is derived by dividing 60000ms by 600ms (the period that includes one process switch) and multiplying the result by 100ms (or the time calculated to do one process switch for the two processes involved). The overhead in this case is equal to 10000ms.

The predictions are confirmed by the statistics generated by an MNSS experiment (given in Appendix C). Thus the fundamental process switching mechanism works for MNSS experiments.

6.2.2 Remote Request Functioning

There are eight distinct phases to the cycle of behavior a user process exhibits in making remote requests:

1. a user process does local activity (processing and long waits),
2. the process initiates a remote request and an appropriate message is generated,
3. the remote request message is transmitted to its destination,
4. a remotelly initiated process is generated,

5. the remotelly initiated process runs to completion,
6. the remotelly initiated process completes and a remote response message is generated,
7. the remote response message is transmitted to the node of the remote request originator,
8. the remote request originator is restarted for local activities.

Remote request behavior can be isolated and verified by an experiment with a CPU bound process making remote requests to a neighboring node (ie. no store-and-forward). Given a remote request interval of 500ms, message sizes of 100 bytes, a link rate of 8000 baud, and a remote process completion time of 500ms, the following cycle can be predicted:

```
Time(ms) > 600 * 200 * 100 * 100 * 600 * 200 * 100 * 100
 Phase   >  1  *  2  *  3  *  4  *  5  *  6  *  7  *  8
```

Using this cycle, system statistics can be derived for a one process, two node system. For example, durning a one minute period this cycle (2000ms) would be repeated thirty times. Therefore local user CPU processing and overhead (phase 1) should be 30x600=1800ms and remote CPU processing and overhead (phase 5) should also be 1800ms. These and other predicted ′phase′ statistics verify the simulation experiment results (Appendix C).

6.2.3 Store-and-forward Functioning

The experiment used in Section 6.2.2 can be modified to show store-and-forward behavior. Instead of having remote requests sent to a node connected directly to the originator´s node, they can be sent to their destination through an intermediary. Phases 3 and 7 of the behavior cycle given in Section 6.2.2 have to be expanded to account for store-and-forwarding:

1-2 ...
3. transmit request message to intermediary node,
3-1. process message and rout it to the appropriate link,
3-2. transmit message to request destination node,
4-6 ...
7. transmit response message to intermediary node,
7-1. process message and rout it to the appropriate link,
7-2. transmit message to response destination.

The remote request cycle time is lengthened to 2400ms (using the Section 6.2.2 system parameters) with the changes to phases 3 and 7 shown below:

```
 Time>..100 * 100 * 100 * 100 *..* 100 * 100 * 100 * 100
Phase>.. 3  * 3-1 * 3-2 *  4  *..*  7  * 7-1 * 7-2 *  8
```

The modified remote request cycle can be used to derive statistics for a system that has store-and-forward behavior. Durning a one minute period the cycle would be repeated 25 times. This means that the store-and-forward system processing overhead (3-1 and 7-1) is equal to 2x100x25=5000ms. This verifies the statistic for system processing in the store-and-forward node generated by simulation (summarized in Appendix C).

Functional verification experiments must be used initially to verify the simulation system and then whenever a change is made to the structure of the simulation model. Therefore an extensive set of experiments is vital to insure correct MNSS operation.

CHAPTER 7

MNSS EXPERIMENTS

Properly implemented, MNSS provides a convenient tool for interesting simulation analysis. The goal of this chapter is to demonstrate the capabilities of MNSS (in particular MNSS/3000 - see Appendix A). An experimental process is presented that meshes with the mechanisms of MNSS. This process, which is essentially the widely aclaimed ´scientific method´, guides the MNSS user in doing effective experimentation. The latter section in this chapter presents an outline for two simulation studies that are interesting in light of commercial computer network applications. Sample results from the two studies, produced using MNSS/3000, are given with brief analysis.

7.1 The MNSS Experimental Process

There are virtually an unlimited number of simulation experiments that can be done using a fully implemented MNSS. From this profusion of experiments the MNSS user must select those that most directly serve his/her purpose. This selection procedure can be formalized into the MNSS Experimental Process which, faithfully used, effectively focuses experimentation in an MNSS simulation study.

The MNSS Experimental Process is summed up by the following list of steps:

1. Form a hypothesis dealing with the behavior of HPC network systems;
2. Design a set of experiments that conceivably will demonstrate the veracity of the hypothesis;
3. Build a number of key experiments that can be used to generate, by way of MNSS experiment modification facilities, the entire set of interesting experiments; and
4. Run the experiments, analyze the results and determine the veracity of the hypothesis.

These steps can be repeated several times with continuing adjustment of the hypothesis to account for experimentally produced information.

The MNSS experimental process can be used to structure a broad range of simulation studies. For example, to explore basic networking structures a hypothesis might be: "Half duplex communications lines require significantly more message buffering for high message traffic than do full duplex lines." This hypothesis can be refined through experimentation to state exactly the relationship between line protocol, line speed, message traffic, and message buffering. Another type of study involves network performance optimization for a particular application. In this case the hypothesis might be:

"For distributed data processing application ′N′ the most cost-effective network configuration is a four node star, with 9600 baud full duplex communications lines, etc." Refinement of this hypothesis leads to an application customized network.

