

1-1-1984

Nial as a Prototyping Tool for Discrete Simulations.

Rickey L. Sell

Follow this and additional works at: <http://preserve.lehigh.edu/etd>



Part of the [Industrial Engineering Commons](#)

Recommended Citation

Sell, Rickey L., "Nial as a Prototyping Tool for Discrete Simulations." (1984). *Theses and Dissertations*. Paper 2114.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

Nial as a Prototyping Tool
for Discrete Simulations

by

Rickey L. Sell

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University

1984

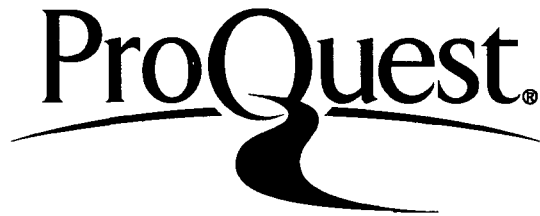
ProQuest Number: EP76387

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest EP76387

Published by ProQuest LLC (2015). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Certificate of Approval

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

August 31, 1984
(date)

Professor in Charge

Chairman of Department

Acknowledgments

Rick Sell gratefully acknowledges all those people who have played a role in bringing this thesis to completion. Special thanks go to his advisor, Professor John C. Wiginton; he not only provided guidance throughout this work but, through his progressive actions, made this thesis possible by introducing the Nial computer language to Lehigh University. Rick's wife, Kathryn, deserves his deepest gratitude for her financial support of their family throughout his education and for enduring the hardships it has caused. Recognition is also given to Rick's parents, Douglas and Marjorie Sell; they offered never-ending encouragement and provided hours of care for his daughter when time became too short. Robert Popiak of Air Products is greatly appreciated for his loan of a computer terminal and modem; without the late-night programming sessions from Rick's home, this thesis would still be in the development stage. Also appreciated is the expertise and cooperation of Doris Lewis who typed this manuscript and organized its contents. Most of all, Rick would like to thank his 3-year-old daughter, Rebecca, for her patience throughout the last year and ability to always make things right; she is truly his best friend and greatest source of inspiration.

Table of Contents

	Page
Certificate of Approval.....	ii
Acknowledgment.....	iii
Table of Contents.....	iv
List of Tables.....	vii
List of Figures.....	viii
Abstract.....	1
1. Introduction.....	3
2. Problem Definition.....	6
3. Background.....	3
3.1 History and Programming Concepts of Nial.....	3
3.2 Computer Simulations.....	10
3.3 Discrete Simulation Concepts.....	11
3.3.1 Systems and Models.....	11
3.3.2 Next Event Approach.....	13
3.3.3 Alternative World Views for Discrete Simulation.....	14
3.4 Event Scheduling Approach Chosen for Nial-Based Simulations.....	18
3.4.1 Details of Event Scheduling.....	18
3.4.2 Additional Features of Event Scheduling.....	22
3.5 Conventional Programming Considerations.....	25
3.5.1 Data Objects--Generation and Manipulation.....	26
3.5.2 Data Structures.....	27
4. Basics of Nial-Based Discrete Simulations.....	34
4.1 Modeling Concepts.....	34
4.1.1 Queues.....	35
4.1.2 Activities.....	36
4.2 Simulation Control Program.....	38
4.3 Data Structures in NBDS.....	43
4.4 Random Variate Generation.....	51
4.5 Data Collection and Statistical Analysis.....	54

5. Data Object Management Efficiently Handled With Nial.....	58
5.1 Entity Records--Creation, Destruction and List Formation.....	58
5.2 Sorting Record Lists.....	62
5.3 Search and Selection of Records From a List.....	64
6. Simulation Elements of NBDS.....	67
6.1 General Format for Operating Rules.....	68
6.2 Symbols for Nodes and Activities.....	69
6.3 Entity Creation and Destruction.....	72
6.4 Queue Selection and Service Selection Nodes.....	74
6.5 SERVICE QUEUE Nodes.....	84
6.6 Activities.....	86
6.7 Resource Queues and Related Simulation Elements.....	89
6.8 ASSIGNMENT Nodes.....	99
6.9 Collection of Supplementary Statistics.....	102
6.10 CONTINUATION Nodes.....	106
6.11 MULTIPLY Nodes.....	107
6.12 MULTIPLE BRANCH Node.....	107
6.13 CONDITIONAL BRANCH Node.....	108
6.14 CLOSE and OPEN Nodes.....	111
6.15 SEED Statement.....	115
6.16 END Statement.....	115
7. Model Building With NBDS Elements.....	117
7.1 Simple Queuing System.....	117
7.2 Computer System With Preemptive Processing.....	118
7.3 Serial Work Stations on a Production Line.....	121
7.4 Traffic Light.....	124
8. General Purpose NBDS Package.....	130
8.1 Input and Internal Organization of Operating Rules.....	131
8.2 Statistical Analysis and Summary Report.....	135
8.3 General Purpose NBDS Examples.....	136
8.3.1 Basic Execution Procedures.....	136
8.3.2 Computer System With Preemptive Processing.....	141
8.3.3 Serial Work Stations on a Production Line.....	144
8.3.4 Traffic Light.....	144

9. Verification of Modeling Elements.....	148
10. Prototyping Special Purpose	
Simulations With NBDS.....	153
10.1 Communications Line Simulation	
Prototype.....	154
10.1.1 Description of Model.....	154
10.1.2 Execution of Program.....	156
10.1.3 Initialization and Input	
of Operating Rules.....	159
10.1.4 Access to Statistical Arrays	
for Summary Report.....	165
10.1.5 Output of Summary Results.....	168
10.1.6 Integration of Operations	
into a Working Program.....	170
11. Conclusions.....	173
List of References.....	176
Appendix A Hierarchical Organization of	
NBDS Baseoperations.....	178
Appendix B Location of Baseoperations in	
NBDS.NDF Script File.....	182
Appendix C Components of NBDS Statistical	
Arrays.....	185
Biography.....	191

List of Tables

<u>Number</u>	<u>Title</u>	<u>Page</u>
4-1	Formulas for Calculating Means and Standard Deviations.....	57
6-1	Symbols for NBDS Modeling Elements.....	70
6-2	NBDS Random and Program Variables.....	73
6-3	Decision Rules Used by Q-SELECT-FWD and Q-SELECT-BHND Nodes.....	76
6-4	Decision Rules Used by SRVR-SELECT Nodes.....	83
6-5	Queueing Disciplines.....	85
6-6	Standard Nial Relational and Boolean Operations.....	110
8-1	Output Statistics of a General Purpose NBDS Run.....	137
9-1	Results of Queueing Model Verification Runs.....	151

List of Figures

<u>Number</u>	<u>Title</u>	<u>Page</u>
3-1	Passage of Time in Next Event Modeling.....	14
3-2	Relationships Between Events, Activities, and Processes.....	15
3-3	Event Scheduling Approach to a Single Server Queueing Model.....	21
3-4	Singly Linked List Structure.....	30
3-5	Removing a Record From a List.....	31
3-6	Doubly Linked List Structure.....	32
4-1	Simple Queueing Model.....	37
4-2	NBDS Simulation Control Program.....	40
4-3	Event Execution Routines.....	42
4-4	NBDS Entity Record.....	48
7-1	Network Diagram of Computer System Model.....	119
7-2	Operating Rules for Computer System Model.....	121
7-3	Network Diagram of Production Line Model.....	123
7-4	Operating Rules for Production Line Model.....	124
7-5	Network Diagram of Traffic Light Model.....	127

<u>Number</u>	<u>Title</u>	<u>Page</u>
7-6	Operating Rules for Traffic Light Model.....	129
8-1	Nial Code for Read_Input Operation.....	134
8-2	Echo Listing of Rules for Computer System Run.....	142
8-3	Summary Results for Computer System Run.....	143
8-4	Summary Results for Production Line Run.....	145
8-5	Histogram for Production Line Run.....	146
8-6	Summary Results for Traffic Light Run.....	147
10-1	Network Diagram of Communications Line Model.....	155
10-2	Input Session From Communications Line Prototype.....	157
10-3	Summary Results for Communications Line Prototype.....	160
10-4	Histogram for Communications Line Prototype.....	161
10-5	Nial Code for INIT_RULES of Communications Line Prototype.....	163

Abstract

Nial (Nested Interactive Array Language) is proposed as a useful tool for prototyping decision-making systems employing discrete simulation. Considered a fifth generation computer language, Nial is based on a mathematical model of data called array theory which provides the definitions for its data operations. Nial is a functional language and is used interactively. Its value in prototyping discrete simulations stems from the ease with which it manipulates data objects within its environment. Discrete simulations demand a great deal of "bookkeeping" in the form of creating, filing, and destroying records; sorting lists; searching lists; and selecting items from a list. The Nial language, with its array-as-data-object concept, is equipped with a rich set of primitive operations ideally suited for carrying out the computer instructions required of those activities.

Nial's functionality, combined with its concise programming capabilities, led to the development of an extensive collection of baseoperations supporting a wide

variety of simulation modeling elements. The individual modeling elements are used as building blocks in developing prototype discrete simulation packages. Represented by symbols, the modeling elements can be combined into a network-like diagram to describe the system of interest. Once defined, the modeling elements are translated into sets of logical-mathematical operating rules which are input to a simulation control program. The control program is supported by the collection of baseoperations and employs an event scheduling approach.

A functional description of each modeling element is given along with the format of its operating rules set. The use of a general purpose simulation package is also described which allows one to experiment with the various modeling elements before developing a specialized prototype. Finally, the process of designing a prototype simulation package is presented through an actual example. Emphasized are the design of an interactive query session as a means for inputting pertinent operating rules, the procedures for generating a summary report, and the organization of the supporting baseoperations into a working program.

1. Introduction

Over the last decade, the information systems and data processing industries have experienced unparalleled growth. As a result of this expansion, system development methods and approaches have been under constant review in an effort to improve the manageability and productivity of development projects. Among the various new methods being proposed in recent years, one in particular stands out--prototyping.

A variety of definitions of the word prototype have been offered in publications dealing with the subject, but Jenkins and Naumann (1) feel Webster's description is adequate:

1. An original model on which something is patterned;
2. An individual that exhibits the essential features of a later type;
3. A standard or typical example.

Other definitions compiled by Canning (2) include "a quick and inexpensive process of developing and testing a trial balloon" and "the first thing or being of its kind."

Whichever definition fits best, one thing is for certain--software prototyping allows end-users a chance to work with the system they are trying to define. With

prototyping, construction of a quick and dirty system begins after the bare minimum of a specification has been prepared. In the end, it has one purpose, and that is to show the users what they are asking for. It gives them some working knowledge of the results that can be achieved by the system they have defined. After definition is complete, the prototype will be discarded and replaced by the operating version of the system. In some cases, a prototype also serves as a useful model to production programmers in designing the logic of the finished product. These benefits are so great, that many DP pundits like Appleton (3) feel prototyping will replace the traditional life cycle approach for developing and maintaining end-user application systems and shared databases.

Prototyping requires software tools that allow designers or programmers to create a working system in a very short time. These resources include such things as on-line interactive systems, database management systems, application development systems, high level languages, generalized input and output software, and libraries of re-usable code. Of all the tools available, the high level languages offer a more responsive tool for most prototype situations because of their interpretive nature and non-procedural code. Because the code is interpreted

and does not require a compilation step, analysts and programmers can perform iterations at a terminal with the user. At the same time, productivity is increased due to the automatic features of the high-level coding. The interactive environment of high level languages also gives users the appearance that information processing resources are physically adjacent and immediately available. User perception of rapid and efficient alteration is what encourages them to discover and evaluate design alternatives.

2. Problem Definition

The objective of this thesis is to demonstrate how Nial, a fifth generation computer language, can serve as a useful tool for prototyping discrete event computer simulations. Nial was chosen for this purpose because it not only satisfies the criteria for high level prototyping languages, but is also equipped with a set of primitive operations that are ideally suited for carrying out the computer instructions demanded by discrete event simulations. In contrast to continuous simulation techniques which rely heavily upon the solution of algebraic, difference, or differential equations, discrete simulations demand a great deal of "bookkeeping" in the form of creating, filing, and destroying records; sorting lists; searching lists; and selecting items from a list. This thesis will demonstrate how easily Nial handles those tasks with its unique array-as-data-object concept.

Numerous examples will be given displaying the conciseness and power of Nial as a programming language for discrete event simulations. However, Nial's real usefulness as a prototyping tool will be demonstrated by presenting a broad set of Nial-based functional units (or baseoperations) that can be combined to quickly build prototype computer simulation packages. The unified modeling approach used to design the simulation elements

will be discussed and a detailed survey of each modeling element given. The procedures for integrating the modeling elements into a working simulation program will be presented through the use of a general purpose simulation package and, finally, the prototyping process itself demonstrated through an actual example.

3. Background

3.1 History and Programming Concepts of Nial

Nial (Nested Interactive Array Language) is a computer language based on a mathematical model of data called array theory. It serves as the model for data manipulated in the system and provides the definitions for the data operations of the language (4). The language was designed through a joint effort of M.A. Jenkins of Queen's University at Kingston, Canada, his co-workers, and Trenchard More of the IBM Cambridge Scientific Center (5,6). Q'Nial, the version implemented on Lehigh University's DECSYSTEM-20 in early 1984 and used in this work, is a portable version of the Nial language suitable for implementation on systems ranging from IBM PCs to the large scale IBM 4341 (7).

Nial is designed as an interactive language with both an immediate execution mode and the ability to execute extensive program texts read from script files or loaded as predefined operations. As an expression based language, the principal unit of computation is an expression that returns a value. Common mathematical notations form the syntactical basis in which expressions are written. Moreover, Nial combines their use with a variety of programming styles ranging from the structured constructs of ALGOL to the recursive style of LISP.

In Nial, the result of an expression evaluation is an array. In the immediate execution mode, the array is displayed as a picture on the user's terminal. The picture shows the structure and content of the evaluation result which aids in understanding the data structure concepts of the language.

Nial may also be used in a command oriented way, in that an expression may be viewed as an imperative by ignoring its result. The language furnishes the syntax for this suppression.

Nial execution occurs in a workspace containing the data and objects defined by the user. For convenience, access to data outside the workspace and programmed interaction with the terminal are provided by system operations for input and output. All text editing is done through the host system interface.

Nial is a functional language which encourages the decomposition of problems into functional units. Each unit is implemented as an operation. Like LISP, it allows creation of new operations and transformers (functional objects that map operations to operations to form new operations), and can treat programs as data.

The value of Nial as a prototyping tool is derived from its use in an interactive environment. Operations can be applied to any kind of data which makes it ideal

for experimenting with problem solving. A new command is entered and immediately the programmer can learn if it was correct. Interpreted like APL and LISP, Nial executes each of its input commands immediately which allows one to observe how the data are transformed by the operation being defined. Furthermore, when all the statements required to solve a problem have been proven, the log of the programming session can be saved and edited. Only those commands that worked are edited into the final program

3.2. Computer Simulations

Computer simulation has been applied to many diverse systems. It has been used for design, procedural analysis and performance assessment for almost three decades and the literature abounds with numerous applications. Areas where computer simulation has been used include air traffic control, communications system design, job shop scheduling, financial forecasting, maintenance scheduling, and water resources development to name just a few (8). Surveys compiled by Shannon (9) revealed that simulation and statistical methods are the most widely-used management science and operations research techniques employed by industry and government.

Computer simulations can be divided into two distinct classes: 1) discrete simulation, and 2)

continuous simulation. Discrete simulation concerns the modeling on a digital computer of a system in which state changes can be represented by a collection of discrete events. Simulated time is advanced from one event to the next and can be a fixed or variable time increment. In continuous simulation, the dependent variables of the model change continuously over simulated time causing smooth changes in the attributes of the system entities. Continuous modeling involves the characterization of the behavior of a system by a set of mathematical equations.

3.3 Discrete Simulation Concepts

3.3.1 Systems and Models

A system is defined as an aggregation or collection of related objects united to perform a specified function. Each object or entity of the system can be characterized by attributes that may themselves be related. For example, a bank, its tellers and the customers all form a system. The teller entities possess attributes of sex, age, experience and salary of which experience and salary may be related. Customers arriving to the bank are also entities and have attributes of sex, age and the type of transaction they are about to request. Any process that causes a change in the state of a system is called an activity. The phrase "state of the

system" describes all of the entities, attributes, and activities as they exist at one point in time. In the bank system, a customer arriving to the bank is an arrival activity. Upon arrival, the state of the bank system changes to reflect the additional person in the bank. If a teller is free to serve the customer, the teller begins a service activity which also changes the state of the system in terms of teller utilization.

Models are descriptions or abstractions of a system. In the physical sciences, models are usually developed based on theoretical laws and principles. The models may be scaled physical objects (iconic models), mathematical equations and relations (abstract models), or graphical representations (visual models). The usefulness of models has been demonstrated in describing, designing, and analyzing systems.

Computer simulation models are mathematical-logical representations of systems which can be carried out in experimental fashion on a digital computer. Therefore, a simulation model can be considered as a laboratory version of a system whose components include the computer, operational rules, mathematical functions, and probability distributions. The behavior of the model is reduced to programmable, logical-decision rules and operations. Such models have also been described as

input-output models (9). That is, they yield the output of the system given the input to its interacting subsystems. Computer simulation models are therefore "run" rather than "solved" in order to obtain the desired information or results. They are incapable of generating a solution on their own in the sense of analytical models but rather serve as a tool for the analysis of the behavior of a system under conditions specified by the experimenter.

3.3.2 Next Event Approach

As mentioned earlier, discrete event simulation on the digital computer involves a system model in which state changes occur at event times. Since the state of the system remains constant between event times, a complete dynamic portrayal of the state of the system can be obtained by advancing simulated time from one event to the next. This timing mechanism is referred to as the next event approach and is used by all modern computer simulation programming languages. By repeatedly advancing to the time of the next event, a simulation is able to skip over the inactive time whose passage in the real world must be endured.

Figure 3-1 illustrates how time is represented and managed when using a next event approach to discrete

simulations. A sequence of events (e_i) are depicted on a

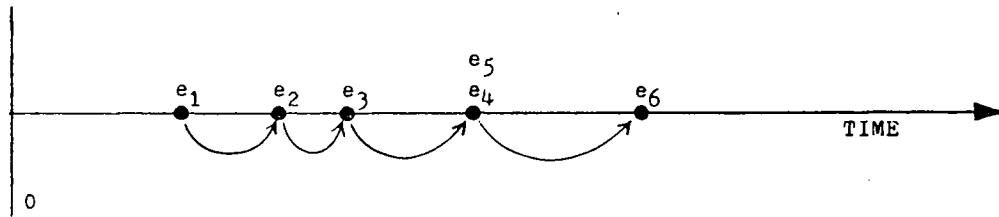


Figure 3-1. Passage of Time in Next Event Modeling

horizontal time axis. The arrows point to the time values at which time is updated and the events occur. Following each event, time is advanced to the exact time of the earliest of all future events. Each event would also represent some change in the state of the system being simulated. For instance, in the bank example presented earlier, an event depicted in Figure 3-1 could represent the arrival of a customer to the bank or the end-of-service and departure of a previous arrival. In the case where there is a simultaneous occurrence of events (e_4 and e_5), e_4 might represent the departure of a previous arrival and e_5 the arrival of a new customer.

3.3.3 Alternative World Views for Discrete Simulation

In developing computer simulation models, the analyst needs to select a conceptual framework for describing the system to be modeled. The framework or

perspective within which the system functional relationships are perceived and described has come to be known as the term "world-view" (10). The world-view employed by the modeler provides a conceptual mechanism for articulating the system description and can be implicitly defined in a simulation language or, where the modeler elects to employ a general purpose computer language, is organized by the modeler himself.

Discrete simulation models can be formulated by: 1) defining the changes in state that occur at each event time; 2) describing the activities in which the entities in the system engage; or 3) describing the process through which the entities in the system flow. The relationships between these concepts are demonstrated in Figure 3-2 by considering the bank system once again. The

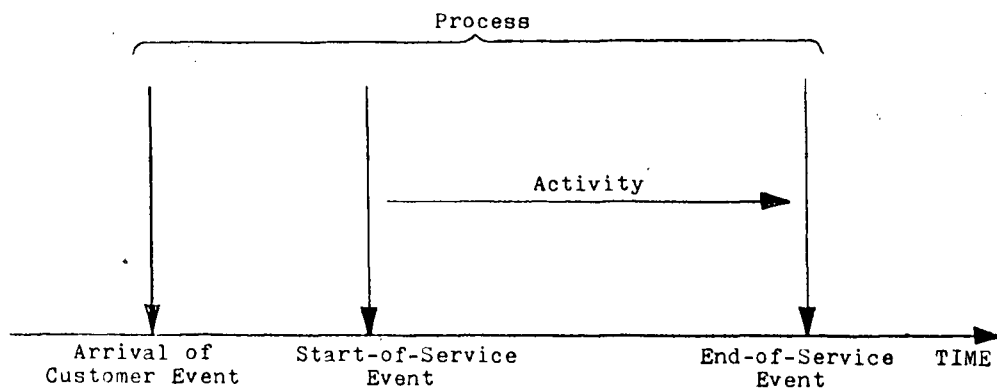


Figure 3-2. Relationships Between Events, Activities, and Processes

arrival of a customer to the bank, the start-of-service for the customer, and the end-of-service for the customer all signify events. As shown, events take place at isolated points in time and bring about a change in the state of the system. Decisions are made at events to start or end activities. An activity is an operation or collection of operations that transform the state of an entity. The service activity in the bank example results in a busy teller and transforms the customer from an arriving to a departing entity. A process is a sequence of events ordered on time and may comprise several activities. In Figure 3-2, the process encompasses the arrival of a customer to its completion of service.