7.2 Two Simulation Studies

The following simulation studies were done using MNSS/3000. The goal of the experimentation was to demonstrate the capabilities of MNSS. Due to the limited nature of the verification of MNSS/3000 (Chapter 6) the results from the studies should be regarded with caution. The results reveal general forms of HPC network behavior and are not predictive of the exact behavior of any particular network. When MNSS/3000 is verified and tuned for a particular HPC system (eg. HP3000) the simulation results generated will be more accurate and specific in showing forms of network behavior.

7.2.1 Incremental Network Expansion

Hypothesis: The incremental processing increase achieved by adding a node to a network is sensitive to the network′s line and node characteristics.

This was an awesome hypothesis to confront head-on, so some simplifying considerations were in order. First, only fully connected networks were examined; each node in the networks of interest had to be linked to every other node by a communications line. Second, the communictions lines in a particular network all had the same specifications (ie. duplex type and line speed). Third, identical standard user loads (based on a standard user process specification) were used at each node. Forth, the parameters varied in defining the user load at a network node were limited to the number of permanent user processes (formally described in Chapter 4, pp. 54) and the remote request level per user process. The permanent user processes were the agents of transaction data processing; they were continuously active (which assumes an infinite supply of transactions to be processed). All were defined with identical local processing characteristics (ie. computation, short wait and long wait requirements). In addition the destination of remote requests originated by a permanent user process had equal probabilities of being any non-local node in the network (ie. a central database application load was not considered).

Based on the hypothesis and the associated limitations a set of experiments was designed with a variety of network configurations, duplex types, line speeds, network processing levels, and remote request levels. Each experiment in the set was specified by drawing parameter values from the following list of alternatives:

1. network configuration - two, three and four node fully

connected networks;

2. duplex type - half and full duplex;
3. line speeds - 1200, 9600 and 19200 baud;
4. network processing level - 12, 18 and 24 permanent user processes in the network;
5. remote request level - low, medium and high.

There are 108 potential experiments that can be built with different combinations of these alternatives.

The low, medium and high ratings for the remote request level are based on the expected processing that occurs locally before a remote request is initiated (ie. the local processing time of a user process from the completion of one remote request to the initiation of another). The ′time between remote request′ function is (1-P)T/P, where P is the probability of a remote request and T is the transaction time. The transaction time is the amount of time (in milliseconds) required to do some basic amount of processing (ie. a transaction). Each ′transaction time′ of processing done by a user process is directed either locally or remotely (ie. a remote request). Single accesses to a database can be classified as transactions, taking a specific amount of processing to complete. The remote request probability is defined for each user process, and is the probability of a transaction needing remote processing (eg. one in five of the transactions originated by user process A requires remote processing; this implies a remote request probability of .2).

Figure 7.1 shows the table of user process specification values for the time between remote requests. For the experiments at hand, a transaction time of 250ms was always used.

7.2.1.1 Experiment Results One

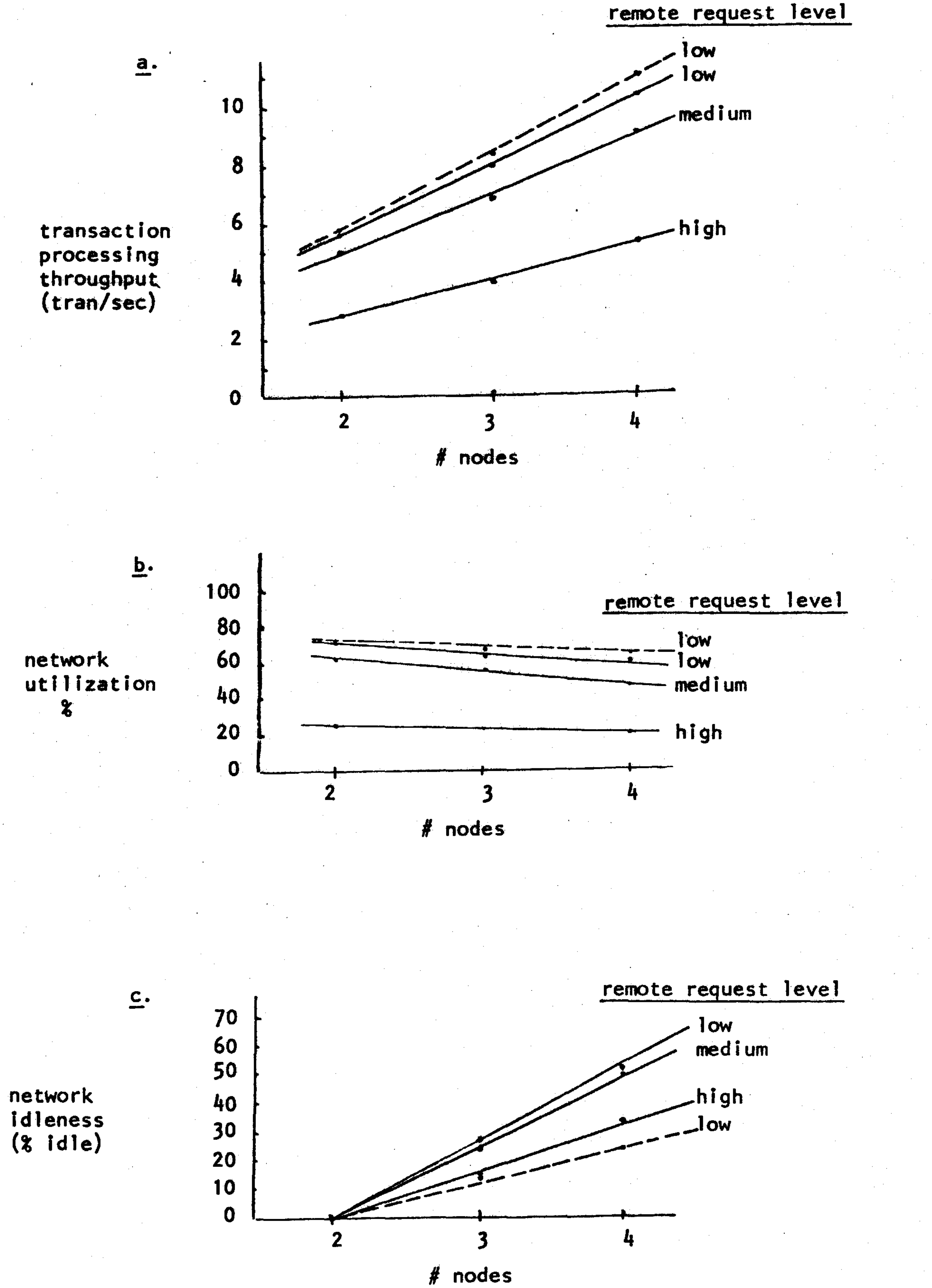
Some interesting experiment results are presented in graphical form in Fiqure 7.2 (a, b and c). The simulation experiments that generated these results were run for a simulated time of 600 seconds. Up to 20,000 events were generated with the running of an experiment (four nodes, high remote request level, 24 user processes). Using a timeshared HP3000 this took 20 seconds of real time. For the results given here all networks were specified to have 9600 baud full duplex lines, and had loads of 12 or 24 user processes.

The processing performance of the networks studied can be measured in terms of transactions per second (ie. the network transaction processing throughput generated by the specified permanent user processes). Figure 7.2a shows the relationships (simulation derived) between the number of network nodes, the remote request rate and the transaction processing throughput. Trend lines have been drawn in to highlight these relationships. These lines illustrate in a very rough way the functions associated with various groups of data points.

	remote request level		
transaction times (seconds)	LOW	MEDIUM	HIGH
.25	2.25	1.00	.25
.5	4.50	2.00	.50
1	9.00	4.00	1.00
2	18.00	8.00	2.00

all values in seconds

Figure 7.1 Remote Request Level Table



------ 12 user processes ------ 24 user processes Lines: 9600 baud, full duplex

Figure 7.2 Incremental Network Expansion Experiment Results

Figure 7.2b puts the transaction processing throughput values shown in Figure 7.2a into perspective. Network utilization is measured as a percentage of the theoretical maximum transaction processing throughput (where there is no system overhead due to communications or process switching). The maximum value for a 250ms transaction time is 1/.250 = 4 transactions per second for one node or 4xN transactions/second for an N node network.

The results given in Figure 7.2a clearly show that network transaction processing throughput increases with incremental network expansion, but that this increase is not proportional to the number of nodes added to the network. As a network increases in size, network transaction processing efficiency (per node) can decrease due to communications overhead or lack of work to do (assuming the number of user processes specified is a constant). Figure 7.2c shows that for the network situations examined here the falling utilization (with incremental growth) is due to some degree to idleness; not enough processing work was available to keep all nodes busy all the time.

A partial solution to the idleness problem is also shown in the Figure 7.2 results. An increase in the number of user transaction processes led to decreased idleness (especially in 3 and 4 node networks) and increased network utilization. This illustrates the sensitivity of network utilization to the character of the network user load.

Another potential solution to the idleness problem was examined and found to be ineffective. This approach entailed increasing the line speeds in the network from 9600 to 19200 baud, while maintaining a user load with 12 processes. If the idleness was due to communications delays then this would decrease idleness and increase network utilization. No significant increase in network utilization was shown when experiments were done; the idleness was not due to communications delays, but rather to delays associated with system processing.

7.2.2 Four Node Network Processing Characteristics

Hypothesis: The total processing capabibility of a four node network is sensitive to node connection characteristics and the applied user load.

This is another sweeping hypothesis that had to be diluted in scope in order to design initial experiments. The simplifying considerations were the same as those of the 'incremental network expansion hypothesis' with two exceptions. First, star and node configurations were used in addition to a fully connected configuration, Second, the remote request destination distributions in some experiments were indicative of a centralized network database application. For these experiments one node was designated the database processor (usually the central node in a star network and an arbitrary node in ring and fully connected networks), and most of the remote requests were directed to this node. In addition the user processes in the database node were primarily processor bound (ie. doing database maintenance) and infrequently initiated remote requests.