Together, the concepts of event, activity, and process give rise to three alternative world-views for building discrete event models. These are called: event scheduling, activity scanning, and process interaction. The event scheduling and process interaction approaches employ a next-event method of organizing event notices. The principal difference among them is the scanning of simultaneous events which may produce different results if there is some interaction between them.

The event scheduling approach sees a system as a collection of overlapping activities. The beginning and ending of each activity are regarded as separate events

which are independently scheduled. A conditional event can be treated as a sub-event within the event routine that causes its release, or it can be scheduled as a separate, concurrent event. Similarly, if an entity is created and is to be involved in an immediate event, that event might be a sub-event or a separately scheduled event.

The process interaction approach concentrates on the individual entities. The system is seen as a set of overlapping activities, causing events as they start and finish, but the activities form related groups, which are the processes. Once committed to a process, an entity will generally proceed through all the activities of the process. If the end of one activity implies the start of another for the same entity, these two events will be executed in sequence, and not scheduled separately. Similarly, if a non-zero activity is encountered, so that the start of an activity implies its immediate end, those two events will also be executed in sequence. An entity will, therefore, be carried through as many events of a process as presently possible.

The activity scanning approach does not specifically use the next-event method, although the simulation proceeds in uneven steps through successive events. All activities have a statement giving the conditions under

which they may be started, including a specification of what entities and resources must be available. Each active entity has an associated clock giving the time when the entity will end the activity in which it is engaged. Scanning the clocks determines which event occurs next. Following the change of state that occurs, all activities are scanned to see which can then be started.

3.4 Event Scheduling Approach Chosen for Nial-Based Simulations

In designing the Nial-based elements for prototyping discrete simulations, it was necessary to choose a single world-view approach to provide a unified conceptual framework within which any combination of elements could function together. Implicit in this decision was the need to provide just one simulation control program (or timing routine) which determines which event is the next to be selected. With these points in mind, the event scheduling approach was selected to provide this unified perspective and therefore deserves a more detailed examination.

3.4.1 Details of Event Scheduling

Discrete event simulation deals primarily with queueing or waiting line problems. In a queueing problem an arrival occurs and demands that a service be

performed. The system responds by performing the service if it can, or by keeping the demand waiting until it can perform it.

Three considerations play roles in the study of problems using queueing-oriented models: 1) the nature of the jobs to be performed; 2) the resources available to complete a job; and 3) the way in which jobs are selected for service. The nature of jobs includes their frequency of occurrence, the number of tasks per job, the resource requirements per task, and the service time per task. Questions relating to available resources might include number and skill types, the assignment of resources by station, and the assignment of resources to tasks. The way in which jobs are selected is defined by the system's logical operating rules.

In the simplest form of a queueing problem a job requiring service arrives at a facility that has one server. If the server is idle he services the job; if he is busy the job is placed in a queue or waiting line to await later service. The state of the system is defined by the number of jobs (or entities) in the facility at a given moment of time. The queue length is measured either by the total number of jobs waiting for service plus the number of jobs in service or by just the number of jobs waiting for service. A state change occurs every time a

job arrives and every time a job departs. Each arrival and departure is an event.

Two additional events occur in this simple queueing system: 1) when the server becomes busy; and 2) when the server is freed or becomes idle. The events, however, are conditional on the occurrence of an arrival or departure event. For example, an arrival when the server is idle causes him to become busy. A departure, when no jobs are waiting, causes him to become idle.

Figure 3-3 displays a flowchart which describes each element of the queueing problem from an event scheduling approach. As shown, the first thing that occurs at an arrival event is a check on the status of the server. If the server is already busy, the arrival entity is filed in a queue where it waits until the server is freed. If the server is not busy, a service time is determined and the arrival scheduled for departure. Since the departure time coincides with the time that service ends, this time is determined by adding the arrival's service time to the time at which service begins. Service times are attributes of every arrival and may be random or nonrandom. Regardless of their character, a simulation model must provide a mechanism for generating these times. Figure 3-3 explicitly shows a computational block for determining service times just

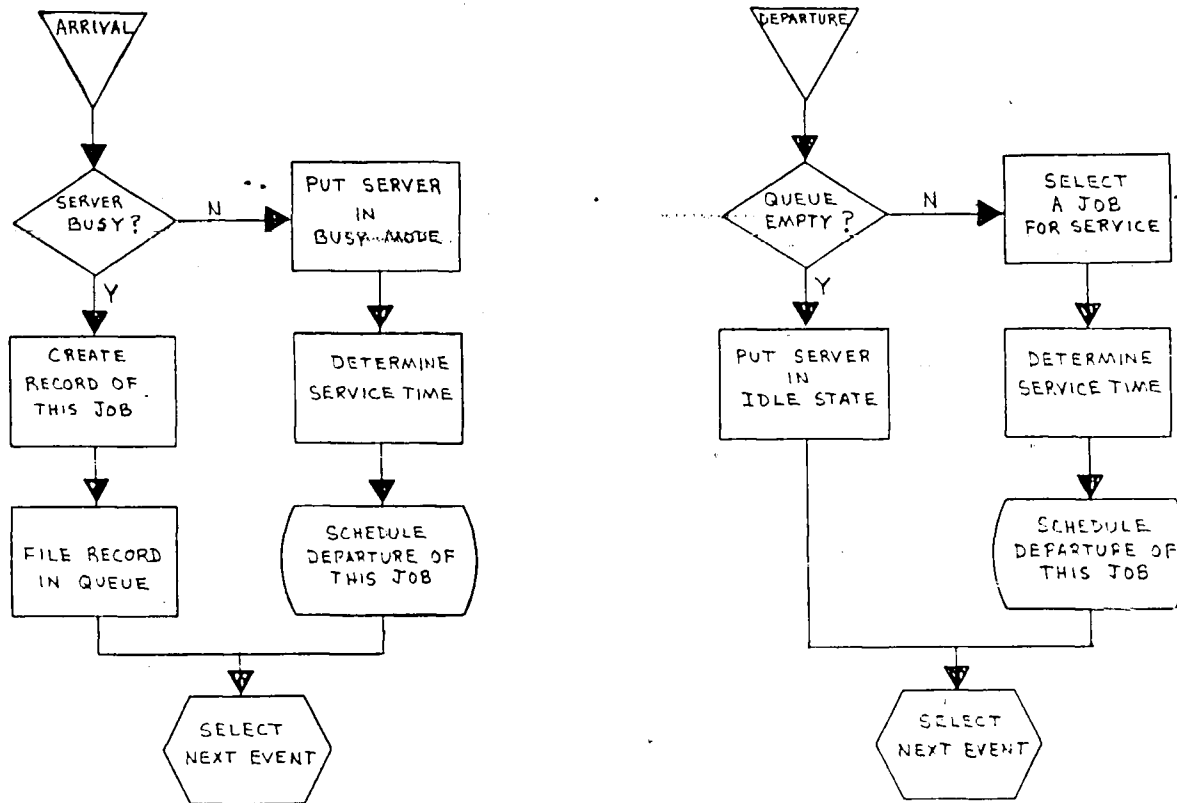


Figure 3-3. Event Scheduling Approach to a Single Server Queueing Model

before the arrival's departure time is scheduled. Service times could also be determined when a job arrives.

Upon processing a departure event, the first instruction is to check whether additional jobs are waiting for service in the queue. If none exist, then the server's status is placed in an idle mode. If more jobs are waiting, then the next job is selected for service, its service time determined, and finally its departure time scheduled.

Note that the final instruction in each event routine is "select next event." This instruction combined with the scheduling instruction forms the basis for making a discrete event simulation work. Whenever an event is scheduled, a record identifying the event and the time at which it is to occur is filed in a special list. When the instruction to select the next event is encountered, the computer simulation searches this list to find and perform the event with the earliest scheduled time. Then simulated time is advanced to this scheduled time, thus skipping the "dead" time. This procedure is the essence of the next event approach to simulation.

3.4.2 Additional Features of Event Scheduling

Whenever the event scheduling approach to simulation modeling is used, a computer program is needed to conduct a search of the list of scheduled events (or

event calendar) to determine which is the next to be executed. This simulation control program has many titles, among which the name timing routine is one of the most common. After every event is performed, control returns to the timing routine, which selects the next event from the list of scheduled events. The event selected is the one with the scheduled time closest to the current simulated time. When time advances to the scheduled occurrence time of the event, control transfers to a code block that executes the steps comprising the event. The code block then transfers control back to the timing routine, which then selects the next event.

As mentioned earlier, a queue is a set of jobs waiting for service. A queue may be thought of as a list from which arrivals are selected for service according to a rule called the queue discipline. For example, jobs may be selected as follows:

1. in the order of their arrival (FIFO),
2. in the reverse order of their arrival (LIFO),
3. in the order of shortest service time,
4. in the order of longest service time,
5. according to a priority number that each job has for service, or
6. at random.

A queue discipline is a rule by which a system operates.

IN A COMPUTER SIMULATION
In a computer simulation this rule is translated into a logical operating rule whose form depends on the queue discipline adopted. For example, adoption of the FIFO discipline means that records are filed in the queue in the order of their arrival times. The next job to receive service is then the first job in the queue. Figure 3-3 provides for this eventual search by including the creation of a record for each job that waits. If selection is a function of a priority number or service time, then that attribute must be part of the job record.

Another procedural feature which needs to be added to the arrival event in Figure 3-3 concerns the generation of additional arrivals. Two methods exist: 1) a sequence of arrivals are prepared in advance of the simulation (no interaction between exogenous arrivals and the endogenous events of the system); or 2) the arrival time of the next entity is determined at the time of arrival of its predecessor. The second method is often referred to as "bootstrapping." It requires keeping only the arrival time of the next entity and is the preferred method of generating arrivals for computer simulations.

The last feature pertains to recordkeeping. As Figure 3-3 appears, all records of arrivals that wait remain in the model. Clearly the list of records grows as a function of simulated time and occupies increasing

storage space in the computer memory. One way to limit this growth is to destroy a record when the corresponding job receives service. The idea of creating and destroying records and searching lists is one of the principal concepts on which discrete event digital simulation is based.

3.5 Conventional Programming Considerations

The concepts of discrete simulation just presented imply a capability to carry out a variety of programmed computer instructions. These include creating, filing, and destroying records; searching lists; sorting lists; selecting members from a list; generating random variates; collecting and analyzing data; and model initialization. By far, most of the work of a simulator is devoted to manipulating various collections of ordered data items, such as event lists and queue files. In a later section of this thesis, it will be demonstrated how Nial, with its array-as-data-object concept, can easily handle these programming requirements. However, in order to appreciate the power of Nial in building computer simulations, this section will detail some of the more frequently encountered data structures used when programming simulations with languages like FORTRAN, PL/1, PASCAL and ALGOL.

3.5.1 Data Objects--Generation and Manipulation

Data structure manipulation is an inherent part of any discrete event simulation. Typical operations during a simulation include:

1. gaining access to the j^{th} record of a list to examine or change the contents of its fields,
2. inserting a new record after the j^{th} but before the $j + 1^{\text{st}}$ record,
3. deleting the j^{th} record from a list,
4. determining the record count in a list,
5. ordering records in a list in ascending order based on the values stores in specified fields, and
6. searching the list for records with given values in certain fields.

Every computer simulation program contains data structures that represent objects of different classes. More specifically, the data structures consist of records of the objects in the simulation, each record containing information regarding the characteristics of a distinct object. A simulation operates on these records as simulated time elapses.

Some simulation programs are designed to deal only with fixed data structures that are allocated either during compilation or at the start of execution. These structures represent fixed numbers of objects of different classes. Other simulation programs are written

to allow both fixed and varying numbers of objects. The way in which a simulation program handles the generation of objects is related to the way it sees a system through its world-view and to whether its base-language is compiled or interpreted.

3.5.2 Data Structures

Rigid and dynamic data structures have important implications for simulations. In FORTRAN, a DIMENSION statement reserves storage space. For example, the statement DIMENSION ATRIB (20,10) instructs the computer to reserve a block of $20 \times 10 = 200$ storage locations for the array called ATRIB. Once the 20×10 memory locations are allocated to ATRIB, they remain so until the program terminates.

Rigid data structures like the DIMENSION statement do not always provide an efficient means of utilizing memory space in the computer. For example, consider a simple simulation problem in which entities travelling through a system collect varying numbers of attributes. Using the same 20×10 arrangement, let ATRIB (K,1) denote the first attribute (eg. arrival time to the system) of the K^{th} arrival and ATRIB (K,2), ... , ATRIB (K,10) the remaining nine attributes. Depending upon the logical routing rules through the system, a particular entity might collect just one attribute or up to all ten. The

straightforward use of a dimension statement in this case would result in the following disadvantages:

1. not all of the column positions would be utilized, thus wasting memory space,
2. it may not be necessary to keep the attribute records of entities that have left the system, and
3. the number of arrivals to the system would be limited, in this case, to 20.

● Crude Data Structure

One method of increasing the utilization of available storage space in the previous example is to vacate the registers once the information is no longer needed and make it available to new arrivals. For example, at the start of the simulation, all the registers are set to zero indicating they are empty. Let the first attribute of each K^{th} arrival, $\text{ATRIB}(K,1)$, hold its nonzero order of arrival. As entities arrive to the system, $\text{ATRIB}(1)$, $\text{ATRIB}(2)$, $\text{ATRIB}(3)$, ... is checked until a K ($K \leq 20$) for which $\text{ATRIB}(K) = 0$ is found. The order number of that entity is then placed in $\text{ATRIB}(K,1)$ and the K^{th} row of storage space reserved for that entity during its lifetime in the system.

When an entity leaves the system, its row of storage space is located in the ATRIB array and cleared by setting each register to zero. This release of space allows those positions to be used again. Although the

maximum number of entities in the system at one time cannot exceed 20, it is clear that the number of entities processed can be considerably greater than 20.

The benefit of space conservation in this example is obtained, unfortunately, at the cost of added computer time. Note that, upon an entity's arrival, the registers must be checked sequentially until an empty set is found for storage. A sequential search must also be conducted when attempting to locate a particular attribute or set of attributes based on order (e.g. minimum service time). In a system with rapid state change and many entities, the computing time consumed in a search of this kind can be expensive.

- Ordered Chains

One way to reduce the search time of ordered arrays is to add information to the record of an entity which points it to the record of the next ordered entity. For instance, if the ATRIB array in the previous example must be ordered by increasing time of arrival to the system, ATRIB (K,1) could contain the arrival time of that entity while ATRIB (K,2) would hold the register address of the next arrival to the system.

This is an effective programming technique for handling records like these and embodies the principles of list processing (11). The records are said to be

chained together or in a list. In general, the records of the entities in the ordered chain are identified in the computer memory by an address. If the records have more than one word, the address is assigned to one of the words, such as the first. One word, or field in a word, called a pointer, is set aside in each record for the purpose of constructing the list. In addition, a special word called the list header is provided for entering the list.

The records are chained together into a structure known as a singly linked list and is illustrated in Figure 3-4. The list header contains the address of the

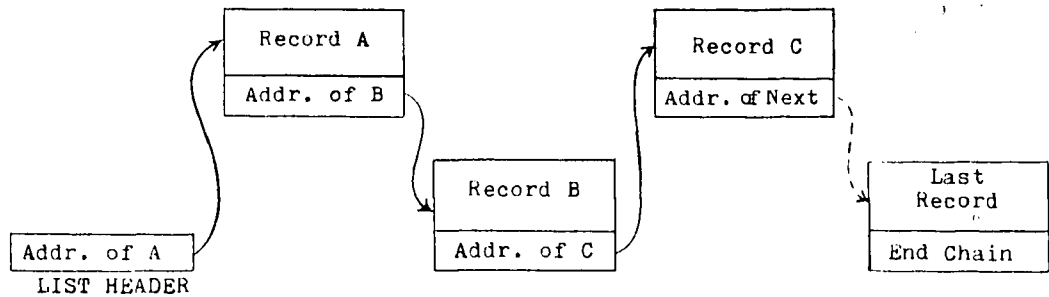


Figure 3-4. Singly Linked List Structure

first record in the list. The pointer of the first record contains the address of the second record, and so on down the list. The last record in the list contains a special end-of-chain symbol in the pointer space to indicate that it is the last member. If the list happens to be empty,

the list header contains the end-of-chain symbol.

Beginning from the header, a search is able to move down the list by following the chain of pointers. If the program needs to remove a record from the list, say record B from list ABC, it simply changes the pointer in A to point C, as illustrated in Figure 3-5.

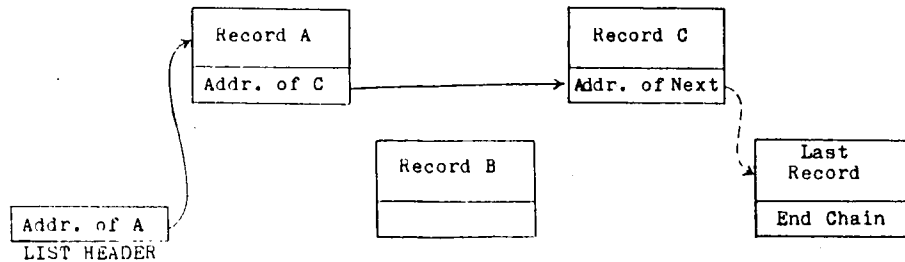


Figure 3-5. Removing a Record From a List

Correspondingly, to insert a record into the list, (for example, to put Z in the list ABC between B and C) the pointer B is set to Z and the pointer of Z set to C. Re-sorting is achieved by a series of removals and insertions.

With a first-in-first-out rule of ordering records, it is also convenient to keep a list trailer that has the address of the last record, because new additions are made at the end. The trailer record avoids the necessity of working along the list to find the last entry. The trailer will also contain the end-of-chain symbol when

the list is empty.

While it is easy to insert a new record after a given record in a singly linked list, it is not so easy to do that before a given record. This is because each record points to its successor, but not to its predecessor. A similar observation applies to deletions: to remove an element from the list one needs a reference to its predecessor.

These difficulties can be avoided by adding a second pointer to the records pointing to the record preceding it (if any). The result is a doubly linked list and is illustrated in Figure 3-6. Now, given references to the

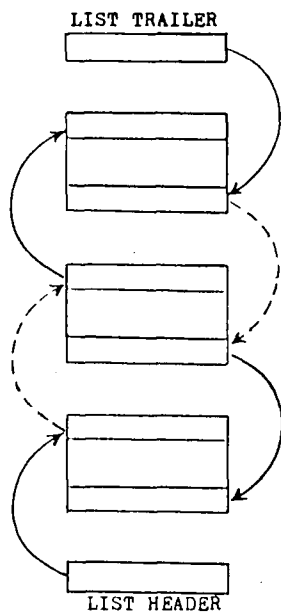


Figure 3-6. Doubly Linked List Structure

list header and trailer, the list can be traversed in

both directions; insertions can be made either before or after any given element in the list; any element can be removed from the list without having to know its successor or predecessor.

Other, more complicated data structures exist that improve the efficiency of handling lists. Directed graphs, binary trees, and heaps are included in these (12). However, they all have one thing in common: while increasing data storage space, the amount of computer time spent checking for data elements is greatly reduced. Even though execution time is minimized, there still exists the drawback of having to reserve space at the beginning of a program and the need not to exceed the specified array size. As a result, many special purpose discrete simulation languages are implemented with programming systems that have list processing and dynamic data structuring capabilities.

4. Basics of Nial-Based Discrete Simulations

The previous section touched on the basic concepts of data structures and their manipulation when programming discrete simulations with general purpose computer languages. The next section will demonstrate how those tasks are concisely and efficiently handled using Nial. However, before those examples are presented, some basic modeling and design concepts of Nial-based discrete simulations (NBDS) need to be described first.

4.1 Modeling Concepts

As stated earlier, event scheduling was chosen as the world-view approach to designing the elements of discrete simulation presented here. While this approach is embedded in the simulation control program and supporting operations, the task of modeling a prototype simulation actually employs a process-oriented perspective. Each element of the process, such as a queue or server, can be represented pictorially using symbols. When these elements are combined together they form network-like structures, similar to those employed by SLAM, a special purpose simulation language (13). Each element of the network represents a set of mathematical-logical operating rules that are provided by the modeler when building the simulation. As entities enter the system, they flow through the network as

prescribed by the operating rules of the particular element they encounter.