The set of experiments based on the simplified 'four node network processing' hypothesis was designed with various network configurations, line speeds, network processing levels, remote request levels and remote request destination distribution types (indicative of centralized and distributed network databases). The experiments in this set were specified using the following list of alternatives:

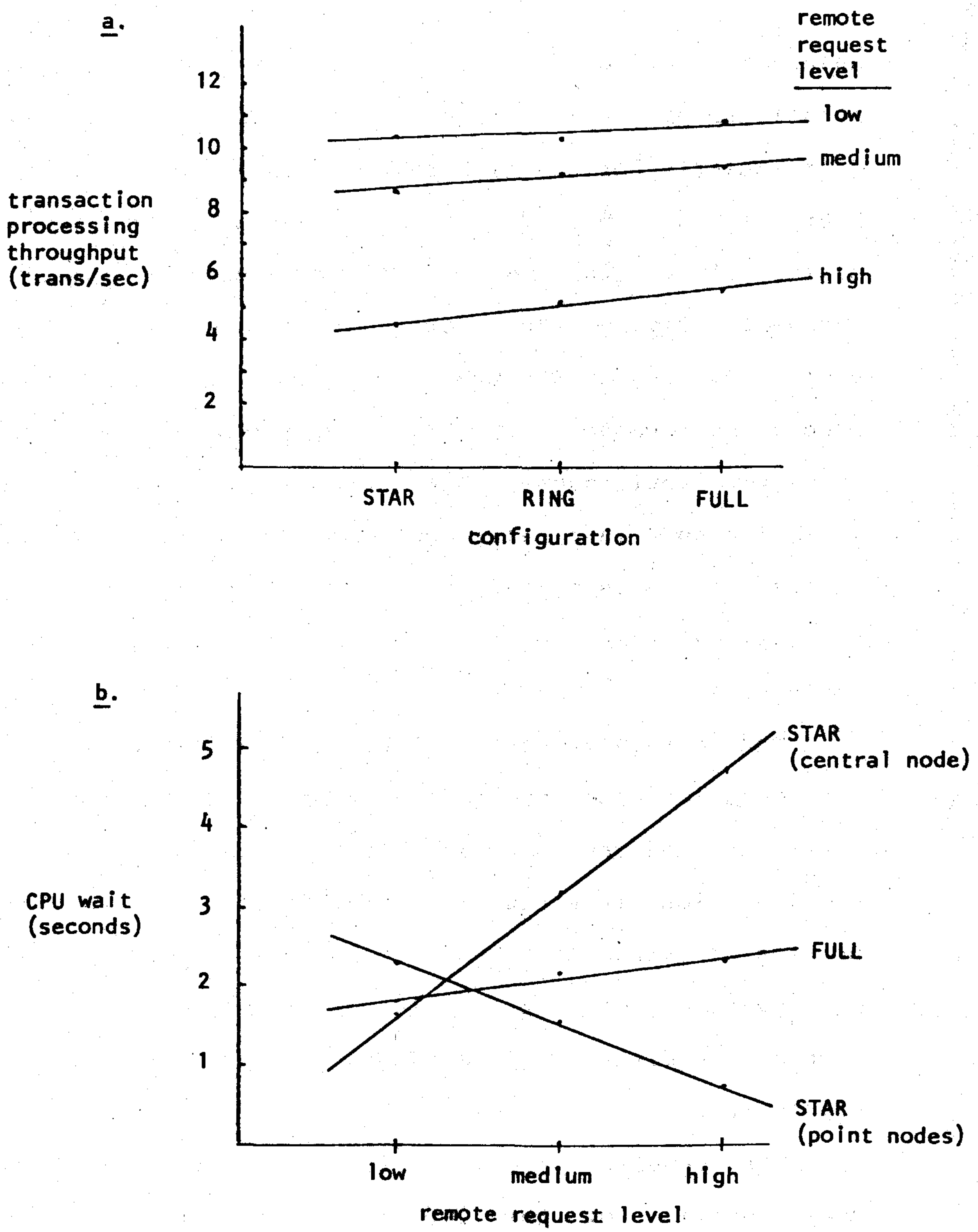
1. network configuration - star, ring and fully connected four node networks;
2. duplex type - full duplex;
3. line speeds - 1200, 9600 and 19200 baud;
4. network processing level - 16 and 24 user processes in the network;
5. remote request level - low, medium and high;
6. remote request distribution - centralized and uncentralized.

Using these alternatives 108 different experiments can be built. The remote request level parameter is specified using the table shown in Figure 7.1, with a transaction time of 250ms.

7.2.2.1 Experiment Results Two

Selected results from ʹfour node networkʹ experiments are shown in Figure 7.3 (a and b). The results given in this section are drawn primarily from distributed database network experiments. The networks were specified with 9600 baud full duplex lines and had loads of 16 user processes. Experiments using centralized network database specifications yielded inconclusive results and need to be supplemented by further experimentation. As with the results described in Section 7.2.1.1 the results presented here were generated by experiments that ran for a simulated time of 600 seconds. In addition, the ʹtransaction processing throughputʹ graph in Figure 7.3a measures the same quantity, in the same units, as the graph in Figure 7.2a.