To provide the tools for quickly developing a prototype simulation, several different kinds of modeling elements were designed and translated into Nial-based operations. A detailed description of the elements will be presented in a later section of this thesis. However, to expand the modeling concepts being presented here, the basic modeling elements of queues and activities will be introduced.

4.1.1 Queues

Two types of queues can be modeled in these simulations. The first one is known as a service queue and represents an area or site to which entities arrive and request the service of a single resource. If a service entity is not available, the arrival waits in the queue until one is freed.

The second type of queue is known as a resource queue. Resource queues are similar to service queues except entities arriving to them can request the service of discrete or variable amounts of resources. If the requested amount of resources is not available, the arrival waits in the resource queue until they are relinquished. Unlike service queues which are associated with a single service activity devoted specifically to

that queue, resource queues can share a bank of resources with other resource queues. In addition, the units of a resource available for use by resource queues can be altered during the computer simulation. This option is not available with service queues where the number of entities serving them remains fixed throughout simulated time.

4.1.2 Activities

Like queues, there are two types of activities--service activities and regular activities. Both allow entities to flow through them to other elements in the network. The passage of an entity through an activity can be delayed for a prescribed period of time although regular activities can be used with no time delays.

The major difference between the two types of activities is the number of concurrent entities they allow to pass through them. Service activities limit the number of entities flowing through them at one time to the number of servers represented by the activity. On the other hand, regular activities have no restriction on the number of entities that can simultaneously flow through them. Service activities are also used exclusively with service queues while regular activities can direct the flow of entities away from any other kind of modeling

element. In particular, regular activities are used to delay the time resources and are utilized by an entity acquiring them at a resource queue.

As an example of how queues and activities are used in the modeling process, Figure 4-1 illustrates a network diagram of a simple queueing model. Note how node-like

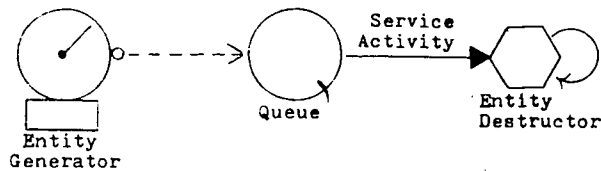


Figure 4-1. Simple Queueing Model

symbols are used to represent the entity generator, the service queue, and the entity destructor while a branching or connecting symbol is used to represent the service activity (the branching symbol is also used to represent regular activities). Various symbols will be used to represent other types of nodes which include milestones and decision points.

The network pictured in Figure 4-1 represents an entire process. Entities arrive to a service area where they are either served immediately or wait for service in the queue. Upon completing the service activity, the

entities depart the system as represented by the entity destructor or termination node. Service and regular activities represent the time delay that an entity encounters as it flows through the system and are two elements responsible for the advance of simulated time in the model.

4.2 Simulation Control Program

The queueing model pictured in Figure 4-1 represents an entire process through which entities flow. However, the computer program which controls that process is designed to sequentially select event notices from an event calendar and execute blocks of code or base-operations corresponding to that particular event or element of the process. The event notices contain both information which is used to transfer control of the program to appropriate operations and information which is used to reference a look-up table of operating rules provided by the modeler before execution of the simulation begins. The operating rules define the unique characteristics of each element of the model and are referenced during events in which those elements are involved. For example, if the next event on the calendar is an arrival to the queue node shown in Figure 4-1, the operating rules for that node are referenced to determine what service activity serves that station and whether or

not the server is busy. If the server is already occupied, the rules would be referenced to determine the queueing discipline for that queue, the maximum number of entities allowed in the queue, etc. Together that information determines what transactions take place during the event and controls the directional flow of the program. Upon completing those transactions, control returns to the timing routine, the next event is selected from the event calendar, and the process repeated until a termination notice is detected. The simulation can be terminated at a given point in simulated time or after a certain number of entities have been processed. Upon detecting a termination notice, the program updates time-persistent statistics and then generates a summary report.

Figure 4-2 presents a flow chart diagram of the NBDS simulation control program. The above discussion includes everything after the event selection block. The first three blocks include all of the initialization steps; here various constants, program variables, and flags are set to their starting values and the operating rules established. Finally, the last executable block before the loop structure initializes the event calendar. Here each set of operating rules for the generation nodes in the model is scanned and the first entities scheduled for

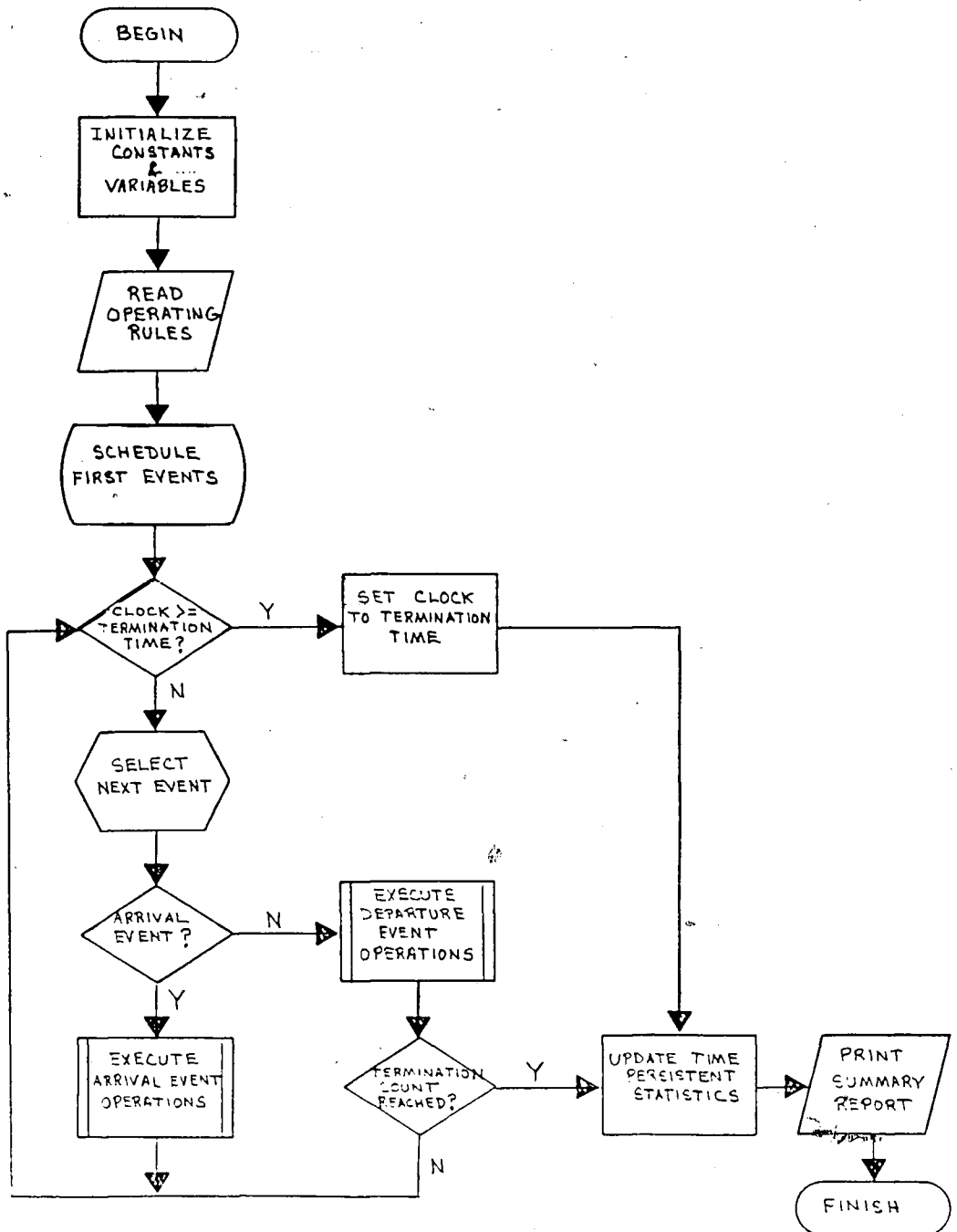


Figure 4-2. NBDS Simulation Control Program

arrival. The operations in this block also establish queue nodes with initial entities in the queue if indicated.

Every simulation program has a three-level hierarchial structure in which the simulation control program occupies the top level and housekeeping functions such as the collection of statistics and generation of random variables occupy the lower level. The middle level is occupied by the arrival and departure routines and the associated operations which process the individual events. These two routines are depicted in Figure 4-3. Note the "bootstrapping" technique for generating the next arrival from a previous arrival in the arrival routine. Except for queue nodes, arrivals to nodes always result in the scheduling of a departure event (in some cases where a zero-time activity follows, the scheduling of a departure is skipped and an arrival to the next node scheduled instead). In the case of an arrival to a queue node, a departure event is scheduled only if the necessary resources or servers are available. Otherwise, the arrival is filed in the queue (see Figure 3-3).

In the departure event, the first task determines whether the departure is from a service activity or a regular activity. Once that has been determined, the appropriate set of operating rules is referenced and the

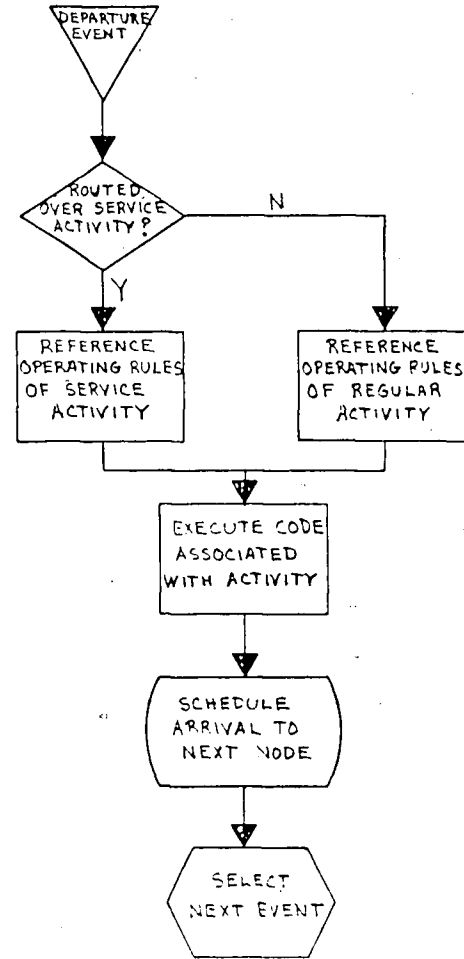
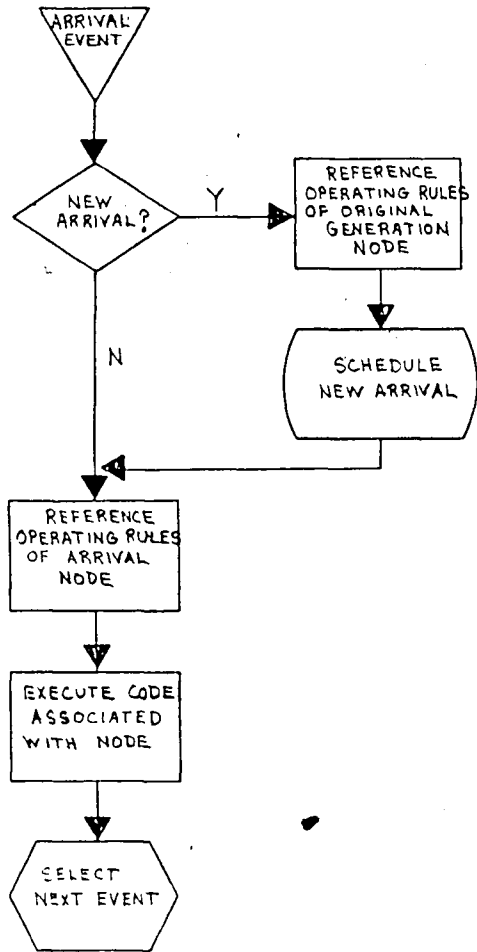


Figure 4-3. Event Execution Routines

code for the given type of activity executed. Unlike the arrival routine, an event is always scheduled at the end of a departure. In this case it is an arrival to the destination node pointed to by the activity. While not detailed in Figure 4-3, a departure event might also lead to the scheduling of another departure. This could result from a service activity which has been freed to service a queue that holds waiting entities. As shown in Figure 3-3, the next entity is selected for service, its service time determined, and the entity scheduled for a departure.

4.3 Data Structures in NBDS

- Statistical Arrays

Two basic data structures were employed in building the simulation operations of this thesis. The first structure resembles a FORTRAN-like one- or two-dimensional array which is used to maintain statistical data and flags associated with queue nodes, service activities, etc. Each row-major ordered array belongs to a particular class of elements and each row of that array belongs to a given element within that class. For two-dimensional arrays, the total number of rows is determined during the program initialization steps and is dependent upon the number of modeling elements in the system. However, the first row of each matrix is never

used to hold data in order to maintain a logical correspondence to the element number to which those data belong. This is due to the addressing feature of Nial in which the first element of an array is address 0. For example, the address (0 2) refers to the third element in the first row of a matrix. Since the modeling elements in the program are logically ordered (e.g. if there are three service activities they are known as servers 1,2 and 3), the first element owns the second row of the matrix and so forth. While the first row of each matrix represents wasted workspace, a logical relationship between element numbers and the position of its data in the matrix is maintained. It also eliminates the need for a costly base-address-plus-offset calculation each time the data for a particular element are referenced in the simulation program.

Nial conveniently establishes and initializes arrays with just one primitive operation, the "reshape" operation. As an example, consider the following expression and its evaluated result pictured in the sketch mode:

```
ACTSTATS:= 4 6 reshape 0
```

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

In the simulation operations to be presented later, the array Actstats is used to maintain statistics on all regular activities. Therefore, the array created above would represent storage space for three activities (including the dummy first row).

Direct assignments can be made to any member of an array in a FORTRAN-like fashion. For example, observe the effect of the following operation on the array Actstats:

```

ACTSTATS @ (1 1) := 45

0 0 0 0 0 0
0 45 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

```

Similar insertions are carried out with the "place" and "placeall" operations which can replace single and multiple items in an array respectively. Observe the use of the latter operation in the example below:

```

ACTSTATS := (10 10 5 9) (3 cart 0 1 2 3) placeall ACTSTATS

0 0 0 0 0 0
0 45 0 0 0 0
0 0 0 0 0 0
10 10 5 9 0 0

```

Selections from an array are carried out with operations corresponding to those demonstrated above:

```

ACTSTATS @ (1 1)

45

(3 cart 0 1 2 3) choose ACTSTATS

10 10 5 9

```


The selection and insertion operations demonstrated in the previous examples provide the mechanism by which most of the data are manipulated in the statistical arrays supporting the various NBDS simulation elements. They're especially useful in combination with each other where one operation selects data from a statistical array and passes them as arguments to a computational operation; the computational operation then returns the same array with new values which are inserted back into the old array.

- Entity Records

In Section 3.5, the basic data structures and bookkeeping concepts involved in programming discrete simulations with general purpose computer languages was presented. Demonstrated was the complex and cumbersome need to provide pointers for each record in ordered lists serving as queue files or event calendars. In NBDS simulations, this task is totally eliminated. The array-as-data-object concept of Nial combined with the entity record structure designed for NBDS produce a highly efficient means for manipulating the data objects within the program. As a result, elements of the simulation and their associated operations are quickly and easily designed--a critical factor of the prototyping process.

In the simulation operations presented here, a single record is created for each entity as it enters the system. The record is a solitary array (or list) of ordered data items which are program and user-defined attributes of the entity. Certain attributes play an important role during the top level execution of the simulation control program while others are only accessed during event routine operations. The entity record's lifetime in the system is spent either on the event calendar or filed in some queue awaiting service. The most important aspect of the entity record is that it "flows" through the simulation program as a complete unit, much like an entity with all its attributes would in a real-world system. This aids the programmer greatly in conceptualizing the simulation element under design.

There are a minimum of eight program-defined attributes in an entity record which occupy addresses 0,1,2, ... 7. All but one of these attributes change dynamically during the simulation run. An entity record can also contain an unlimited number of user-defined attributes beginning at address 8. Thus, from run to run, the length of the entity record can vary but, once set during the initialization steps, remains fixed throughout an individual simulation run.

Figure 4-4 displays an example of an entity record

0	1	2	3	4	5	6	7	8								
:	D	:	14.8	:	79.5	:	QUE1	:	SRV1	:	14.8	:	:	:	30.	:

Figure 4-4. NBDS Entity Record

containing the eight basic attribute positions common to all records plus one user-defined attribute. The contents of the first eight fields could describe the state of an entity record on the event calendar of a run simulating the simple queueing problem depicted in Figure 4-1. The following definitions are identified by an address and summarize what type of attribute each field holds and its relevance to the simulation program:

① The first field indicates whether the entity is scheduled for a departure or an arrival. In this example the character D indicates the entity is scheduled for a departure. The character strings A0 and A1 are used to indicate a freshly generated arrival to the system or a later arrival to a node somewhere else in the system respectively. This field is referenced each time an event is selected from the event calendar to determine which event routine is to be executed.

② This field holds the generation time or time the entity first entered the system and is the only

program-defined attribute that remains fixed throughout the simulation. The example in Figure 4-4 indicates the entity was generated at time 14.8.

② The third field contains scheduled event times and is the field on which the event calendar is ordered. In the example given, the entity is scheduled to depart a service activity at time 79.5. Each time an event is selected from the event calendar, this field is referenced and simulated time advanced to the value contained there.

③ At the time of generation or at the end of each departure event, an arrival is scheduled and the destination of that arrival entered into the fourth field of the entity record. That field is then referenced at the start of each arrival event to determine which blocks of code will be executed next. In the given example, the four-membered character string QUE1 indicates that, prior to its scheduled departure, the entity had arrived at service queue #1.

④ Whenever an entity's departure is scheduled from a node, the activity over which it is to be routed is entered into the fifth position of the entity record. This field is then referenced during a departure event to determine which type of activity is ending as well as its activity number. The character string SRV1 in the sample

record indicates that the entity is being served by service activity #1.

⑤ The next field contains the simulated clock time at which an entity arrives to a node. This value is always entered when an entity is newly generated or at the end of each departure event. It is used to calculate the time an entity has been waiting for service in a queue. As shown in Figure 4-4, the value of this time is 14.8, the same time the entity entered the system. ^a

⑥ & ⑦ The next two fields are utilized only when resource queues are specified in the model. Since the example in Figure 4-4 is only modeling a service queue, these two fields would remain empty throughout the simulation as indicated. The first field at address 6 is reserved to hold the identification number of the last resource queue encountered while the next field is used to indicate the number of resources acquired at that queue and currently being utilized by the entity. These two attributes play an important role in the preemption of entities utilizing a particular resource which will be detailed later in the thesis.

⑧ The last field pictured in Figure 4-4 holds one of a possibly infinite number of user-defined attributes. While not yet discussed, entities can acquire user-defined attributes anywhere in the network by

passing through ASSIGNMENT nodes. ASSIGNMENT nodes compute the value of a given attribute using program variables, simple mathematical expressions, or by accessing one of the program random variate generators.

While not included in the network pictured in Figure 4-1, suppose an ASSIGNMENT node preceded the service queue and was used to compute the service time for that entity. Assuming this was the first user-defined attribute, it would be stored in field 8 and its value referenced at the time that entity would be scheduled for service. In the given example, field 8 contains a service time of 30.0 time units. Since the entity is scheduled for a departure at time 79.5, a quick subtraction indicates that service began for this entity at time 49.5. Furthermore, subtracting the time contained in field 5 from the start-of-service time reveals the entity waited in the service queue for 34.7 time units before receiving service.

4.4 Random Variate Generation

Since the modeling elements presented in this thesis are designed to simulate stochastic processes, a mechanism must be available for generating random variables. This requires a source of uniformly distributed random numbers which, in turn, are transformed into a new set of random variates from a

variety of continuous and discrete probability distributions. Independent samples that are uniformly distributed in the interval (0,1) provide the basis for generating samples from all other distributions. Nial provides a utility for generating random integers which, when divided by the largest value of the specified interval, would produce random numbers between 0 and 1. However, the random number generator of the version implemented on the DECSYSTEM-20 and used in this work (Q'Nial, release 1, version 3.02) did not function. Therefore, a new operation was built to provide uniformly distributed random numbers between 0 and 1.

A wide variety of methods have been developed for random number generation. One of the more common ones is the multiplicative congruential generator which was the choice for this work. It has the form:

$$Z_{i+1} = a \cdot Z_i \pmod{m} \quad (1)$$

$$r_{i+1} = Z_{i+1}/m \quad (2)$$

where Z_0 is the initial seed value and r_i is the i^{th} pseudorandom number. It provides a maximal period of 2^{B-2} before recycling occurs on a computer with B bits/word. The assignment of values to the constants a and m and the seed value Z_0 has been the subject of a great amount of research. Fishman (14) presents a widely accepted set of procedures for choosing those values that was the basis

for designing the random number generating operation used in this work. In addition, Deo (15) demonstrates how the correct choice of constants can save the modulus division step during the computation. The guidelines proposed by both authors resulted in a random number generator having the form:

$$Z_{i+1} = .189277 * Z_i \quad (3)$$

$$r_{i+1} = Z_{i+1} / 34359738368 \quad (4)$$

The value of m is one greater than the largest integer held in one word by the DECSYSTEM-20. Its choice makes the division of the product a Z_i unnecessary for the modulo operation in the original equation. A machine word cannot hold an integer larger than $(m-1)$. Therefore, as soon as the product exceeds $(m-1)$, overflow would automatically occur, leaving only the remainder. In the DECSYSTEM-20 no action is taken when this occurs but overflow sets the sign bit and the result becomes negative. Therefore, the result is simply adjusted by taking the additive inverse of Equation 4.

The NBDS operation which contains this random number generator is named RANNUM. Any odd integer can be used as an initial seed value for RANNUM which is provided at the start of a simulation run. RANNUM actually employs two random number generators of the form just given. However, they are seeded with different values and alternate

between each other when RANNUM is called upon to generate a random number. The use of alternating generators helps reduce the potential for nonrandomness when a 2-tuple of independent uniform random numbers is called for (14).

With the means to generate uniformly distributed random numbers from 0 to 1, transformation algorithms were built into operations to generate random variates from several probability distributions. They include the exponential, uniform, Erlang and normal distributions and were all adopted from FORTRAN-like algorithms presented in various simulation textbooks (10,15,16). Their use as a modeling element will be detailed in a later section.

4.5 Data Collection and Statistical Analysis

Each of the basic simulation elements presented earlier are supported in the event routines with baseoperations designed solely for the purpose of collecting data. Those operations act upon statistical arrays maintained for each of the elements in the model as demonstrated earlier. Those data are automatically collected during each event routine or each time there is a change in the state of a system variable. The user can also initiate the collection of statistics on user-defined entity attributes and global system variables. This is carried out during the passage of entities through a specialized modeling element known as

a TALLY node whose use will be detailed later.

There are two basic types of statistics collected during an NBDS run. The first class is derived from time independent samples and include the accumulated sums and squares on discrete observations of such variables as service time or time spent in a queue. If specified in a TALLY node, certain attributes or global variables can be maintained as grouped data and displayed later as frequency and cumulative distributions in a histogram. The second class of statistics is derived from time dependent samplings. They are collected over intervals of simulated time with the points between each interval marked by a change in the state of the variable under observation. Time-weighted statistics on variables such as the number of entities waiting in a queue or utilization of an activity or resource fall into this class.

In the general purpose NBDS program, the data collected during an simulation run is automatically analyzed during summary operations at the end of the run. In special purpose prototypes, the user has the freedom to determine which summary statistics are needed and codes his own summary operations. Where applicable, sample means and standard deviations are calculated for both types of statistics. The formulas used for

calculating those statistics are summarized in Table 4-1. The components of those equations represent several of the components of the statistical arrays maintained for each element.

Table 4-1 Formulas for Calculating Means and Standard Deviations

Statistics Based
Upon Observations

Sample Mean

$$\bar{x} = 1/n \sum_{i=1}^n x_i$$

Sample Standard Deviation

$$s = \sqrt{\frac{\sum_{i=1}^n x_i^2 - n \cdot (\bar{x})^2}{n - 1}}$$

Statistics For
Time-Persistent
Variables

$$\bar{x} = \frac{\int_0^T x(t) \cdot dt}{T}$$

$$s = \sqrt{\frac{\int_0^T x^2(t) \cdot dt}{T} - (\bar{x})^2}$$

n = number of samples

T = Total Time interval

5. Data Object Management Efficiently Handled With Nial

Section 3.5 reviewed some of the basic recordkeeping tasks involved when programming discrete event simulations with general purpose computer languages. Those nontrivial tasks deal mainly with maintaining ordered lists of records representing entities waiting for service in a queue or serving as a file for event notices. Now that the reader has just gained a basic understanding of the modeling concepts, data structures, and program control associated with NBDS, this section of the thesis will demonstrate how easily and efficiently Nial handles those same programming requirements. Nial's usefulness will also be exhibited through examples of record creation, record destruction, searching, sorting and selecting records from a list of records.

5.1 Entity Records--Creation, Destruction and List Formation

Every discrete event simulation program must be able to create records of entities representing new arrivals to the system and most programs will have a need to destroy unwanted records to free up space in the computer memory. Section 4.2 detailed the points of entity record creation which clearly indicate that most of the record creation during a simulation run occurs as a result of the bootstrapping step in the arrival event.

Here each new arrival to the system triggers the scheduling of its successor. In NBDS the CREATE operation serves this purpose by returning as its value a record (or array) of the next arrival to be placed on the event calendar. In CREATE, the variable is assigned this new record as follows:

```
NXTARVL: = A0 ARVT ARVT (SECOND GRULES)
          AVRT ' ' ' ' LINK (N_TRIBS RESHAPE 0)
```

In this example AVRT represents the arrival time of the entity which is the sum of the present simulated time and a random interarrival time period. The interarrival time period is chosen from a probability distribution specified in the operating rules for the source generation node as is the destination node of the new entity indicated by SECOND GRULES. The variable N_TRIBS specifies the total number of user-defined attributes each entity can be assigned and together with the RESHAPE 0 operation initializes each value to zero. The primitive LINK operation "links" the resultant attribute array with the other items in the record to create a single heterogeneous linear data structure known as the entity record. As an example, suppose the arrival of the next entity was determined to occur at time 30.0, its first destination is QUE1, and it can be assigned up to two user-defined attributes. A picture of NXTARVL would

appear as:

```
+-----+-----+-----+-----+
|A0|30.|30.|QUE1| |0|0|
+-----+-----+-----+-----+
```

Note, how the use of blank character strings in place of unassigned variables maintains the correct address spacing described in the previous section.

The record in the above example completely characterizes that particular entity upon its arrival to the system. It can be entered into any other list as a complete unit by simply nesting all its members into a single array and linking it to the other members of the list. For example, consider an arrival event in which the above record represents an entity arriving to QUE1. Assume that all the service activities are occupied when it arrives and must therefore wait for service in the queue. Also assume that prior to filing this record in the queue, the queue already contains two other entities waiting for service. Identified as QRECS, a picture of this file might appear as follows:

```
+-----+-----+-----+-----+
|A0|5.2|5.2|QUE1| 5.2|0|0| |A0|9.9|9.9|QUE1| 9.9|0|0|
+-----+-----+-----+-----+
```

Since the leftmost member of QRECS is usually served first, the ordering of records in QRECS could reflect a FIFO queueing discipline (note event time in each third

field). If the record representing the arrival is assigned to the variable ARVLREC, its entrance into this file simply requires the following expression:

QRECS: = QRECS LINK SOLITARY ARVLREC

A picture of this file would now appear as:

```

+-----+
|A0:5.2:5.2:QUE1: 5.2:0:0:|A0:9.9:9.9:QUE1: 9.9:0:0:|A0:30.30:QUE1: 30.0:0:|
+-----+

```

ARLVREC, with all its attributes, has now become the third member of the array QRECS. In actual practice, QRECS is placed in a collective file known as QFILE (a file containing all the service queue files). The resulting three level nested array is a good example of Nial's array-within-an-array concept. The example also demonstrates how easily lists can be created without having to provide a system of pointers to the individual data objects--the pointers are naturally embedded in the Nial language itself.

Once an entity waiting in a queue file has been scheduled for service, there must be a means to destroy or eliminate that record from the file. In NBDS this is carried out with the primitive "rest" operation which drops the first member from a list, leaving everything after the first item still intact. Since all queue files are usually served in order of lowest-address-first, the

first record in QRECS would be the next to receive service. Once scheduled, its record would be destroyed using the following expression:

```
QRECS: = REST QRECS
```

If QRECS started with the same three records shown above, the resultant picture of this last operation would appear as:

```
+-----+
|A0:9.9:9.9:QUE1: 9.9:0:0:|A0:30.30.QUE1: 30.0:0:|
+-----+
```

5.2 Sorting Record Lists

In the previous example, QRECS identified a file of records representing a queue with a FIFO queuing discipline. Therefore, to maintain a FIFO (or LIFO) ordering when records are added to the list, the new record needs only to be "linked" to one end of the list. However, at times the records in a queue file are ordered using a discipline which keys on certain attributes such as service time or time-in-system. In these cases the NBDS operation SORTUP or SORTDOWN is used depending upon which direction the file is to be ordered. As an example, consider the three-membered QRECS file used before but this time with a service time value stored in the first user-defined attribute position (address 8). A picture of the FIFO-ordered file would still appear like this:

```
+-----+
|A0:5.2:5.2:QUE1: 5.2:0:0:25:|A0:9.9:9.9:QUE1: 9.9:0:0:12:|A0:30.30.QUE1: 30.0:0:7.6:|
+-----+
```


number 2 (event time). The value then returned by SORTUP is the event calendar of records in order of earliest event time. By physically maintaining the event calendar in this order, each time the "select next event" instruction is encountered in the simulation control program, the program merely selects the first record from CALENDAR as the next to be processed. Again a rather complicated recordkeeping task is reduced to just a few lines of code using Nial.

5.3 Search and Selection of Records From a List

As mentioned earlier, an important programming consideration in discrete event simulation is the ability to search lists for records with a given value in certain fields. There may also be a need to remove that record from the list for use elsewhere in the program. A good example where this is used in NBDS is during the preemption of entities utilizing a particular resource. This element of NBDS allows entities arriving to a specialized preemption node to preempt the activity of other entities using a specified resource and acquire those resources for its own use. The preempted entities are then sent to a given resource queue until additional resources are made available again while the entity causing the preemption is scheduled for departure with the newly acquired resources.

An initial step in the preemption process is a search of the event calendar to find those entities currently utilizing the resource in question. For example, consider an event calendar containing the following records at the time of a preemption:

ID:6.1:75.2:RQU1:ACT:6.1:15.2:	40:101.101. RQU1: 101.	ID:25.6:132. RQU1:ACT:1:25.6:15.2:
--------------------------------	------------------------	------------------------------------

Assuming the number identifying the desired resource, RN, is 1, a visual search of each record's field #6 indicates the first and last entities scheduled for departure from regular activity #1 are each currently utilizing 5 units of the resource. A programmed search for these two records is achieved through use of the following expression:

POSTNS: = RN FINDALL EACH (6 PICK) CALENDAR

where POSTNS is assigned as its value the array (0 2) containing the positional addresses of the records in CALENDAR utilizing resource RN. Assuming both entities are preempted, their records can be culled from the event calendar and assigned to the array PRMPTRECS for selective processing elsewhere using the expression:

PRMPTRECS: = POSTNS CHOOSE CALENDAR

whose resultant picture would appear as:

```

+-----+
+D:6.1:95.2:RQU1:ACT1:6.1:1:5.12:ID:25.6:132.1:RQU1:ACT1:25.6:1:5.12:
+-----+

```

However, CALENDAR still contains the preempted records.

To remove them simply requires the expression:

```

CALENDAR: = ((TELL TALLY CALENDAR) EXCEPT
POSTNS) CHOOSE CALENDAR

```

which would leave the record scheduled for an arrival to RQU1 as the only record on the event calendar.

Obviously there are more steps involved in the preemption routine. However, the three lines of code presented in these examples demonstrate how simply a list search and item selection is carried out using Nial. From these and the previous programming examples, the reader should be able to appreciate how the power of Nial reduces many of the routine programming chores demanded by other computer languages.

6. Simulation Elements of NBDS

The process of developing prototype discrete simulation systems using NBDS requires the following steps:

1. model development and translation into a network diagram,
2. provision of a set of operating rules for each element of the model,
3. design and coding of operation(s) to provide a means for input of the system operating rules into the control program,
4. design and coding of operation(s) to generate a summary report, and
5. integration of simulation control program, element operations, input operation(s), and output operation(s) into a working simulation program.

The basic concepts of modeling and network diagrams involved in the first step were described earlier in Section 4. The last three steps will be dealt with later in the thesis when examples of the general and special purpose NBDS programs are presented. The purpose of this section is to provide the reader with enough information about the different modeling elements of NBDS to be able to complete the second task listed above. Presented will be the function of each element along with the content and format of its associated operating rules. Also, for each element Appendix A lists the names of the baseoperations built to support them in NBDS programs.

The operations are listed in a hierarchical fashion to indicate which operations are used within another. In addition, Appendix B contains an alphabetical listing of each NBDS operation and the beginning line number of its location in the general purpose NBDS script file. That script file is named NBDS.NDF and resides in the Lehigh University Computer Center tape library under Volume Serial Number JCW002. Together, Appendix A and B provide a quick reference to the operations and their source code required to support a particular simulation element in an NBDS prototype.

6.1 General Format for Operating Rules

Before each simulation element is presented, some general guidelines concerning the written format of the operating rules need to be introduced:

- Each component in a single set of operating rules becomes a member of a solitary array or list. Therefore, the members within a single set of operating rules must be delineated from each other by maintaining one or more spaces between them--not by separating them with commas or some other special character.

- All alphabetic data is entered in upper case.

- Non-numeric information is entered as Nial character strings and must therefore be enclosed by single quotes (eg. 'FIFO' or 'QUE1'). In some cases

numeric information must be entered as character strings as well. Those cases will be specifically indicated.

- Numeric data can be entered as integers or real numbers.

- The ordering of components in each set of rules is critical. Therefore, if a particular component does not need to be specified, it should be replaced by a blank character string (eg. ' ').

- The first member of every set of rules is at least a four-membered character string of which the first three characters identify its element type (eg. QUE or ACT). The last character(s) is an integer which uniquely identifies that set of rules among several of the same type (eg. QUE1, QUE2, etc.). Numbering begins with 1 and should (although not necessary in the general purpose package) be continuous.

6.2 Symbols for Nodes and Activities

Table 6-1 provides a listing of suggested symbols for depicting network models of NBDS systems. Any symbol can be employed by a modeler to represent a particular element as long as it distinguishes itself from others and fits nicely into the network diagram. Table 6-1 also lists the page number of this text in which the operating rules format for the given node or activity can be found.

Table 6-1 Symbols for NBDS Modeling Elements

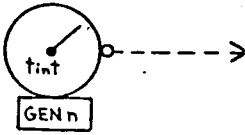

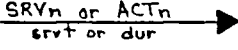
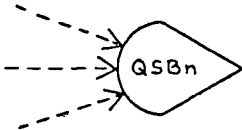
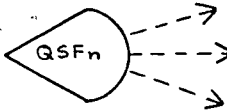
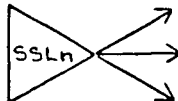

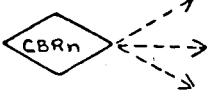
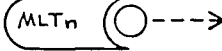
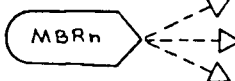
<u>Name</u>	<u>Symbol</u>	<u>Page Containing Operating Rules Format</u>
GENERATE		72
SERVICE QUEUE		84
SERVICE or REGULAR ACTIVITY		87, 88
Q-SELECT-BHND		81
Q-SELECT-FWD		77
SRVR-SELECT		83
CONTINUATION		107
CONDITIONAL BRANCH		109
MULTIPLY		107
MULTIPLE BRANCH		108

Table 6-1 (continued)

<u>Name</u>	<u>Symbol</u>	<u>Page Containing Operating Rules Format</u>
RESOURCE QUEUE		90
Resource Bank		92
RESOURCE FREE		93
RESOURCE ALTER		95
RESOURCE PREEMPT		98
CLOSE		113
OPEN		114
ASSIGNMENT		101
TALLY		104
TERMINATE		74

6.3 Entity Creation and Destruction

As mentioned earlier, every simulation program must provide a means for entity creation and destruction. These are provided for in NBDS through the use of GENERATE and TERMINATE nodes respectively.

- GENERATE Node

The operating rules format for the GENERATE node is shown below:

'GENn' 'dest' 'tint' tfg gmax

Collectively these rules are known as Genrules. In the leading label above (as in all other rules labels), n represents the unique integer number assigned to each set of rules within the given class. The character string dest is the destination node of each new entity which corresponds to that node's identifying label. The next character string, tint, refers to the time interval between generations. It can be substituted with a constant value or any NBDS probability distribution or program variable. The codes and associated parameters for the latter two options are displayed in Table 6-2. The next two rules, tfg and gmax, are the simulated time at which the first entity is generated and the maximum number of entities generated by that node respectively. Gmax becomes infinity if not specified and is the only member of those rules allowed to be left blank.

Table 6-2. NBDS Random and Program Variables

<u>Name</u>	<u>Associated Parameters</u>		<u>Definition</u>
UNFRM	MIN	MAX	A sample from a uniform distribution between the interval MIN and MAX
EXPON	MN		A sample from an exponential distribution with mean MN
NORML	MN	STD	A sample from a normal distribution with mean MN and standard deviation STD
ERLNG	MN	NS	A sample from an Erlang distribution which is the sum of NS exponential samples each with mean MN
ATRI ^B	N		N th user-defined attribute of an entity
GVAR	N		N th global variable
CLOCK			Current simulated time
TGEN			Generation time of an entity
RANNUM			A sample from a uniform distribution of random numbers in the interval (0,1)

An example of a set of GENERATE rules is shown below:

```
'GEN1' 'QUE1' 'UNFRM 5 10' 0 100
```

where GEN1 generates entities with a time interval between generations that is drawn from a sample of times uniformly distributed between 5 and 10. GEN1 is also shown to direct its newly created entities to the node QUE1 and begins generating them at time 0. Generation of entities ceases once a total of 100 have been created.

● TERMINATE Node

The TERMINATE node only requires two members in its operating rules:

```
'TRMn' tc
```

where tc is the termination count for the node TRMn. When the total number of entities terminated by that node is equal to tc, the simulation run is ended. If more than one TERMINATE node exists, the first one to reach its tc will end the simulation run. If tc is left blank, there is no limit to the number of entities destroyed by that node. The following is an example of a set of TERMINATE rules which would end a simulation run after processing 1000 entities at node TRM1:

```
'TRM1' 1000
```

6.4 Queue Selection and Service Selection Nodes

As described earlier, service queues are

locations in a network where arriving entities request the service of one or more discrete service entities represented by a single service activity. If all the servers are busy upon its arrival, the entity waits in the queue until one becomes available. These simple concepts are easily modeled. However, before a description of QUEUE nodes and service activities is given, several features and modeling elements related to them deserve attention first.

- Q-SELECT-FWD Node

Q-SELECT-FWD nodes provide one of several ways in which entities can be routed to different locations in a network. When an entity arrives to a Q-SELECT-FWD node, it is routed without delay to one of several parallel queues designated by the node. The choice of queues is made based upon a priority decision rule specified in the set of rules for the node. A summary of those decision rules is listed in Table 6-3.

As an example, consider the diagram of a Q-SELECT-FWD node and associated queue nodes shown below:

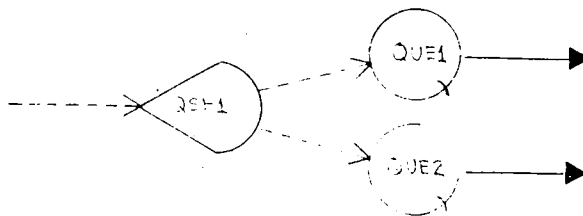


Table 6-3. Decision Rules Used by Q-SELECT-FWD
and Q-SELECT-BHND Nodes

<u>Code</u>	<u>Definition</u>
PRI	Select first available queue node from given order
CYC	Selection of queue nodes still designated in order but choose first available node after last one selected
RAN	Select queue node at random
LNQ	Select queue with largest number of waiting entities
SNQ	Select queue with smallest number of waiting entities
SRC	Select queue with smallest remaining capacity
LRC	Select queue with largest remaining capacity

If the queue selection rule designated by the Q-SELECT-FWD node was PRI and the given order was (1 2), an entity arriving to the node would always be routed to QUE1 providing room was available in the queue. Otherwise, the entity would default to QUE2. This particular node is useful in modeling the arrival of customers to a multi-queueing service area such as a supermarket checkout area or fast-food counter where the customer has a choice among several service lines.

The operating rules for this selection node are collectively known as Qsfrules. Their format is shown below:

'QSF n ' (n_1 n_2 ...) 'qsr' 'BLK or Balk To'

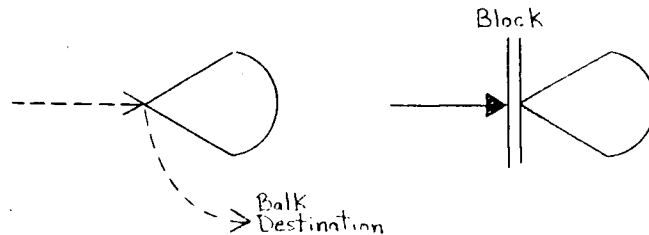
where the second component is an array of integer values identifying the numbers of the possible destination queue nodes (note: these values must be enclosed in parentheses to maintain their identity as a single component in the top level of the rules array). The character string designated by qsr refers to the code of one of the decision rules listed in Table 6-3. The last member of these rules describes a feature of queue nodes not yet discussed--balking and blocking.