The results shown in Figure 7.3 graphically illustrate the effect of communications overhead on transaction processing throughput. The effect is most noticeable when there is a high remote request level. With the star configuration there is a higher communications overhead than with a ring configuration, which in turn is higher than the overhead in a fully connected configuration (since in a ring the processor also has to perform message routing for messages



Lines: 9600 baud full duplex   Loads: 16 user processes

Figure 7.3 Four Node Network Experiment Results

destined to other nodes). This is one cause for the relationships of throughput values in Figure 7.3a (particularly for the high remote request level relationship). There is a processing (system and user) bottleneck in the center of a star configuration caused by communications overhead. Figure 7.3b shows how long on the average a user process waits in the CPU queue before it gains control for a processing quantum. The wait in the center of a star network is significantly higher than the wait at the points of a star, or in any node in a fully configured network. At the center of a star there is a great deal of message traffic that results in communications processing and chronic preemption of user processing.

The communications processing disadvantages of the star configuration wane into insignificance when the remote request level is reduced. As is shown in Figure 7.3a, the star configuration is virtually equivalent in terms of transaction processing throughput for a low remote request level. The disappearence of the bottleneck with a decreasing level of remote requests is confirmed by the results given in Figure 7.3b.

The results described in this section reflect only a small portion of the information generated by ´four node network´ experiments. For example, the results of experiments done with central database specifications indicate that a star network can sometimes operate as effectively as a fully connected network even with high remote request levels (this is because most requests are directed to the central node). Further experimentation and analysis is needed to provide a clearer overall picture of network performance with centralized database applications. In any case the results documented here demonstrate what can be done when MNSS is used within the context of a clearly defined network analysis study.

CHAPTER 8

CONCLUSIONS

The development of MNSS led to a number of insights and accomplishments that made the effort worthwhile. The initial careful examination of computer network systems identified a need for analysis tools that could be used to explore the complex behavior associated with these systems. MNSS was then developed to help meet this need.

MNSS is a multifaceted system that includes a modelling discipline, the building blocks for a set of minicomputer network models, and a simulator. The MNSS modelling discipline is simple to use but very powerful as a method of providing abstract representations of system structures. Using the MNSS modelling discipline building blocks were developed for set of minicomputer network models. The principle building block is an HPC system model. The modelled minicomputer networks are collections of HPC systems. The rising popularity of network systems (including HPC network systems) makes an understanding of their behavior a valuable commodity. The MNSS simulator is the principle agent in providing HPC network system performance data. The flexible, easy to use characteristics of the simulator are what make MNSS a worthwhile system analysis tool.

8.1 Limitations Of MNSS

There are a number of limitations to MNSS in its current state of development. These limitations encompass both easily correctable functional shortcomings and also more fundamental design problems.

There is no formal procedure yet defined that can be used to initialize a network model in such a way as to avoid noticable transitory effects when the model is used for simulation. Currently with certain network situations the simulation statistics generated for maximum queue lengths and maximum queue waits reflect the initial setup of the model and not the overall system behavior that is of interest (eg. starting with all permanent user processes in the CPU-wait queue may produce a system state that does not occur under realistic load conditions).

There are two potential solultions to this problem. One solution is to 'phase in' the permanent user load while doing simulation with a system model. This would entail inserting permanent user processes into the CPU-wait state over a period of time instead of all at once when simulation begins. For this to be successful the phase in time would have to be short compared to the total simulation time. A second solution is to begin gathering initialization sensitive statistics only after all startup effects have disappeared. The simulation time after the delay in statistics gathering should be sufficiently long to insure that the final simulation statistics are valid.

The usefulness of MNSS results is now limited by the incomplete process of HPC system model verification. Completion of the verification procedure presented in Chapter 6 will result in proving the model correct for a particular HPC system; this is required before MNSS results can be used with complete confidence. In addition since the HPC system model can be used to represent a class of systems (ie. HP produces a series of computers with related architectures), there must be an adjustment and verification of the model for each type of HPC system to be analyzed in a simulation study.

There are a number of limitations in the current implementation of the HPC model description. These include:

1. the model description lacks a specification for computer system primary memory sizes that can be used in calculating CPU-control process switching overhead,
2. there is no convenient way to model networks of computers with different processing speeds (ie. all computers in a network must either have the same processing speed or the user load specification must be adjusted to account for differences), and
3. the CPU-control process switching overhead does not directly take into account secondary memory to primary memory transfer rates (there should be a parameter to

specify this rate for each system).

These limitations can easily be overcome and will be eliminated with the next stage of MNSS development.

8.2 Extensions To MNSS And Further Study

With additional study and development a number of extensions to MNSS could be made to increase its capabilities. The extensions discussed in this section do not necessarily require major modifications to MNSS as it is currently implemented.