When an entity arrives to a queue whose servers are fully occupied and there is no more room in the queue, two different actions are possible: 1) the entity can be

routed to another node in the network (called balking); or 2) if the entity was routed to the service queue by a service activity, it can wait outside the queue until room becomes available; however, until that entity can enter its destination queue, the service activity which served it is prevented from servicing any other entities. The second possibility is called blocking and can only affect service activities. In this case the server tied up by the blocked entity is not considered to be utilized but is not free to resume service either. A situation like this can be represented by a forklift transporting commodities to a loading zone. If on arrival to the loading zone there is no more room available to unload its goods, the forklift and goods must sit idle until the next set of goods is removed from the queue. In the case of the first possibility, an entity can balk out of the system by being routed to a TERMINATE node or it can assume a new destination anywhere else in the network (note: balking is not permitted to queues which allow blocking).

In the Qsfrules example listed earlier, the character string 'BLK' entered in the last position would allow entities arriving to the Q-SELECT-FWD node to be blocked if they were routed there by a service activity and all the possible destination queues were full. If

balking was desired instead, the code for the destination node would be entered in this position (note: statistics are automatically kept on balks from individual queues but not in the case of balks from a Q-SELECT-FWD node). If neither balking or blocking was desired, the last position in the single set of Qsfrules would be left blank. Below are the symbols for balking and blocking respectively, used here with Q-SELECT-FWD nodes:



An example of a single set of Qsfrules is shown below:

```
'QSF1' (1 2) 'PRI' 'TRM2'
```

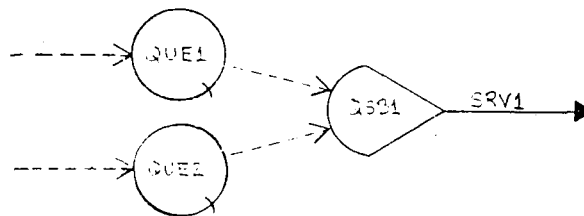
These rules could describe the two parallel queues example presented earlier but indicate that entities would balk out of the system (destroyed by TERMINATE node 2) if both destination queues were at capacity.

- Q-SELECT-BHND Node

A second node used in conjunction with a set of parallel queue nodes is the Q-SELECT-BHND node. It is associated with single or multiple service activities and

is responsible for selecting the next queue to be serviced when a service entity is freed from a previous activity. It provides a "look behind" capability in contrast to the "look forward" capability of a Q-SELECT-FWD node and is referenced only when a service entity it precedes completes a service activity. The Q-SELECT-BHND node does not interfere with the request-for-service of an entity arriving to one of the parallel queues it polices. If a service entity is available, that entity is immediately served; otherwise, the entity waits in the queue and is then selected for service based upon the decision of the Q-SELECT-BHND node. Like the Q-SELECT-FWD node, this node selects the next queue to be serviced based upon a priority decision rule. The decision rules and their codes for the Q-SELECT-BHND node are the same as those used by Q-SELECT-FWD nodes which are listed in Table 6-3.

Below is a partial network diagram of a Q-SELECT-BHND node and its associated queues:



Assuming SRV1 just finished a service activity, QSB1

would check to see if any entities were waiting in QUE1 or QUE2. If both queues were in use, the choice to serve a particular queue would be made based upon QSBL's decision rule. For instance, if the rule was specified as LNQ, the queue containing the largest number of entities would be served next. If both queues contained an equal number of entities, the first queue in the list of queues would be served. It's easy to see from this example that the Q-SELECT-BHND node models the perspective of a service activity. A situation where this would be a useful modeling element is in a manufacturing process where separate queues develop along a line that are serviced by one or several activities.

The operating rules for the Q-SELECT-BHND node are collectively designated as Qsbrules. Their individual format is shown below:

'QSBn' (n₁ n₂ ...) 'qsr' 'srvid'

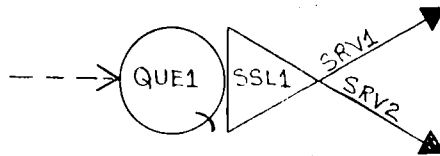
where the second and third members are identical to those in Qsfrules. The last element refers to the identification code of the service activity or service selection node following the Q-SELECT-BHND node. Service selection nodes provide a choice among several different service activities and will be discussed in the next segment. As an example of a set of Qsbrules, consider the following:

'QSBl' (1 2) 'LNQ' 'SRV1'

These rules would describe the example given earlier in which SRV1 selected from either QUE1 or QUE2 depending upon which had the greater number of waiting entities.

● SRVR-SELECT Node

The last preliminary node that needs to be described is the SRVR-SELECT node. This node is used when there is a need for an entity arriving to a queue to select a particular service activity from among several serving that same queue. That is, every service activity is allowed multiple service entities but this node allows for multiple service activities as well. A SRVR-SLCT node is situated between its associated queue and the service activities it polices as shown below:



Like the previous two selection nodes, the SRVR-SELECT node is governed by a set of priority decision rules listed in Table 6-4. If the rule for the above example was specified as RAN, an entity arriving to QUE1 would select either SRV1 or SRV2 at random if both service activities were idle.

Table 6-4. Decision Rules Used by SRVR-SELECT Nodes

<u>Code</u>	<u>Definition</u>
PRI	Select first available service activity from a given order
CYC	Select service activities from a given order but select first available one after last one selected
SBT	Select service activity having smallest busy time
LBT	Select service activity having largest busy time
RAN	Select service activity at random

The collective name for the operating rules of this node is Sllrules. Below is the format for a single set:

'SSLn' (n₁ n₂ ...) 'ssr'

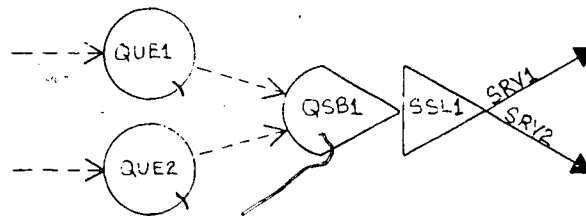
where (n₁ n₂ ...) represents the solitary array of service activity numbers provided for selection and ssr is the decision rule code. The following is an example of these rules based on the diagram presented earlier:

'SSL1' (1 2) 'RAN'

Again, only integer numbers identifying each service activity is used in the second component while the entire list is enclosed in parentheses to maintain its singularity among the other members.

SRVR-SELECT nodes are useful when modeling a system where a particular source of entities require different processing times or a particular service activity is given a higher priority. For instance, customers arriving

to a fast-food counter with multiple servers might choose the counterperson who takes the least time to prepare their order based on previous experience or management might encourage such a situation themselves by the proper line arrangement. As mentioned earlier, SRVR-SELECT nodes can also be used with Q-SELECT-BHND nodes. The partial network diagram shown below demonstrates such a combination:



6.5 SERVICE QUEUE Nodes

By now the reader should be familiar with the function of SERVICE QUEUE nodes and their network symbol. Therefore, this next section will be devoted entirely to a discussion of its operating rules format.

The operating rules for a QUEUE node are referred to collectively as Qrules. The format for a single set is shown below:

'QUEn' 'disc' qn qmax 'BLK or Balk To' 'srvid' 'qsid'

The character string disc refers to the queueing discipline for that queue. Table 6-5 lists the codes and

Table 6-5. Queueing Disciplines

<u>Code</u>	<u>Definition</u>
FIFO	Entities served in order of arrival
LIFO	Entities served in reverse order of arrival
LVFn	Entities served in order based on low-value-first of n th user defined attribute
HVFn	Entities served in order based on high-value-first of n th user defined attribute
RAND	Entities served at random

descriptions of those available in NBDS. The elements qn and qmax refer to the initial number of entities in the queue at the start of the simulation and the maximum allowable number of entities in the queue at any one time respectively. If qn is greater than 0, service begins immediately at the start of the simulation for as many entities as there are available servers (note: if beginning simulation with entities already in the queue, LVF or HVF queueing discipline cannot be used). If there is no limit to the number of entities a QUEUE node can hold, then qmax should contain a blank character string. Blocking and balking can be specified if qmax is zero or greater. If blocking is desired, the fifth position should contain the character string 'BLK'. If balking is desired, this element should be replaced with the code name of the destination balked to. The next position represented by srvid holds the identification of the

associated service activity or SRVR-SELECT node. Finally, the element qsid refers to the code name of a Q-SELECT-FWD or Q-SELECT-BHND node associated with the queue. As implied by the single position for two different options, an individual QUEUE node can only be associated with one type of queue selection node at a time.

As an example of a set of Qrules, consider the following:

```
'QUE1' 'FIFO' 0 ' ' ' ' 'SRV1' ' '
```

This represents the simplest QUEUE node possible. QUE1 maintains a FIFO queueing discipline, starts with no entities in the queue, and is serviced by the activity SRV1. Note the blank character strings which indicate no limits on the queue length, no blocking or balking, and no association with queue selection nodes respectively. Again, their inclusion is important as the position of each rule in a set is critical.

6.6 Activities

Since the role of activities in a network has already been established, this next section will simply highlight the differences between the two types and describe their respective operating rules content and format.

● Service Activities

Service activities limit the number of concurrent entities travelling over them and are used only in conjunction with service queues. Their operating rules are known collectively as Svrules and each set has the following structure:

```
'SRVn' 'dest' 'srvt' nsrvs 'ssl'
```

As in the rules for a GENERATE node, dest represents the code for the destination node to which an entity is delivered by the activity. The character string srvt is the designated service time for that activity and is the time period by which an entity's passage through the activity is delayed. Like the tint rule of Genrules, srvt is substituted with a constant value or any code and associated parameters for the random variates and program variables listed in Table 6-2. The descriptor nsrvs is the number of parallel servers represented by the service activity. Lastly, the character string ssl represents the name of a SRVR-SELECT node with which the service activity is associated. If the service activity is not associated with one, this last position is left blank.

As an example of a set of Svrules consider the following:

```
'SRV1' 'QUE3' 'ATTRIB(1)' 1 'SSL1'
```

Here the last member indicates SRV1's association with

SRVR-SELECT node SSL1 (as in the previous example where SRV1 was named as one of two possible service activities an entity could select from when arriving to the node SSL1). In this case, the destination of an entity utilizing SRV1 is the service queue QUE3. In addition, the service time for that entity is determined by the value of its first user-defined attribute, ATTRIB(1). Finally, the value at address 3 indicates this is a single server activity.

- Regular Activities

Regular activities are used to transport entities away from nodes other than QUEUE nodes and have no restriction on the number of entities utilizing them at one time. A collection of operating rules describing regular activities is referred to as Actrules. A single set of Actrules has the following format:

```
'ACTn' 'dest' 'dur' 'N/S'
```

where dest again represents the code of the destination node to which the activity delivers its entities and dur represents the duration of time an entity's progress is delayed through the activity. Dur is similar to srvt of Srvrules except a zero time duration can be specified for the activity by simply leaving its character string blank. The last member in this set of rules gives a modeler the choice of whether or not to collect

descriptive statistics on a given activity. In some cases regular activities are required only to provide a link between nodes with no need to report their utilization statistics, etc. In these situations, the modeler has the option of supplying the character string 'N/S' in the last position of the operating rules statement to prevent the wasted collection of statistics.

Below is an example of a set of Actrules:

```
'ACT1' 'TRM1' ' ' 'N/S'
```

where entities passing through ACT1 are directed to TERMINATE node 1 without delay. Also, the last member in this set of rules indicates that no statistics describing the use of this activity are to be collected during the simulation run.

6.7 Resource Queues and Related Simulation Elements

As described in Section 4.1, resource queues are similar to service queues except entities arriving to the latter type must acquire "resources" as opposed to discrete service entities to continue their passage through the queue. Resources are delegated to a queue from an external source which can be shared by multiple resource queues. Arriving entities can request variable amounts of a given resource and, once acquired, depart from the queue over a regular activity branch. This next section details the operating rules required by resource queues and also introduces several of the modeling

elements used with them.

- RESOURCE QUEUE Node

Below is the format of a set of operating rules for a RESOURCE QUEUE node which are referred to collectively as Rqrules:

'RQUn' 'disc' qn qmax 'Balk To' rn 'nrqst' 'actid'

Except for the leading label, the first four components are identical in name and function to those in a set of Qrules. The Balk To variable at address 4 is also the same as that for Qrules but, as this rule implies, resource queues only allow balking--not blocking. The last three members of this set of rules are unique to RESOURCE QUEUE nodes. Rn is an integer number identifying the source of resources for the queue (see next section). The character string nrqst represents the number of resources an entity arriving to the queue requests and is one of three rules in this family of elements where numeric information must be entered as a character string. The value might be a constant but, like the time delays for activities, the value of nrqst may also be obtained by specifying one of the random variates or program variables listed in Table 6-2. Finally, actid refers to the identifying code for the regular activity emanating from the RESOURCE QUEUE node.

An example of an individual set of Rqrules is given

below:

'RQUL' LVF2' 0 ' ' ' ' 1 'TRIB(2)' 'ACT1'

Going from left to right, this set of rules specifies that entities waiting for resources in RQUL queue up in order of low-value-first based on their second user-defined attribute. The third component indicates the simulation begins with no entities in the queue. The next two blank character strings mean there is no limit to the capacity of the queue and, hence, no balking from the queue respectively. The value at address 5 indicates that resources acquired at RQUL are held by resource bank #1. Furthermore, the next position specifies that entities arriving to the queue request resource amounts equal to the value contained in its second user-defined attribute (therefore, by virtue of the queueing discipline, waiting entities requesting the least amount of resources are served first). Finally, the last component indicates that entities travel from RQUL over regular activity ACT1.

● Resource Banks

Resource banks are used to hold specified amounts of resource for allocation to designated resource queues. A bank of resources varies dynamically throughout a simulation run but can never exceed its capacity or drop to a negative amount. They lie outside the actual network model but must be specified in the operating

rules when RESOURCE QUEUE nodes are used.

Together the operating rules for Resource Banks are known as Rscrules. A single set of rules assumes the following format:

'RSCn' capac (n_1 n_2 ...)

where RSCn identifies the particular bank of resources and capac specifies the number or amount of resources available for allocation at the start of the simulation. The last member of these rules is a solitary array which lists the integer numbers of those RESOURCE QUEUE nodes associated with the resource bank. The order in which those queues are listed is important during the reallocation of resources which will be discussed in the next two segments.

As an example of a set of Rscrules, consider the following:

'RSC1' 10 (1 2 3)

Here resource bank RSC1 starts the simulation run with 10 units of resource which are allocated to entities arriving at resource queues RQU1, RQU2, and RQU3 (provided the requested amount is still available). However, if additional resources become available or previously acquired resources are relinquished during the simulation run, reallocation of those resources to waiting entities begins immediately by polling each of

the three resource queues in the prescribed order.

● RESOURCE FREE Nodes

Resources previously acquired by an entity are relinquished by routing the entity through a RESOURCE FREE node. RESOURCE FREE nodes specify the amount of resources given up by an arriving entity as well as the originating Resource Bank to which they are returned for reallocation. Since all entities arriving to a RESOURCE FREE node will trigger the release of additional resources, care should be taken by the modeler to preserve the balance of resources in the system by not releasing more than was originally available. Where variable amounts of resources are acquired by entities arriving to a resource queue, it is a good practice to record that amount in the entity record as a user-defined attribute. That attribute can then be referenced upon an entity's arrival to a RESOURCE FREE node to determine the correct amount of resources to relinquish. In the event an excess balance of resources is released at a free node, the amount of available resources will not increase beyond the given Resource Bank's capacity.

The operating rules for RESOURCE FREE nodes are referred to collectively as Freerules. An individual set has the following format:

'FREN' rn 'nrel' 'actid'

where rn specifies the number of the Resource Bank to which nrel resources are returned by entities arriving to the node. Like the nrqst component of Rqrules, nrel is a character string which may contain a constant value or expression derived from the random variates and program variables listed in Table 6-2.

An example of a set of Freerules is the following:

```
'FREL' 1 'ATRI(2)' 'ACT2'
```

where entities arriving to FREL release acquired resources in an amount equal to the value stored in its second user-defined attribute. The relinquished resources are given back to Resource Bank #1 where they are immediately reallocated to the resource queues prescribed by RSC1. The regular activity ACT2 then routes arrivals away from FREL.

- RESOURCE ALTER Nodes

At certain points in a network model, there may be a need to adjust the capacity of a particular Resource Bank. This is accomplished in NBDS by routing an entity through a special element known as a RESOURCE ALTER node which is a particularly useful element for simulating employee work breaks or scheduled machine maintenance.

The operating rules for RESOURCE ALTER nodes are known collectively as Altrules. An individual set has the

following structure:

'ALTn' rn 'nalt' 'actid'

where rn is the number of the Resource Bank whose resource capacity is being altered and the character string nalt represents the amount by which the capacity is altered. Nalt may be negative or positive and may also be defined by one of the variables listed in Table 6-2. As in the other related rules, actid represents the name of the regular activity which routes arrivals away from ALTn.

An important point should be made concerning the use of a RESOURCE ALTER node. If the arrival of an entity to an alter node would reduce the capacity of its designated Resource Bank below the number of resources currently available, no effort is made to recover the difference from entities currently in possession of them. Instead, the entities flow through the network as usual and when they finally encounter a RESOURCE FREE node, the resources released at the node simply aren't reallocated by the designated Resource Bank. Also related to this is the effect of repeated arrivals to an alter node that decrements the capacity of a given Resource Bank. Once the capacity has been reduced to zero, any additional arrivals to the alter node will not create a negative deficit.

Consider the following set of Altrules:

'ALT1' 1 '~10' 'ACT3'

Here entities arriving to ALT1 will initially cause a reduction in the capacity of Resource Bank #1 by 10 units and then be routed away from the node by the regular activity ACT3. Note the use of the tilde symbol in the value ~ 10 to indicate a negative number. This is a Nial convention and is in contrast to the normal dash (-) reserved as the operation symbol for subtraction.

- RESOURCE PREEMPT Nodes

RESOURCE PREEMPT nodes are useful elements for modeling situations where the utilization of a resource by an entity is suddenly interrupted by another (eg. machine breakdowns or an interruption in the transmission of a message over a shared communications line). Here entities arriving to a preempt node will initially request the use of a given amount and type of resource just as entities do upon arriving to a resource queue. If a sufficient amount of resources is available, they are allocated to the entity in the normal fashion and the entity proceeds on through the network. However, if the available resources cannot satisfy the entity's request, then the entity will attempt to preempt that same resource from entities already utilizing them until its requirements are satisfied. Resources may only be

preempted from entities currently engaged in a regular activity (ie. not from entities waiting in a queue or engaged in a service activity). A preemption attempt on an entity engaged in a regular activity will only be successful if the value of a given user-defined attribute gives the preemption entity a higher priority. If an entity arriving to a RESOURCE PREEMPT node fails to acquire its requested number of resources, it balks to a destination node specified by the modeler.

If more than one entity is preempted in a single attempt, preemption begins with those entities scheduled for the latest departure event. If only a portion of the resources owned by the last entity preempted were required to satisfy the preemption, the remainder is made available to other entities waiting for that resource. Preempted entities are sent to a designated resource queue where they are established as the first entities waiting for the resource (in order of earliest departure time first). Their remaining processing time is saved in the third program-defined attribute of their entity record which is later used as their activity duration when reassigned resources. Preempted entities which resume activity take up at the same place in the network from which they were preempted. However, because preempted entities may be sent back to a queue different

from their original source, they may resume activity with a resource different from the one they possessed when preempted. Also, if an entity targeted for preemption is in possession of more than one resource type, only the last resource will be given up; all other resources remain in that entity's possession when sent back to a resource queue.

The operating rules for a RESOURCE PREEMPTION node are known collectively as Pmtrules. Below is the rules format for an individual set:

'PMTn' rn 'nrqst' 'LVn or HVn' rq 'Balk To' 'actid'
where nrqst represents the amount of resources from Resource Bank # rn initially requested by an entity arriving to PMTn. The character strings LVn and HVn of the fourth component specify the type of priority and attribute number an entity arriving to this nodes assumes when attempting a preemption of another entity. Rq is an integer number identifying the resource queue to which a preempted entity is sent while Balk To represents the code name of the node to which an entity is sent when unsuccessful in a preemption attempt. In the event of a successful preemption (or normal acquisition of available resources), the entity arriving at PMTn is routed away from the node over regular activity actid.

To illustrate the use of a set of Pmtrules, consider

the following:

'PMT1' 1 '5' 'HV3' 1 'TRM2' 'ACT4'

Here an entity arriving to PMT1 requests 5 units of resource from Resource Bank #1. If they are not available, or only partially available, then an attempt is made to preempt entities in possession of that resource until enough resources are acquired to satisfy the arrival's request. Entities determined as candidates for preemption will only be preempted if the value of their third user-defined attribute is less than that of the entity arriving to PMT1 (equal values using either priority scheme will not cause a preemption). If a successful preemption occurs, the preempted entity is sent to RESOURCE QUEUE node #1 where it waits at the head of the line for available resources; the entity causing the preemption then continues its journey through the network over regular activity ACT1. If unsuccessful in acquiring the requested units of resource, the entity arriving to PMT1 immediately balks to TRM2 where it is terminated.

6.8 ASSIGNMENT Nodes

Up to now a great deal of attention has been given to user-defined attributes of an entity. As just presented in the discussion of the RESOURCE PREEMPTION node, user-defined attributes are required to establish a

priority system among entities involved in a preemption attempt. They are also useful to hold pre-determined service times or record the units of resource acquired at a given RESOURCE QUEUE node. To carry out these assignments in an NBDS prototype, entities are routed through ASSIGNMENT nodes. These elements assign values to members of an entity record reserved to store user-defined attributes. The array of attributes is collectively referred to as ATRIB. Arrivals to an ASSIGNMENT node can also change the value of a globally defined program variable contained in the array GVAR.

The statement used to assign values to any of these variables must begin with the name of the array and the address value of its position in the array (addressing begins with 1). Following the assignment symbol, the right-hand side of the expression can contain a single constant or any of the NBDS variables listed in Table 6-2. The expression can also contain any combination of constants and variables with any number of arithmetic operations as long as it conforms to the constructs of Nial and employs the correct arithmetic symbols. As examples of valid NBDS assignments at an ASSIGNMENT node, consider the following:

```
ATRIB(2): = EXPON(10) + ATRIB(1)
```

```
GVAR(1): = GVAR(1) * 2
```

In the first example, an entity arriving to the node would have the value of its second user-defined attribute replaced with the sum of a sample drawn from an exponential distribution (with mean 10) and its first user-defined attribute. In the second example, the global variable GVAR(1) is assigned as its value the product of its present value and the constant 2.

The operating rules used to define ASSIGNMENT nodes are known collectively as Assnrules and have the following individual format:

```
'ASSn'  'dest'  'exp1'  'exp2'  ...  'expK'
```

where an individual node can have K separate expressions. As shown, ASSIGNMENT nodes are not associated with an activity. Instead, an entity arriving to the node is immediately passed to its destination node as indicated by dest. Also note that each expression must be entered as a character string.

As an example of an individual set of Assnrules consider the following:

```
'ASS1'  'QUE3'  'ATRIB(2): = EXPON(10) +  
ATRIB(1)'  'GVAR(1): = GVAR(1) * 2'
```

Here the expressions used were given in the previous examples and would result in the assignments described there. Also, an entity arriving to ASS1 is shown to be immediately routed to the node QUE3.

An additional element associated with ASSIGNMENT nodes also needs to be introduced here which concerns the initialization of global variables. Like the attributes for an entity, the values of all GVAR variables are automatically initialized to zero at the start of a simulation run. However, if the modeler wishes to initialize a particular variable with a non-zero starting value, this can be accomplished by including one or more INIT statements with each set of Assnrules during the input of the operating rules. One statement is used for each global variable to be initialized and has the following format:

```
'INIT' 'exp'
```

where the generalized INIT label is used with each initialization and exp represents any GVAR assignment expression discussed earlier. For example, the following statement would initialize the previously used global variable to the value 5 at the start of the simulation run:

```
'INIT' 'GVAR(1): = 5'
```

The collective name of a group of INIT statements is Inits.

6.9 Collection of Supplementary Statistics

While most of the simulation elements presented so far were designed to collect their own set

of descriptive statistics, NBDS also has the capability to collect supplementary statistics on user-defined attributes, global variables, and other system variables. Time independent statistics can be collected on most system variables by routing entities through a TALLY node while time dependent statistics can be maintained on any global variable through use of a TIMD statement.

- TALLY Node

Each time an entity arrives to a TALLY node, one of the following types of variables can be automatically collected as an individual observation:

1. TSYS - the length of time an entity has spent in the system to that point.
2. INT(n) - the difference between the arrival time (CLOCK) of an entity to the TALLY node and a mark time stored in user-defined attribute n.
3. BTWN - the time between arrivals to the TALLY node, using the first arrival as a reference point.
4. ATRIB(n) - the value of user-defined attribute n.
5. GVAR(n) - the current value of global variable n.

The statistical array Tallystats stores most of the data collected at TALLY nodes. Those data include minimum and maximum values, total number of observations, and the information necessary to estimate means and standard deviations at the end of the simulation run. TALLY nodes

can also collect the data required to generate a histogram at the end of the simulation run. The histogram is divided into 17 cells and depicts the frequency distribution of values observed on a designated variable over the length of a simulation run. Those data are maintained as separate arrays within a single array known as the Freqfile.

The operating rules for a collection of TALLY nodes are referred to as Tallyrules. The format for an individual set is shown below:

```
'TLYn'  'dest'  'title'  'typ' ('HIST' ll ul)
```

As with ASSIGNMENT nodes, entities passing through a TALLY node are immediately routed to the next destination represented by dest. The character string title is any name the modeler chooses to identify the type of variable being observed while typ refers to one of the code names for the five different variable types listed above. The last member in these rules is a solitary array specifying the need for a histogram. Along with the character string 'HIST', the modeler must provide an estimated range of observed values for the variable by specifying its lower limit (ll) and upper limit (ul) respectively. If no histogram is desired, the three-membered array is replaced with a blank character string.

Below is an example of an individual set of

Tallyrules:

```
'TLY1' 'TRM1' 'Time in System' 'TSYS' ('HIST' 10 500)
```

These rules indicate TLY1 is used to collect the duration of time an entity arriving to that node has spent in the system to that point. Since the entities are subsequently terminated, that time period represents their total lifetime in the system. Furthermore, a histogram is called for depicting the frequency distribution of those observations estimated to lie between 10 and 500 time units.

- TIMD Statement

The TIMD statement is another NBDS element that is not directly represented by a network symbol but instead is used to designate a system global variable for the collection of time-persistent statistics. Data describing the value of a global variable over the duration of the simulation run are maintained in the statistical array Glbstats. That information can be used at the end of the simulation to generate mean values and standard deviations as well as minimum and maximum observed values.

The TIMD statement is similar to the INIT statement in that it may only be used to name a single global variable. A group of TIMD statements are referred to collectively as Timrules. The structure of an individual

statement is shown below:

```
'TIMD' 'title' n
```

where each statement is preceded with the generalized label TIMD. As before, title refers to a user-supplied character string which uniquely names the variable of interest and n represents the integer number of its address in the GVAR array. For example, in the TIMD statement shown here:

```
'TIMD' 'Number in System' 2
```

GVAR(2) is used to monitor the number of entities in the system at any one time and is designated to be maintained as a time dependent variable.

6.10 CONTINUATION Nodes

In many instances during the design of a network model there is a need to separate regular activities into two or more activities with distinct time delays. There may also be a need to immediately follow a service activity with a regular activity; or where a node (like the TALLY node) is not associated with an activity, there may be a need to delay an entity's departure from that node to its next destination. In all three cases, a CONTINUATION node can be used to solve the problem.

The operating rules for a CONTINUATION node are known collectively as Contrules. The format for an individual set is shown below:

'CONn' 'ACTn'

This two-membered set of rules just contains its identifying code name and the code name of the regular activity which delivers entities from the node. As shown, it simply models a sequential arrival and departure event with a time delay (specified by ACTn) in between.

6.11 MULTIPLY Node

When a network model calls for the simultaneous generation of multiple entities, either from a GENERATE node or some other location, the MULTIPLY node will satisfy that requirement. When an entity arrives to a MULTIPLY node, it is replicated any number of times, afterwhich the parent and its clones are immediately routed to a single destination node.

A collection of operating rules for MULTIPLY nodes are referred to as Multrules. An individual set has the following format:

'MLTn' nm 'dest'

where nm identical entities are routed from MLTn to dest for each arrival to the node.

6.12 MULTIPLE BRANCH Node

This NBDS modeling element is identical to the MULTIPLY node just presented except the multiple entities created at the node are individually routed to

two or more different destination nodes in the network.

The collective name for the operating rules of MULTIPLE BRANCH nodes is Mbrnrules. Individually, a set of Mbrnrules contains the following format:

```
'MBRn' ('dest'_1 'dest'_2 ... 'dest'_K )
```

where K identical entities are routed to multiple destination nodes from MBRn for each arrival to the node (that includes the parent entity as well). Note the use of a single array to hold all the destination nodes. The maintenance of this array as a single member in the top level of the rules is critical.

An example of a single set of Mbrnrules is given below:

```
'MBR1' ('QUE1' 'QUE2' 'QUE3')
```

Here an entity arriving to MBR1 is replicated three times and evenly distributed to three different service queues.

6.13 CONDITIONAL BRANCH Node

CONDITIONAL BRANCH nodes are useful NBDS modeling elements in that they provide important decision points within a network. An entity arriving to a CONDITIONAL BRANCH node is confronted with a sequence of conditional statements, each of which is associated with a different destination node. The entity begins testing each condition and is routed to the destination of the first one satisfied.

Below is the format for a set of CONDITIONAL BRANCH node operating rules collectively known as Cbrnrules:

```
'CBRn' ('cond'_1 'dest'_1 ) ('cond'_2 'dest'_2 )  
... ('cond'_K 'dest'_K)
```

where CBRn contains K sets of conditions and associated destination nodes. Here each condition (cond) and destination node (dest) are defined together as a single array within the array of operating rules and represent a single branch from the node CBRn. When an entity arrives to a CONDITIONAL BRANCH node, it begins testing each condition in the order given in the set of Cbrnrules. If a given condition is satisfied, the entity automatically departs the node to the destination node associated with the conditional statement and no further testing is carried out. If a given condition is not satisfied, then the entity tests the next one in line. If none of the conditions are satisfied, the entity is routed to the last destination node specified as a fail-safe measure. Therefore, the modeler could actually substitute a blank character string for the last condition to be tested but, for clarity's sake, should be spelled out explicitly.

The conditional statements can contain any constants, random variates, or program variables listed in Table 6-2 but must be used with the Nial relational and Boolean operations listed in Table 6-6. Note that the

Table 6-6. Standard Nial Relational and Boolean Operations

<u>Operation</u>	<u>Definition</u>
>	greater than
<	less than
=	equal to
>=	greater than or equal to
<=	less than or equal to
≠	not equal to
and A	logical and of items of A
Not A	reverse the logical value of A
or A	logical or of items of A

NBDS variable RANNUM can be used to specify a probability in a conditional statement such as the following:

```
RANNUM > 0.20
```

Here the condition is satisfied if the uniformly distributed random number generated by RANNUM is greater than 0.20. If used in a set of Cbrnrules, an arriving entity would face an 80% chance of being routed to the associated destination node.

As an example of a set of Cbrurules, consider the following:

```
'CBR1' ('ATRIB(1) = 1' 'CON1') ('ATRIB(1) = 2' 'CON2')
```

Here the conditional statements are testing for a certain attribute value of the entity arriving to CBR1. If the entity's first user-defined attribute is equal to 1, it is routed to the CONTINUATION node CON1. Otherwise the attribute value is considered to be equal to 2 and the entity departs to CON2.

6.14 CLOSE and OPEN Nodes

In many queueing situations there is often a need to temporarily suspend service to a particular queue or group of queues. Such might be the case when a bank teller takes a lunch break or a machine on an assembly line is shutdown for maintenance. Another example would be a traffic light at an intersection where the flow of traffic is halted in one or more directions for a given

period of time. In the case of the bank teller breaking for lunch, the customers lined up for service in his queue would most likely be directed to another teller still in service. However, in the last example, the drivers lined up at a traffic light would be forced to wait in line until they could pass through a green light.

All of the situations described above can be modeled with CLOSE and OPEN nodes. In addition, these modeling elements can be applied to both service queues and resource queues.

● CLOSE Nodes

When an entity arrives to a CLOSE node, a designated queue or group of queues is closed for service. In the case of a service queue, any entities currently in a service activity are permitted to complete that service. Likewise, entities from a resource queue currently in possession of a resource are allowed to keep it until scheduled for release. However, when the service activity is over, its servers are idled or when the resource is relinquished, the designated resource queue is not polled for reallocation. At the time of closure, an additional action may take place. If the designated queue permits balking (as indicated by its fifth operating rule), the modeler has the option of sending all the entities currently in that queue to the

destination node specified by the balking rule. Any additional arrivals to the queue automatically balk to the given destination node while that queue remains closed. If balking is not specified, then any additional arrivals to the queue simply enter the queue and wait until service resumes.

The operating rules for a CLOSE node are referred to collectively as Clsrules. An individual set has the following format:

```
'CLSn' 'qtyp' (n1 n2 ...) 'BALK' 'actid'
```

where the character string qtyp specifies the type of queue(s) to be closed. Here the code QUE specifies a service queue while RQU specifies a resource queue. The third member of these rules is an array of integer numbers identifying the queue or queues of that type to be closed. (Note: CLOSE nodes cannot specify service queues associated with Q-SELECT-FWD nodes). The next item gives the modeler the option to balk all current and future entities to the balking destination node specified by that queue. If this rule is left as a blank character string, balking will only occur if: 1) the operating rules for the queue specify it; and 2) the queue's capacity is reached while the queue remains closed. The last operating rule for a CLOSE node specifies which regular activity routes the arrival away from the node.

An example of a set of Clsrules is shown below:

'CLS1' 'RQU' (1 2) ' ' 'ACT5'

Here an entity arriving to CLS1 forces the closing of resource queues 1 and 2 and is routed away from the node on ACT5. The blank character string at address 3 allows normal balking to occur from those nodes if indicated in their respective Rqrules.

● OPEN Node

OPEN nodes are used to resume service on a previously closed queue. When an entity arrives to an OPEN node associated with service queues, the specified queues are served immediately if a service entity is available. Likewise, an entity arriving to an OPEN node associated with resource queues results in the immediate polling of those queues if available resources exist. Also, if automatic balking was specified, that restriction is lifted as well.

A collection of operating rules for OPEN nodes are referred to as Opnrules. An individual set is structured as follows:

'OPNn' 'qtyp' (n₁ n₂ ...) 'actid'

where the rules are identical to those of Clsrules by the same name. As an example of an individual set, consider the following:

'OPN1' 'RQU' (1 2) 'ACT6'

Here an 'entity' arriving to OPN1 would allow the same queues closed by the earlier example to resume normal activity.

6.15 SEED Statement

The SEED statement is another statement which is read into the program along with the operating rules at the start of a simulation run. It allows the user a choice of ten different seed values for the random number generating operation RANNUM (actually two different seed values are picked with this statement, one for each of the two alternating generators within RANNUM). The format of this statement is:

'SEED' n

when n is any integer from 1 to 10 inclusive. If no SEED statement is included with a group of operating rules, the seed value defaults to the first one.

6.16 END Statement

The last NBDS element to be presented is appropriately named the END statement. It is read into the NBDS program like all other statements and gives a user the option to end a simulation run at a particular simulated time. The statement has the simple format:

'END' time

where time is the CLOCK time at which the simulation run

is terminated. Together with the termination count specification of TERMINATE nodes, a modeler has the option of terminating the simulation run based on number of entities processed or simulated time elapsed.

7. Model Building With NBDS Elements

Now that the reader has been introduced to the modeling elements of NBDS, this section will present a few examples of how those elements can be combined to model systems. Emphasis will be placed upon the method by which the elements are symbolized and integrated into a pictorial representation of the system and, more importantly, how that information is translated into the appropriate operating rules for input into an NBDS simulation program. Having developed some actual sets of operating rules, the next section will demonstrate how they are input into the general purpose NBDS program and internally organized into a working set of rules.

7.1 Simple Queueing System

As an introductory example of how a network model can be translated into a set of NBDS operating rules, refer back to Figure 4-1. Pictured is a network diagram of a simple queueing system. Assume the system has the following characteristics:

- the time between arrivals to the queue is exponentially distributed with a mean of 5 minutes,
- arrivals to the queue wait for service in order of their arrival,
- the queue is serviced by a single service entity whose service time is uniformly distributed between 2 and 15 minutes,

- the simulation run ends after 1000 entities have been processed.

To translate Figure 4-1 into a set of NBDS operating rules requires just four lines:

```
'GEN1' 'RUE1' 'EXPON 5' 0 ' '
'QUE1' 'FIFO' '0' ' ' 'SRV1' ' '
'SRV1' 'TRM1' 'UNFRM 2 15' 1 ' '
'TRM1' 1000
```

When properly interpreted by the control program, this set of rules is all the NBDS program needs to successfully execute the simulation of this system.

7.2 Computer System With Preemptive Processing

As a means of introducing a network model containing the family of resource nodes, consider the network diagram pictured in Figure 7-1. The system pictured there could represent a simple computer system in which incoming jobs are placed in a queue until allocated a sufficient amount of memory space for processing by the CPU. The job queue in this case is modeled by a RESOURCE QUEUE node. Jobs arriving to the system queue up in an order inversely proportional to their memory requirements. Memory is allocated to the jobs from a finite source, here modeled by a Resource Bank. Once a job acquires memory space, processing begins by the CPU, represented in this case by regular

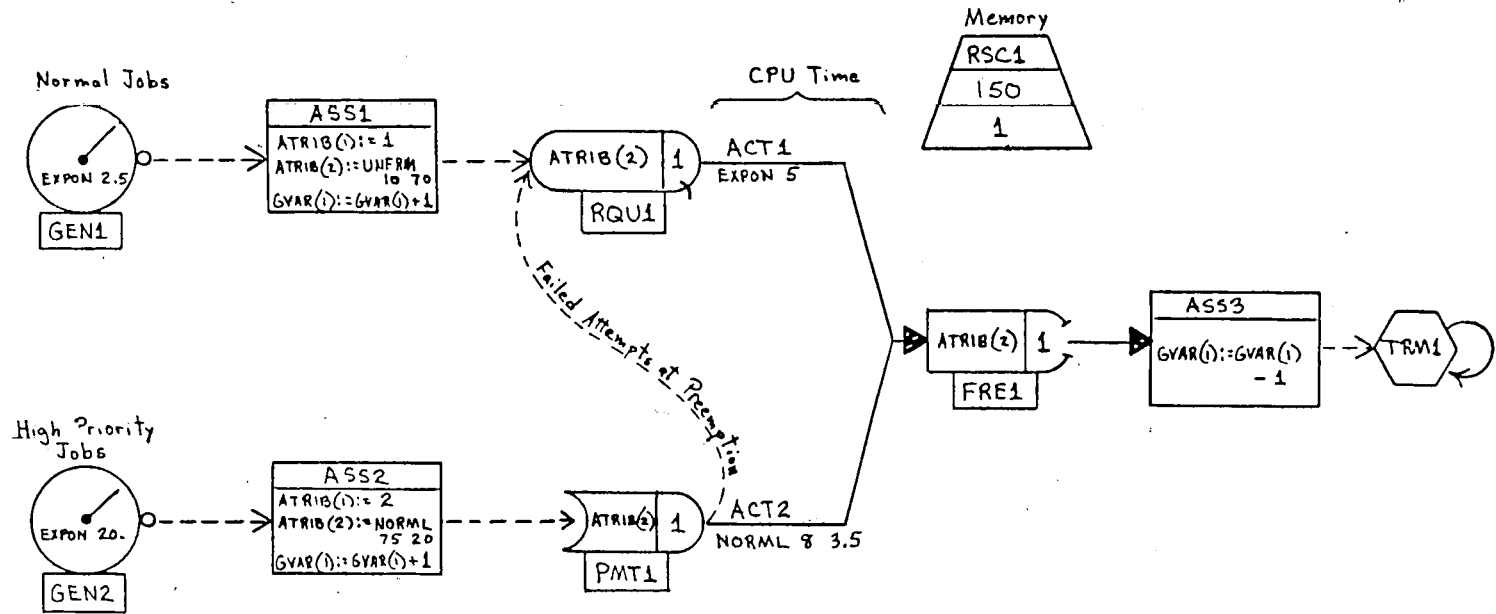


Figure 7-1. Network Diagram of Computer System Model

activities. Upon completion, the job relinquishes the memory by passing through a RESOURCE FREE node and leaves the system. While most of the jobs are processed in an order based on their memory requirement, some jobs enter the system that hold priority over all jobs irregardless of their memory demands. Those jobs will preempt jobs already being processed if enough memory is not available for use when they arrive. This element of the system is modeled with the RESOURCE PREEMPT node shown in Figure 7-1.

For illustrative purposes, assume the system has the following characteristics:

- normal jobs arrive to the system with exponentially distributed interarrival times whose mean is 2.5 sec; high priority jobs arrive with the time between jobs distributed exponentially and having a mean of 20 sec,
- the memory requirement of normal jobs is uniformly distributed between 10 and 70 pages; the memory required by high priority jobs is normally distributed with a mean of 75 ± 20 pages,
- the total available memory is 150 pages,
- CPU time of normal jobs is exponentially distributed with a mean of 5 sec; CPU time of high priority jobs is normally distributed with a mean of 8 ± 3.5 sec.