With some changes to the user load specification part of the MNSS simulator, trace data compiled by monitoring a particular application could be used directly to specify the user load for a simulation experiment. This feature would be very useful in doing a simulation study aimed at optimally configuring a network for an existing application (that is the application is running on some available system).

Selective submodel simulation is not now provided by MNSS. By extending MNSS to include this capability, users would have more control in doing simulation studies. MNSS could be used to simulate network message traffic with a full HPC network model or with only the communications submodels (messages would be generated at message sources using frequency distributions). The principle difficulty in implementing selective submodel simulation is how to characterize the universe as seen by the selected submodel. This aspect of the universe must be reduced from an active simulation model to an analytic function or trace data (the most likely solution). This must be done dynamically when a user selects a submodel for simulation.

Models are now described interactively using predefined building blocks made from the basic modelling discipline structures (eg. groups, entities, etc.). These building blocks (eg. processing submodels, half/full duplex communications structures, etc.) are integral parts of an implemented MNSS; there is no possibility of simulating different types of systems (eg. computer networks and supermarkets). Additional study into the conversion of MNSS into a generally applicable simulation system (GASS?) is a worthwhile endevor. In particular, capabilities to interactively describe simulation models using basic MNSS modelling discipline structures can be developed. If the capabilities of MNSS are extended in this way then a new range of simulation possibilities arise. The scope of these possibilities can be determined to some extent by determining the relationship of the MNSS modelling discipline to other discrete modelling disciplines. The MNSS modelling discipline is very general and may include many (or all) of the modelling capabilities of these other disciplines. The potential for enhancement and use of MNSS appears to be virtually boundless.

APPENDIX A

THE MNSS/3000 IMPLEMENTATION

MNSS has been implemented in SPL on an HP3000 computer system. SPL is an ALGOL-like language, which has constructs applicable to structured programming (eg. while ... do ..., if ... then ... else ..., etc.). These constructs are used extensively in the MNSS program for top-down structuring and modularization of function. The HP3000 on which MNSS was implemented supports software such as SPL and mathematical library functions (eg. natural log, random number generator), and hardware such as disks, CRT′s and lineprinters. This support was necessary in order to take advantage of the interactive and simulation capabilities of MNSS.

This implementation of MNSS (MNSS/3000) has several characteristic features that reflect its versitility as a system analysis tool. These features include the selection of MNSS function options from menus of alternatives, intelligent dialog interaction, disk storage of simulation experiments, and hard-copy MNSS information display. These features are needed not only for MNSS/3000, but also for any effective implementation of MNSS.

A MNSS/3000 function menu is prefaced by the statement "SELECT OPTION", and is composed of a list of alternatives with associated numbers (0-n). The user is prompted for a reply by a ">", and selects an alternative by entering a number. If the user enters a number that does not correspond to an alternative in the menu, then "INVALID RESPONSE" is output and the user is again prompted. A valid response will result in the activation of the corresponding alternative. Another form of interaction used in the MNSS/3000-user interface is the responsive dialog. MNSS/3000 outputs a question, prompts the user with ">" and then waits for a response. The response is evaluated by MNSS/3000 and a proper followup is output. The goal of the responsive dialog is to channel information to and from MNSS/3000 without irrelevent (dumb) questioning.

The nature of the display and storage of simulation experiment information are very important in determining the usefulness of any simulation system. MNSS/3000 provides the capabilities to store and retrieve simulation experiment specifications represented as disk files. A user can avoid the trouble of respecifying an experiment each time it is to be run. In addition, there may be standard experiment kernels that can be built upon to produce desired experiments. These kernels can be kept on disk and retrieved whenever necessary. Lineprinter output of MNSS/3000 information is provided at the user's direction to supplement the normal form of interactive output. In many cases this is a desireable alternative to information display at interactive station (eg. CRT's produce no printouts and teletypes often take ages to produce a low quality printout).

A MNSS/3000 experiment is constructed in line with the principles discussed in Section 5.2 (´Representation Of A Simulation Experiment´). It can be used by the simulator to generate behavioral information, saved for future use, or modified to yield related experiment specifications.

APPENDIX B

VERIFICATION EXPERIMENT

```
* * * * * * * * * * * *
*                     *
* NETWORK DESCRIPTION *
* * * * * * * * * * * *


NODE CNT=3
LINK CNT=4

LINK    CONNECT   DUPLEX   RATE    WSIZE   DELAY
 0       0->1      FULL    8000
 1       1->0      FULL    8000
 2       1->2      FULL    8000
 3       2->1      FULL    8000


* * * * * * * * * *
*                 *
* MESSAGE ROUTING *
* * * * * * * * * *


 NODE 0
TO 1 VIA 1
TO 2 VIA 1
NODE 1
TO 0 VIA 0
TO 2 VIA 2
NODE 2
TO 0 VIA 1
TO 1 VIA 1
```