Figure 7-2 lists the set of translated operating rules describing the system above. Note how the priority between the two types of jobs is assigned with ATRIB(1)

```

'GEN1' 'ASS1' 'EXPON 2.5' 0 ' '
'GEN2' 'ASS2' 'EXPON 20.0' 0 ' '
'ASS1' 'RQU1' 'ATRIB(1):=1' 'ATRIB(2):=UNFRM 10 70' 'GVAR(1):=GVAR(1)+1'
'ASS2' 'PMT1' 'ATRIB(1):=2' 'ATRIB(2):=NORML 75 20' 'GVAR(1):=GVAR(1)+1'
'RSC1' 150 1
'RQU1' 'LVF2' 0 ' ' ' ' 1 'ATRIB(2)' 'ACT1'
'PMT1' 1 'ATRIB(2)' 'HV1' 1 'RQU1' 'ACT2'
'ACT1' 'FRE1' 'EXPON 5' ' '
'ACT2' 'FRE1' 'NORML 8 3.5' ' '
'FRE1' 1 'ATRIB(2)' 'ACT3'
'ACT3' 'ASS3' ' ' 'N/S'
'ASS3' 'TRM1' 'GVAR(1):=GVAR(1)-1'
'TRM1' ' '
'TIMD' 'Number Jobs in System' 1
'END' 3600

```

Figure 7-2. Operating Rules for Computer System Model

when the jobs enter the system. This attribute is later referenced in the priority rule of PMT1 (see HV1). Also note how the global variable GVAR(1) is used to keep track of the number of jobs in the system at any one time. Together with the TIMD statement, GVAR(1) is reported as a time dependent variable at the end of the simulation run. Also demonstrated is the use of ACT3 as a timeless activity to the next node with no statistics collected on it. Finally, the END statement indicates the simulation run is to end after 3600 simulated time seconds have elapsed.

7.3 Serial Work Stations on a Production Line

A more complicated scheme of SERVICE QUEUES is presented in this example where a portion of an automobile production line is modeled containing a series

of work stations. The network depicting this model is shown in Figure 7-3 where units arrive to the first work station for a particular set of operations and are then distributed between two separate work stations for another series of operations. The first work station is an area large enough to store three automobiles at a time (not including the ones being operated on) but the succeeding stations have room to store only one automobile apiece. Therefore, if each of the downstream work stations has a unit awaiting service when another arrives, the arrival is blocked. If the storage capacity of the first work station is exceeded, the excess automobiles are transported to a yard outside the manufacturing plant and stored there for later service.

For the purpose of this illustration, assume the additional system characteristics:

- the time between arrivals to the first work station is uniformly distributed between 12 and 20 minutes,
- the first work station is serviced by two parallel workers whose service times are normally distributed with a mean of 20 ± 5 minutes,
- the next two work stations are each manned by one person whose respective service times follow an Erlang distribution of 3 samples each with means of 12 and 15 minutes respectively,
- automobiles completing service by the first work station are distributed to the first of the next two parallel

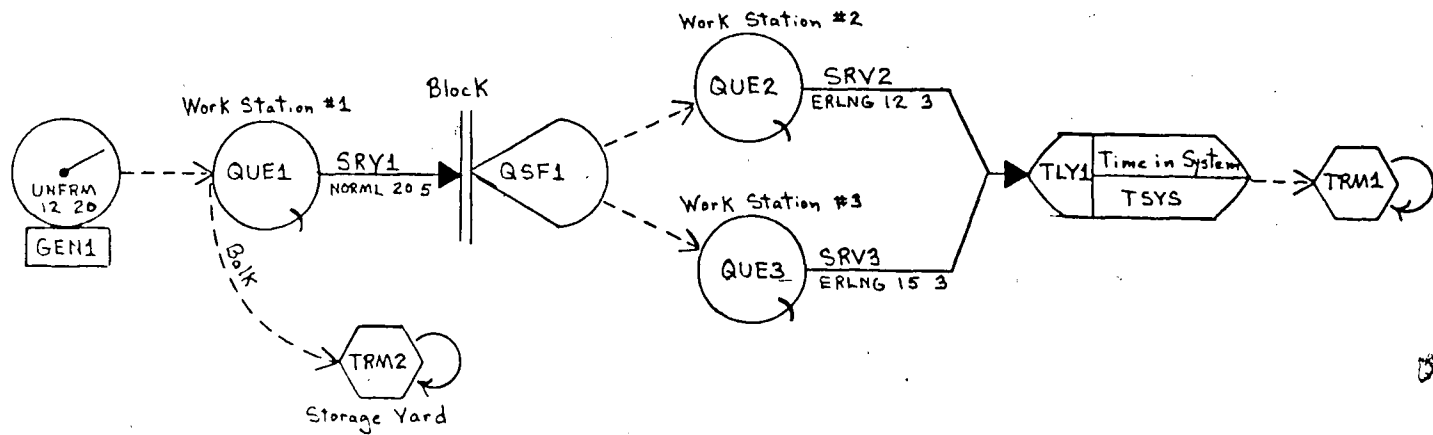


Figure 7-3. Network-Diagram of Production Line Model

stations which has room in its queue.

The system described above is translated into the set of operating rules shown in Figure 7-4. The concepts

```
'GEN1' 'QUE1' 'UNFRM 12 20' 0 ' '
'QUE1' 'FIFD' 0 3 'TRM2' 'SRV1' ' '
'SRV1' 'QSF1' 'NORML 20 5' 2 ' '
'QSF1' (2 3) 'SNQ' 'BLK'
'QUE2' ' ' ' ' 1 ' 'SRV2' 'QSF1'
'QUE3' ' ' ' ' 1 ' 'SRV3' 'QSF1'
'SRV2' 'TLY1' 'ERLNG 12 3' 1 ' '
'SRV3' 'TLY1' 'ERLNG 15 3' 1 ' '
'TLY1' 'TRM1' 'Time in System' 'TSYS' ('HIST' 40 180)
'TRM1' ' '
'TRM2' ' '
'END' 1000
```

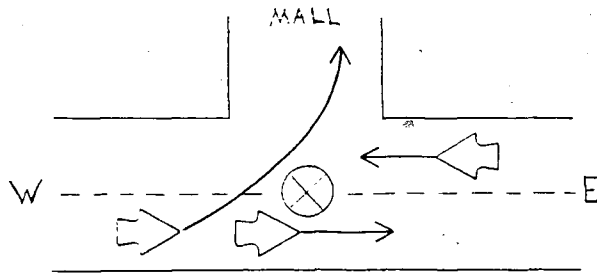
Figure 7-4. Operating Rules for Production Line Model

demonstrated in this example include balking, blocking, and queue selection. Also included in the network is a TALLY node to collect statistics on the time each unit spends in the system. Those data are summarized in a histogram at the end of the simulation as specified in TLY1.

7.4 Traffic Light

This last example of a traffic light features a more extensive network of resource nodes and also demonstrates the use of CONDITIONAL BRANCH nodes, the CLOSE node, and the OPEN node. It models a traffic light at a 3-way intersection pictured in the diagram

below:



The heavily travelled east-west street intersects with the entrance to a shopping mall. Eastbound traffic desiring to enter the mall must make a left turn in front of westbound traffic. Because these are single lanes, eastbound traffic backs up behind any cars waiting to make a left turn.

Again, for the purpose of illustration, assume the system has the additional characteristics:

- if cars arrive to the intersection when the light is green and there are no cars waiting in front of them, they pass straight through without delay; when traffic is backed up, cars passing through the intersection are delayed by a normally distributed time period of 3 ± 1.5 seconds which represents the time it takes the car to regain momentum,
- cars making a left turn into the mall are also delayed by a constant time of 1 second; cars turning right from the westbound lane experience no time delays and therefore right turns have no effect on the system,
- the light stops traffic in the E-W

directions for a period of 30 seconds while cars exit the mall; it then turns green on the eastbound side only for 15 seconds to give any cars starting out a clear path to make a left turn; after that 15 seconds, the light turns green on the westbound side and both lanes are allowed passage for the next 45 seconds until the light turns red again,

- the arrival pattern of cars from each direction in exponentially distributed with an average of 7 seconds between cars westbound and an average of 8.5 seconds between cars eastbound; also, one out of every ten eastbound cars make a left turn.

Figure 7-5 contains the network diagram of this traffic light system. Note how the lanes are modeled with Resource Banks, each having a capacity of one unit. To pass through the intersection, therefore, each car must acquire the unit of resource assigned to its lane at its respective RESOURCE NODE. However, eastbound cars turning left into the mall also request the use of the resource unit assigned to westbound traffic. Since westbound traffic has priority over that resource, any cars making a left turn must wait until all westbound traffic passes or until they are allowed the 15 second free period at the start of a cycle. This network model also demonstrates several uses of CONDITIONAL BRANCH nodes. In the first case they are used with probability branching to direct 10% of the eastbound arrivals to ASS1 while the remainder are directed to ASS2. At each ASSIGNMENT node the arrivals are assigned a value in their first ATRIB

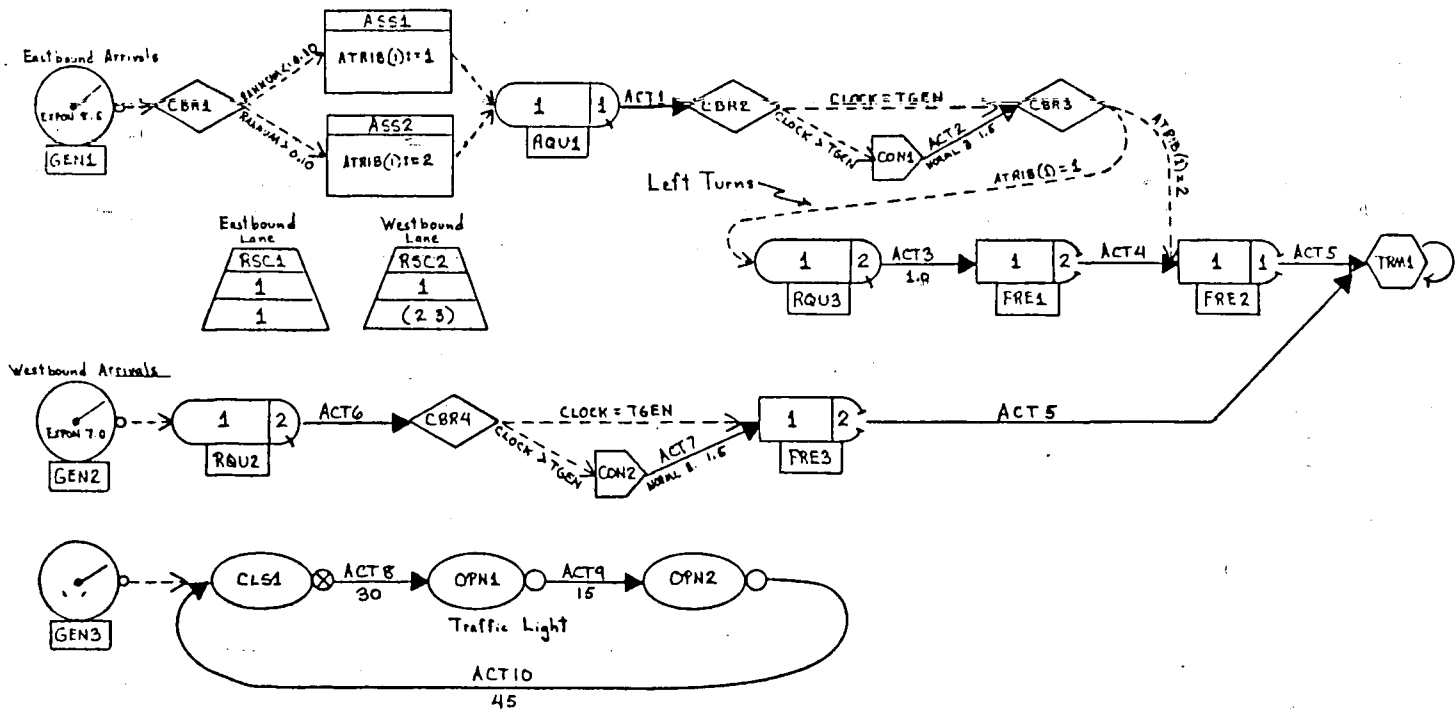


Figure 7-5. Network Diagram of Traffic Light Model

array which is used by CONDITIONAL BRANCH nodes later in the network to separate those cars making a left turn. Also note how a set of CONDITIONAL BRANCH nodes and the use of the NBDS variables CLOCK and TGEN determines whether a car has been waiting in line or not. If found to be waiting, the car is directed over an activity having the appropriate time delay. Finally, an effective use of CLOSE and OPEN nodes is illustrated in which they simulate the phases of the traffic light. Note how only one entity is initially generated to CLS1 to begin the traffic light cycle. Figure 7-6 lists the NBDS operating rules required to carry out a simulation of this model for a simulated time period of one hour.

```

'GEN1' 'CBR1' 'EXPON 8.5' 0 ' '
'GEN2' 'RQU2' 'EXPON 7' 0 ' '
'GEN3' 'CLS1' ' ' 0 1
'RSC1' 1 1
'RSC2' 1 (2 3)
'RQU1' 'FIFO' 0 ' ' ' 1 '1' 'ACT1'
'RQU2' 'FIFO' 0 ' ' ' 2 '1' 'ACT4'
'RQU3' 'FIFO' 0 ' ' ' 2 '1' 'ACT3'
'FRE1' 2 '1' 'ACT4'
'FRE2' 1 '1' 'ACT5'
'FRE3' 2 '1' 'ACT5'
'CBR1' ('RANUM <= 0.10' 'ASS1') ('RANUM > 0.10' 'ASS2')
'CBR2' ('CLOCK = TGEN' 'CBR3') ('CLOCK > TGEN' 'CON1')
'CBR3' ('ATRIB(1) = 1' 'RQU3') ('ATRIB(1) = 2' 'FRE2')
'CBR4' ('CLOCK = TGEN' 'FRE3') ('CLOCK > TGEN' 'CON2')
'CON1' 'ACT2'
'CON2' 'ACT7'
'ASS1' 'RQU1' 'ATRIB(1):=1'
'ASS2' 'RQU1' 'ATRIB(1):=2'
'CLS1' 'RQU' (1 2) ' ' 'ACT8'
'OPN1' 'RQU' 1 'ACT9'
'OPN2' 'RQU' 2 'ACT10'
'ACT1' 'CBR2' ' ' '
'ACT2' 'CBR3' 'NORML 3 1.5' 'N/S'
'ACT3' 'FRE1' '1' ' '
'ACT4' 'FRE2' ' ' 'N/S'
'ACT5' 'TRM1' ' ' 'N/S'
'ACT6' 'CBR4' ' ' '
'ACT7' 'FRE3' 'NORML 3 1.5' 'N/S'
'ACT8' 'OPN1' '30' 'N/S'
'ACT9' 'OPN2' '15' 'N/S'
'ACT10' 'CLS1' '45' 'N/S'
'TRM1' ' '
'END' 3600

```

Figure 7-6. Operating Rules for Traffic Light Model

8. General Purpose NBDS Package

One of the primary objectives of this thesis was to develop a set of Nial-based operations that could be used to prototype decision-making systems employing discrete simulation. As the first few simulation elements were developed and tested for this purpose, an interactive query/answer routine was used to input the various operating rules into the control program. However, as the list of simulation elements grew, this task became slow and cumbersome which led to the design of a non-interactive, batch-read technique to inputting the rules. What evolved as a result of all this work was actually a rudimentary simulation language whose features include a set of operating rules having their own vocabulary and syntax. Admittedly this general purpose NBDS package lacks many essential features of a good simulation language (eg. extensive error checks and debugging facilities), but its being presented here for three important reasons:

1. To learn how to build a working prototype simulation system using NBDS, one must understand how to correctly organize the operating rules so that the control program is able to interpret them properly; also, one must learn how to access the statistical arrays at the end of the simulation run to report the desired results. Use of the general purpose NBDS package provides a good vehicle to learn those tasks.

2. While developing the model of a system one wishes to build a specific simulation package for, use of the general purpose NBDS package is very helpful in testing certain elements of the model, developing the required set of operating rules, and deciding what data are necessary to report.
3. The current set of simulation elements together with the control program provide a good basis on which to build and test additional simulation operations when needed for a specific modeling purpose.

The discussion of the general purpose NBDS package will begin by detailing how the operating rules are organized within the program. Following this, is a description of the summary statistics automatically printed at the end of a simulation run. Finally, examples of actual general purpose NBDS runs will be presented using some of the operating rules developed in the previous section.

8.1 Input and Internal Organization of Operating Rules

Once entered into the NBDS control program, each set of operating rules is organized into a three level nested array. At the top level is the single collective array (eg. Qrules, Srvrules, Inits, etc.) which holds the middle level containing one or more individual sets of like operating rules. At the bottom level are the individual rule elements owned by each set of operating rules. To obtain their singularity within

the top level, each set of like operating rules must be entered into its collective parent array as a solitary array. Furthermore, each solitary array of rules must be ordered within its parent array so that its address corresponds to the integer number of its rules identification label. Since numbering of the rules always begins with 1, the first position in each parent array (address 0) is simply a blank character string. Each parent array is initialized with a blank character string by the operation INIT_RULES at the start of a simulation run. The use of this dummy position is similar to the use of the first row of dummy values in every statistical array: it maintains the logical correspondence between the rules' set number and its position in the parent array.

As an example of an ordered array of operating rules, consider the following collection of Srvrules pictured in the sketch mode:

```

+-----+
|:SRV1:QSF1:NCRML 20 5:2:|:SRV2:TLY1:ERLNG 12 3:1:|:SRV3:TLY1:ERLNG 15 3:1:|
+-----+

```

The individual sets of rules shown above were taken from the list of operating rules for the Production Line example given in the previous section (see Figure 7-4).

Note the empty character string in the first position of the parent array Srvrules and the logical ordering of its member arrays in the next lower level.

The Srvrules array given above is in a form ready for execution within the NBDS program. It is an important task of one developing a prototype simulation package to organize all the sets of rules used by the system in a similar fashion. In the general purpose NBDS package this process is carried out automatically by the operation READ_INPUT. All one has to do is list the operating rules in a script file named Input.Dat in a format similar to the examples given in the previous section. Once execution begins, each line of the script file is read into the program, converted into a solitary array, and linked with other member rule sets of its kind.

The code for READ_INPUT that performs these tasks is displayed in Figure 8-1. Note how the use of the CASE-expression provides a convenient way to select the appropriate set of operating rules by keying on the rules identification label of each set (here assigned to the variable Typ). It also provides the means for an error check on the rules labels by defaulting to an error message if the code does not match any of those listed. Both these techniques are used in many other NBDS operations where selections are made from several


```

      READ_INPUT IS (
EOF:= EXECUTE '??eof';
NFILE:= OPEN 'INPUT.DAT 'r;
LINE:= EXECUTE READFILE NFILE;
NAME:= FIRST LINE; DATE:= SECOND LINE; TITLE:= THIRD LINE;
EACH WRITESCREEN ' ' ('Input Statements For ' LINK TITLE
LINK ':')
LINE:= EXECUTE READFILE NFILE;
WHILE LINE ^= EOF DO
  WRITE LINE;
  TYP:= 3 TAKE (FIRST LINE);
  CASE TYP FROM
'GEN' : GENRULES:= GENRULES LINK SOLITARY LINE END
'QUE' : QRULES:= QRULES LINK SOLITARY LINE END
'QSF' : QSFRULES:= QSFRULES LINK SOLITARY LINE END
'QSB' : QSBRULES:= QSBRULES LINK SOLITARY LINE END
'SSL' : SSLRULES:= SSLRULES LINK SOLITARY LINE END
'SRV' : SRVRULES:= SRVRULES LINK SOLITARY LINE END
'RSC' : RSCRULES:= RSCRULES LINK SOLITARY LINE END
'RQU' : RQRULES:= RQRULES LINK SOLITARY LINE END
'FRE' : FREERULES:= FREERULES LINK SOLITARY LINE END
'ALT' : ALTRULES:= ALTRULES LINK SOLITARY LINE END
'FMT' : PMTRULES:= PMTRULES LINK SOLITARY LINE END
'CLS' : CLSRULES:= CLSRULES LINK SOLITARY LINE END
'OPN' : OPNRULES:= OPNRULES LINK SOLITARY LINE END
'ACT' : ACTRULES:= ACTRULES LINK SOLITARY LINE END
'TRM' : TERMRULES:= TERMRULES LINK SOLITARY LINE END
'MLT' : MULTRULES:= MULTRULES LINK SOLITARY LINE END
'CBR' : CBRNRULES:= CBRNRULES LINK SOLITARY LINE END
'MBR' : MBRNRULES:= MBRNRULES LINK SOLITARY LINE END
'CON' : CONTRULES:= CONTRULES LINK SOLITARY LINE END
'ASS' : ASSNRULES:= ASSNRULES LINK SOLITARY LINE END
'TLY' : TALLYRULES:= TALLYRULES LINK SOLITARY LINE END
'TIM' : TIMRULES:= TIMRULES LINK SOLITARY LINE END
'INI' : INITS:= INITS LINK SOLITARY LINE END
'SEE' : CHOOSE_SEED (SECOND LINE) END
'END' : TERMT:= SECOND LINE END
ELSE EACH WRITESCREEN ' ' ((FIRST LINE) LINK ' NOT A VALID RULE
CODE.')
      ABORT:= END_SIM:= 1;
  ENDCASE;
  LINE:= EXECUTE READFILE NFILE;
ENDWHILE;
CLOSE NFILE;

```

Figure 8-1. Nial Code for Read-Input Operation

expressions depending upon a key rules identifier. Also note line 5 of Figure 8-1. As it implies, the first line of every Input.Dat file must contain the name of the simulator, the date, and title of the run. All these variables are entered as character strings.