Note: there are no uniform, special, exponential or destination distributions specified for the verification experiment

```
* * * * * * * * *
*               *
* CONSTANT DIS  *
* * * * * * * * *

ENTRY   REFER   VALUE
  0     1007    20000000
  1     1002    100
  2     1001    500

* * * * * * * * * * *
*                   *
* CHARACTERIZATIONS *
* * * * * * * * * * *

ENTRY   REFER   CTIME   CPUT   LONG   MEMC   MSGL
  0     1002     C0      C0     C0     0      C1
  1     1002     C2      C0     C0     1      C1
```

6.2.1 Process Sharing

```
* * * * * * *
*           *
* PROCESSES *
* * * * * * *

ENTRY   NODE   TYPE   CHAR   RDIS   RDES   RCHAR
  0      0      0      0      C0     2       1
  1      0      0      0      C0     2       1
```

6.2.2 Remote Request Functioning

```
* * * * * * *
*           *
* PROCESSES *
* * * * * * *

ENTRY   NODE   TYPE   CHAR   RDIS   RDES   RCHAR
  0      0      0      0      C2     1       1
```

6.2.3 Store-and-Forward Functioning

```
* * * * * * *
*           *
* PROCESSES *
* * * * * * *

ENTRY   NODE   TYPE   CHAR   RDIS   RDES   RCHAR
  0      0      0      0      C2     2       1
```

APPENDIX C

EXPERIMENT RESULTS

```
* * * * * * * * * * * * * * * * * * *
*                                   *
* 6.2.1  PROCESS SWITCHING OVERHEAD *
* * * * * * * * * * * * * * * * * * *



* * * *
*     *
* CPU *
* * * *


             * * * * * UTILIZATION TIMES * * * * *
NODE    SYSTEM    LUSER    HUSER    OVERHD    IDLE     REMPR
 0      0         50000    0        10000     0        0
 1      0         0        0        0         60000    0
 2      0         0        0        0         60000    0



* * * * * * *
*           *
* CPU WAIT  *
* * * * * * *

NODE    MAX-SIZE    MAX-WALL    ENTRIES    TIME*LEN    TOT-WAIT
 0         2          700       100        70000       69400
 1         0          0         0          0           0
 2         0          0         0          0           0
```

```
* * * * *
*        *
* LINK   *
*        *
* * * * *


LINK     CONNECT     IRMIT     IDLE
 0        0->1       0         60000
 1        1->0       0         60000
 2        1->2       0         60000
 3        2->1       0         60000


* * * * * * *
*           *
* LINK WAIT *
*           *
* * * * * * *


LINK     CONNECT    MAX-SIZE    MAX-WAIT    ENTRIES    TIME*LEN    TOT-WAIT
 0        0->1        0           0           0          0           0
 1        1->0        0           0           0          0           0
 2        1->2        0           0           0          0           0
 3        2->1        0           0           0          0           0
```

```
* * * * * * * * * * * * * * * * * * *
*                                   *
* 6.2.2  REMOTE REQUEST FUNCTIONING *
* * * * * * * * * * * * * * * * * * *



* * * *
*     *
* CPU *
* * * *
             * * * * * UTILIZATION TIMES * * * * *

NODE   SYSTEM   LUSER   RUSER   OVERHD   IDLE   REMPR
 0     9000     15000   0       3000     33000  15000
 1     9000     0       15000   3000     33000  0
 2     0        0       0       0        60000  0


* * * * * * *
*           *
* CPU WAIT  *
* * * * * * *


 NODE   MAX-SIZE   MAX-WAIT   ENTRIES   TIME*LED   TOT-WAIT
  0        1         100        30        3000       3000
  1        1         100        30        3000       3000
  2        0         0          0         0          0
```

```
* * * * *
*       *
* LINK  *
* * * * *


LINK    CONNECT    TRMIT    IDLE
 0       0->1      3000     57000
 1       1->0      3000     57000
 2       1->2      0        60000
 3       2->1      0        60000
```

```
* * * * * * *
*           *
* LINK WAIT *
* * * * * * *


LINK    CONNECT    MAX-SIZE    MAX-WAIT    ENTRIES    ITEM*LEN    TOT-WAIT
 0       0->1         1           0          30          0           0
 1       1->0         1           0          30          0           0
 2       1->2         0           0          0           0           0
 3       2->1         0           0          0           0           0
```

```
* * * * * * * * * * * * * * * * * * * * * *
*                                         *
* 6.2.3  STORE-AND-FORWARD FUNCTIONING    *
* * * * * * * * * * * * * * * * * * * * * *



* * * *
*     *
* CPU *
* * * *

             * * * * * UTILIZATION TIMES * * * * *

NODE    SYSTEM    LUSER    RUSER    OVERHD    IDLE    REMPR
 0       7500     12500    0        2500      37500   12500
 1       5000     0        0        0         55000   0
 2       7500     0        12500    2500      37500   0


* * * * * * *
*           *
* CPU WAIT  *
* * * * * * *


NODE   MAX-SIZE   MAX-WAIT    ENTRIES    TIME*LEN    TOT-WAIT
 0        1         100         25        2500        2500
 1        0        0           0         0           0
 2        1         100         25        2500        2500
```

```
* * * * *
*        *
*  LINK  *
* * * * *


LINK    CONNECT    IRMIT    IDLE
 0        0->1     2500     57500
 1        1->0     2500     57500
 2        1->2     2500     57500
 3        2->1     2500     57500



* * * * * * *
*            *
*  LINK WAIT *
* * * * * * *

LINK    CONNECT    MAX-SIZE    MAX-WAIT    ENTRIES    TIME*LEN    TOT-WAIT
 0        0->1         1           0          25          0           0
 1        1->0         1           0          25          0           0
 2        1->2         1           0          25          0           0
 3        2->1         1           0          25          0           0
```

REFERENCES

1. Adkins, G. and Pooch, U., "Computer Simulation: a Tutorial", Computer, April 1977, pp. 12-17.

2. Anderson, J. and Brown, J., "Graph Models Of Computer Systems: Application To Performance Evaluation Of An Operating System", Proc. of the International Symposium on Computer Performance Modelling, Measurement and Evaluation, March 1976, pp. 166-178.

3. Beilner, H. and Waldbaum, G., "Submodel Simulation", Proceedings of the 1973 Summer Computer Simulation Conference, pp. 167-171.

4. Bowdon, E., Mamarch, S., and Salz, F., "Performance Evaluation In Network Computers", Proc. ACM SIGSIM Symposium on the Simulation of Computer Systems, June 1973, pp. 66-75.

5. Chappell, S., et. al., "Functional Simulation In The LAMP System", Proc. of the 13th Design Automation Conference, 1976, pp. 42-47.

6. Chattergy, R. and Pooch, U., "Integrated Design And Evaluation Of Simulation Programs", Computer, April 1977, pp. 40-45.

7. Chou, W. and McGregor P., "A Unified Simulation Model For Communication Processors", Proceedings of the 1975 Symposium on Computer Networks: Trends and Applications, June 1975, pp. 40-46.

8. Conant, G. and Wecker, S., "DNA: An Architecture For Heterogenous Computer Networks", Paper presented at the third International Conference on Computer Communication, August 1976.

9. Cooley, P., "The Underlying Structure Of Simulation Problems And Simulation Software", Eighth Annual Simulation Symposium, 1975, pp. 45-55.

10. Coop, D., "An Analytical Approach To Measurement, Evaluation, And Prediction Of Computer Performance", Ph.D. Diss., Department Of Electrical Engineering, University Of California, Berkley, 1971.

11. Forrester, J., "Industrial Dynamics", M.I.T. Press, 1961.

12. Hoang, H., "A Traffic Simulator For Packet-Switching Communications Networks", Proc. of the 1975 Summer Computer Simulation Conference, pp. 671-675.

13. Ireland, M., "Simulation Of CIGALE 1974", The Forth Data

Communications Symposium: Network Structures In An Evolving Operational Environment, 1975.

14. Ireland, M. et al, "Computer Networks Simulation System", University of Waterloo CCNG Report E-25, May 1974.

15. Jasper, D., "Principles Of Network Design", Proc. of the IEEE Computer Society 1974 Symposium on Computer Networks: Trends and Applications, May 1974, pp. 1-5.

16. Jayakumar, M. and McCalla, T., "Simulation Of Microprocessor Emulation Using GASP-PL/1", Computer, April 1977, pp. 20-26.

17. Linsenmayer, G. and Ligomenides, P., "A General Computer Network Model", Trends and Applications 1976: Computer Networks, November 1976, pp. 155-161.

18. Lynch, A., "Distributed Processing Solves Mainframe Problems", Data Communications, November/December 1976, pp. 17-22.

19. MacDougall, M., "Computer System Simulation: An Introduction", Computing Surveys, September 1970, pp. 191-198.

20. Mahmoud, S. and Riordon, J., "Protocol Condiderations For Software Controlled Access Methods In Distributed Data Bases", Proc. of the International Symposium on Computer Performance Modelling, Measurement and Evaluation, March 1976, pp. 241-264.

21. Merten, A. and Teorey, A., "Considerations On The Level Of Detail In Simulation", Proc. ACM SIGSIM Symposium on the Simulation of Computer Systems, June 1973, pp. 137-143.

22. Nutt, G., "Evaluation Nets For Computer System Performance", AFIPS Proc. FICC, 1972, pp. 279-286.

23. Reiser, M., "Interactive Modeling Of Computer Systems", IBM Systems Journal, 1976 no. 4, pp. 309-327.

24. Schneider, G., "A Modular Approach To Computer Network Simulation", Computer Networks: The International Journal of Distributed Informatique, September 1976, pp. 95-98.

25. Schneidewind, N., "The Use Of Simulation In The Evaluation Of Software", Computer, April 1977, pp. 47-53.

26. Shiino, T., "A New Traffic Simulator For Network Systems - SONET", Proc. of the 1973 Summer Computer Simulation

Conference, pp. 113-118.

27. Svobodova, L., "Computer Performance Measurement and Evaluation Methods: Analysis And Applications", American Elsevier, 1976.

28. Wecker, S., "The Design Of Decnet - A General Purpose Network Base", Paper presented at ELECTRO/76, May 1976.