If the sets of operating rules within a given class are listed in Input.Dat in a continuous, logical order starting with rule number 1, no further action would be required by the NBDS program to organize the rules following READ_INPUT. However, the general purpose NBDS package goes one step further by providing a set of rules sort operations which allow one to enter the rules into the script file in any order. The sets of rules are also allowed discontinuous numbering. The operation at the top level of these sorting procedures is named RULE_SORT. It also performs the critical task of creating and initializing the arrays used to maintain statistics on the various simulation elements. Because it keys on the number of elements in a given class, RULE_SORT creates just that amount of storage space required to collect statistics on the elements of that particular run.

8.2 Statistical Analysis and Summary Report

Upon detecting the end of a simulation run, the general purpose NBDS control program exits the event processing loop and begins executing a series of

statistics update and reporting operations. Update operations are used to update any time related statistics associated with a given simulation element. For example, if service queues were used in a particular run, the operations UPDATE_Q and SUMMARY_Q would be executed in that order at the end of the simulation. UPDATE_Q updates time dependent variables related to service queue lengths while SUMMARY_Q estimates average queue lengths, waiting times in the queue and average times between balks from the queues. In addition, SUMMARY_Q selects other pertinent data from Qstats, organizes all the summary data for output, and then prints a summary report of descriptive queue statistics. Similar operations are carried out for other elements in the simulation. Table 8-1 summarizes the statistical results automatically printed at the end of a general purpose NBDS run for queues, activities, resource banks, time independent and time dependent variables.

8.3 General Purpose NBDS Examples

8.3.1 Basic Execution Procedures

Assuming the user has already prepared a script file of operating rules and has access to a Nial workspace, the execution of a general purpose NBDS requires just three steps:

Table 8-1. Output Statistics of a General Purpose NBDS
Run

Service and Resource Queues

- Average queue length
- Maximum queue length
- Number of entities left in queue at end
- of those entities receiving service, the average delay time in the queue for just those entities not immediately served
- the average waiting time of all entities receiving service
- Number of balks from the queue
- Average time between balks from the queue

Resource Banks

- Current capacity of the bank
- Average utilization of bank over time
- Maximum number of resource units utilized at one time
- Current number of resource units utilized

Service Activities

- Number of servers in activity
- Current number of busy servers
- Total numbers of entities served

Table 8-1: (continued)

Service Activities (continued)

- Average utilization of service activity
- Fractional average of time the service activity is blocked
- Average service time including wait in queue
- Minimum service time including wait in queue
- Maximum service time including wait in queue
- Maximum idle time for a single server activity or the maximum number of idle servers at one time for a multiple server activity
- Maximum busy time for a single server activity or the maximum number of busy servers at one time for a multiple server activity

Regular Activities

- Average number of entities routed over the activity at one time
- Maximum number of activities routed over the activity at one time
- Current number of entities engaged in activity
- Total number of entities routed over activity

Time Independent Variables

- Mean and standard deviation of observations
- Minimum observed value

Table 8-1. (continued)

Time Independent Variables (continued)

- Maximum observed value
- Total number of observations

Time Dependent Variables

- Mean and standard deviation over time
- Minimum observed value
- Maximum observed value
- Current value

- 2
1. the script file of NBDS operations* is loaded into the workspace and evaluated using the command:

LOADDEFS "NBDS.NDF

(note: for a silent load, use "NBDS 0)

2. the command GO is entered which initiates execution of the control program; at this point, an introductory header is printed followed by a listing of all the operating rules as they appear in the file Input.Dat,
3. after all the operating rules are printed, a message follows requesting the user to check the input statements for obvious errors; if an error is detected, the user simply keys CTRL G which aborts the run and drops him back into the command mode; from there, the required corrections can be made by accessing the host editor with the EDIT "INPUT.DAT command; if corrections were needed, Step 2 is repeated; otherwise, the user simply keys RETURN which initiates execution.

When the simulation is finished executing, a second header is printed containing all the information in the first line of Input.Dat (simulator's name, run title, and date). Also printed will be the simulated start and finish times of the run. Finally, the summary statistics will be printed for all those elements in the simulation that appear in Table 8-1.

*note: the DECSYSTEM-20 would not allocate sufficient workspace to load NBDS.NDF in its entirety; therefore, the script of unused operations was replaced with the empty array NULL which preserved the operation in name but freed up needed workspace.

8.3.2 Computer System With Preemptive Processing

As a first example of a general purpose NBDS simulation, consider the computer system modeled in Section 7.2. An identification statement was added to the top of the list of operating rules shown in Figure 7-2 and the entire set entered into the author's DECSYSTEM-20 directory as the file INPUT.DAT. After loading NBDS.NDF and entering the GO command, the header was printed along with an echo listing of the operating rules (in sketch mode) as shown in Figure 8-2. Upon checking the rules for errors and keying RETURN, a message indicating that the program was executing appeared.

The output of summary results for this simulation run is displayed in Figure 8-3. As shown, the run ended at simulated time 3600. Reported are the standard results for resource queues, resource banks, regular activities, and time dependent variables. Statistics of interest for this particular run might include the memory queue length (average length of Que 1), waiting time of all jobs in the queue (average wait time in Que 1), utilization of computer memory (average utilization of Res 1), number of preemptions (count for Act 2), number of jobs processed by the CPU (count for Act 1), and average number of jobs in the system at one time (mean value for time dependent variable "Number Jobs in System.").


```

Q*Nial: Twenex Release 1 Version 3.02
clear workspace
LOADDEFS *NBDS 0
GO

```

```

*****
*                                     *
*               General Purpose       *
*               Nial-Based           *
*               Discrete Simulations  *
*                                     *
*****

```

Input Statements For COMPUTER SYSTEM EXAMPLE:

```

+-----+
|GEN1|ASS1|EXPON 2.5|0| |
+-----+
+-----+
|GEN2|ASS2|EXPON 20.0|0| |
+-----+
+-----+
|ASS1|RQU1|ATRIB(1)|=1|ATRIB(2)|=UNFRM 10 70|GVAR(1)|=GVAR(1)+1|
+-----+
|ASS2|PMT1|ATRIB(1)|=2|ATRIB(2)|=NORML 75 20|GVAR(1)|=GVAR(1)+1|
+-----+
|RSC1|150|1|
+-----+
|RQU1|LVF2|10| |1|ATRIB(2)|ACT1|
+-----+
|PMT1|1|ATRIB(2)|HV1|1|RQU1|ACT2|
+-----+
|ACT1|FRE1|EXPON 5| |
+-----+
|ACT2|FRE1|NORML 8 3.5| |
+-----+
|FRE1|1|ATRIB(2)|ACT3|
+-----+
|ACT3|ASS3| IN/S|
+-----+
|ASS3|TRM1|GVAR(1)|=GVAR(1)-1|
+-----+
|TRM1| |
+-----+
|TIMD|Number Jobs in System|1|
+-----+
|END|3600|
+-----+

```

Check input statements for obvious errors. If none, key RETURN
else key CTRL G to abort run.

Program executing. Please wait.....

Figure 8-2. Echo Listing of Rules for Computer System Run

00000000 SUMMARY RESULTS 00000000

Run Id: COMPUTER SYSTEM EXAMPLE
 Simulator: RICK GELL
 Run Date: 1 AUGUST 1984

Simulation Started @ Time : 0
 Simulation Ended @ Time : 3600

0000 RESOURCE QUEUE STATISTICS 0000

QUEUE DELAY STATISTICS

Queue #	Avg Lnsth	Max Lnsth	No. Remain	Avg Delay Time \$
1	113.1827 +/- 3.5595	151		5111.568 +/- 28.666

ARRIVAL TO START-OF-SERVICE STATISTICS

Queue #	Avg Wait Time \$	Max Wait T	No. Balks	Avg T Btwn Balks
1	117.0019 +/- 23.004	381.84	0	no value

\$ for those arrivals which do not immediately acquire resources
 \$* for all arrivals to queue which acquire resources

0000 RESOURCE BANK STATISTICS 0000

Res #	Current Capac	Avg Utilization	Max Util	Current Util
1	150	106.25 +/- 39.611	149.94	105.46

0000 REGULAR ACTIVITY STATISTICS 0000

Act #	Avg Utilization	Max Util	Current Util	Count
1	112.0325 +/- 1.1321	6	1	1633
2	210.29763 +/- 0.50405	21	1	134

0000 TIME DEPENDENT VARIABLE STATISTICS 0000

Identification	Mean Value	Min Value	Max Value	Current Value
Number Jobs in System	5.5157 +/- 4.0095	0	19	7

Figure 8-3. Summary Results for Production Line Run

8.3.3 Serial Work Stations on a Production Line

After adding an identification statement to the beginning of the operating rules listed in Figure 7-4, the example of the production line model in Section 7.3 was simulated. The summary report is listed in Figure 8-4 and demonstrates the standard set of results for service queues, service activities, and time independent variables. Statistics of interest in this simulation run might include the number of autos waiting at each station for service (average queue lengths), the time those autos spent in the queue waiting for service (average delay time and average wait time), the number of autos which had to be bypassed to storage in the yard (number of balks), the utilization of each work station (average utilization of service activity), fraction of time the workers at the first work station were blocked while transporting autos to the next two parallel stations (average blockage of Srv 1), and the average time a unit spent in the system (mean value for time independent variable "Time in System"). Also generated by this simulation run was a histogram displaying the distribution of times-in-the-system for all the autos processed (see Figure 8-5).

8.3.4 Traffic Light

For the sake of completeness, the traffic

Run Id: PRODUCTION LINE EXAMPLE
 Simulator: RICK SELL
 Run Date: 1 AUGUST 1984

Simulation Started @ Time : 0
 Simulation Ended @ Time : 1000

**** REGULAR QUEUE STATISTICS ****

QUEUE DELAY STATISTICS

Queue #	Avg Lnsth	Max Lnsth	No. Remain	Avg Delay Time \$
1	111.2481 +/- 1.1297	31		2134.834 +/- 26.094
2	210.88165 +/- 0.32302	11		1134.534 +/- 19.229
3	310.85425 +/- 0.35286	11		1140.925 +/- 22.851

ARRIVAL TO START-OF-SERVICE STATISTICS

Queue #	Avg Wait Time \$	Max Wait T	No. Balks	Avg T Bwn Balks
1	1122.578 +/- 26.809	106.87		6160.137 +/- 81.265
2	2131.976 +/- 20.646	73.072	0	no value
3	3135.587 +/- 25.487	107.05	0	no value

\$ for those arrivals which do not receive immediate service
 \$\$ for all arrivals to queue which receive service

**** SERVICE ACTIVITY STATISTICS ****

UTILIZATION-RELATED STATISTICS

Srv #	No. Srvs	No. Busv @ End	No. Srvd	Avg Utilization	Avg Blocksd
1	21	21	5310.55139 +/- 0.33985	0.35547	
2	11	11	2610.97205 +/- 0.14484	0.	
3	11	11	2210.92174 +/- 0.26858	0.	

SERVICE TIME-RELATED STATISTICS \$

Srv #	Avg Service Time	Min Srv T	Max Srv T	Max Idle T/Srvs	Max Busv T/Srvs
1	1143.053 +/- 27.966	8.8898	130.04	21	21
2	2168.846 +/- 27.34	13.806	142.28	17.92	958.24
3	3175.074 +/- 31.737	12.563	128.12	64.131	894.63

\$ include wait in queue

**** TIME INDEPENDENT VARIABLE STATISTICS ****

Identification	Mean Value	Min Value	Max Value	No. Obsvrs
Time in System	128.32 +/- 49.79	27.994	220.43	48

Figure 8-4. Summary Results for Production Line Run

HISTOGRAM FOR Time in System

Count	Rel Freq	Cum Freq	Cell Limit	20	40	60	80	100
3	0.0625	0.0625	40	***				
0	0.	0.0625	50.667	C				
1	0.020833	0.083333	61.333	* C				
0	0.	0.083333	72.	C				
5	0.10417	0.1875	82.667	***** C				
1	0.020833	0.20833	93.333	* C				
7	0.14583	0.35417	104.	***** C				
6	0.125	0.47917	114.67	***** C				
2	0.041667	0.52083	125.33	** C				
3	0.0625	0.58333	136.	*** C				
3	0.0625	0.64583	146.67	*** C				
4	0.083333	0.72917	157.33	**** C				
2	0.041667	0.77083	168.	** C				
0	0.	0.77083	178.67	C				
3	0.0625	0.83333	189.33	*** C				
3	0.0625	0.89583	200.	*** C				
5	0.10417	1.	Infinity	***** C				
48				20	40	60	80	100

Figure 8-5. Histogram for Production Line Run

light example in Section 7.4 was simulated with the operating rules listed in Figure 7-6. The summary report is displayed in Figure 8-6. Obviously the statistics of interest for a traffic light simulation would be the line length of cars stemming in both directions from the light (average queue lengths) and the time a driver had to wait at the light until he could pass (average wait time in queue).

***** SUMMARY RESULTS *****

Run Id: TRAFFIC LIGHT EXAMPLE
 Simulator: RICK GELL
 Run Date: 15 JULY 1984

Simulation Started @ Time : 0
 Simulation Ended @ Time : 3600

**** RESOURCE QUEUE STATISTICS ****

QUEUE DELAY STATISTICS

Queue #	Avg Lnsth	Max Lnsth	No. Remain	Avg Delay Time
1	112.1414 +/- 2.5983	141	8	25.846 +/- 18.5891
2	213.7688 +/- 3.383	151	5	30.021 +/- 17.2891
3	310.11577 +/- 0.31994	11	1	22.903 +/- 15.0971

ARRIVAL TO START-OF-SERVICE STATISTICS

Queue #	Avg Wait Time	Max Wait T	No. Bkks	Avg T Btwn Bkks
1	117.898 +/- 19.535	103.641	0	no value
2	2126.03 +/- 19.057	80.692	0	no value
3	318.4648 +/- 14.352	44.108	0	no value

* for those arrivals which do not immediately acquire resources
 ** for all arrivals to queue which acquire resources

**** RESOURCE BANK STATISTICS ****

Res #	Current	Capacity	Avg Utilization	Max Util	Current Util
1			110.38297 +/- 0.48611	11	11
2			110.37902 +/- 0.48514	11	11

**** REGULAR ACTIVITY STATISTICS ****

Act #	Avg Utilization	Max Util	Current Util	Count
1	110. +/- 0.	11	01	413
2	310.012778 +/- 0.11231	11	01	461
3	610. +/- 0.	11	01	519

SIMULATION RUN COMPLETE!!!

Figure 8-6. Summary Results for Traffic Light Run
 - 147 -

9. Verification of Modeling Elements

Each NBDS modeling element presented in this thesis underwent a battery of manual checks to verify that the functional units of code performed as they were intended, generating the correct statistical results. In addition to simple hand simulations on paper, visual run-time checks were carried out by taking advantage of Nial's useful "picture" facility. Entire arrays containing statistical data or entity records filed in a queue or on the event calendar were output before and after each important step by inserting simple "write" commands in the code. For instance, a WRITE QSTATS command placed before and after a record was filed in a queue would display the entire contents of the statistical array QSTATS. The components of the array acted upon during the event were then immediately checked for correctness. Another useful facility was the BREAK command; when encountered during execution, evaluation of the expression it was contained in would stop immediately, giving the user total control of the environment. The contents of any array or the value of any variable could then be inspected by simply entering its name. By typing RETURN, execution resumed at the exact point where it was interrupted.

To present verification checks for each NBDS

modeling element is beyond the scope of this thesis.

However, the application of some basic queueing theory will demonstrate that the underlying queueing principles of the elements are valid.

The basic theory of a single-server queueing system was developed by Khintchine and Polloczek and results in the following formula (17):

$$E(w) = \frac{\rho^2}{2(1-\rho)} \left\{ 1 + \left[\frac{\sigma_{t_s}}{E(t_s)} \right]^2 \right\} \quad (5)$$

where $E(w)$ = mean number of items waiting for service (not including one being served)

ρ = facility utilization of one serving facility

$E(t_s)$ = mean service time for all items

σ_{t_s} = standard deviation of service times

This formula is used to make queue size estimates in a variety of applications. It applies to exponential interarrival times, any distribution of service times, and any dispatching discipline provided that its selection of the next item to be serviced does not depend on the service time.

To test whether this formula applies to the queueing mechanisms built into the NBDS modeling elements, a set of operating rules (like the ones in Section 7.1) were created to model a single-server queueing system. Exponential interarrival times and service times were

specified with mean values of 5 and 10 time units respectively. A series of 25 general purpose NBDS runs were conducted with this set of rules, each with different initial seed values for the random number generators. Each run was allowed to proceed until 1000 entities were processed.

Having collected 25 independent determinations for $E(w)$, its sample mean and standard deviation were estimated. Table 9-1 lists those results as 0.481 ± 0.093 respectively along with the individual values of $E(w)$ for each run. Since $\sigma_{t_s} = E(t_s)$ for exponential distributions and $\rho = 0.50^*$, the Khintchine-Polloczek formula predicts $E(w)$ to be 0.50 for these simulation runs. Assuming the results for $E(w)$ are normally distributed, a test-of-hypothesis was performed to determine whether the queueing model agrees with the P-K theory. The test statistic used was:

$$t = \frac{\bar{x} - u_0}{s / \sqrt{n}} \quad (6)$$

where x = sample mean

u_0 = population mean

s = sample standard deviation

n = sample size

* $\rho = E(n) \cdot E(t_s)$ where $E(n)$ is the inverse of the interarrival time.

Table 9-1. Results of Queueing Model Verification Runs

<u>Seed Stream</u>	<u>Observed E(w)</u>
1	0.394
2	0.468
3	0.440
4	0.644
5	0.496
6	0.364
7	0.490
8	0.412
9	0.502
10	0.582
11	0.485
12	0.372
13	0.337
14	0.610
15	0.680
16	0.463
17	0.524
18	0.396
19	0.475
20	0.558
21	0.410
22	0.449
23	0.635
24	0.466
25	<u>0.384</u>

$$t = \frac{\bar{x} - u_0}{s / \sqrt{n}}$$

$$\frac{0.481 - 0.50}{0.093 / \sqrt{25}} = 1.018$$

$$'d.f.' = n-1 = 24$$

$$\bar{x} = 0.481$$

$$s = 0.093$$

Equation 6 has a student's t distribution with $(n-1)$ degrees of freedom (18).

To reject the null hypothesis that $E(w) = 0.50$ at a 0.05 level of significance, the absolute value of t must exceed 2.064 for a two-tailed test (18). Substituting the given values of x , u_0 , s , and n into Equation 6 yields a value of 1.018 with 24 degrees of freedom. Therefore, the null hypothesis is not rejected and NBDS is shown to be an adequate tool for simulating the behavior of normal queueing systems.

10. Prototyping Special Purpose Simulations With NBDS

The largest tasks involved in prototyping special purpose NBDS systems are understanding the individual modeling elements, their associated set of operating rules, and the organization of those rules within the NBDS program. Once this has been accomplished and a model of the system of interest is in hand, the job is reduced to designing a user interface for entering the variables of the system into the NBDS program and designing a summary report of the results. Since users of the prototype should not be required to understand how to use NBDS itself, the batch technique of inputting the operating rules to the program employed by the general purpose NBDS package is unacceptable. Therefore, the basic structure of the operating rules used in the simulation must be defined beforehand followed by an interactive mode by which the user inputs only those rules (or variables) of interest. Likewise, a prototype simulation package should not have to rely on the generalized summary report provided by the general purpose program. More descriptive headings are required and only those descriptive statistics that are pertinent to the simulation should be reported.

This next section will highlight some of the basic operations of input and output for specialized NBDS

packages. The integration of those procedural operations with the baseoperations already provided by NBDS will also be discussed. However, in order to provide a basis for that discussion, this section will begin by presenting an example of a specialized NBDS prototype that could be used by telecommunications network analysts.

10.1 Communications Line Simulation Prototype

10.1.1 Description of Model

This prototype simulation package was designed to investigate the behavior of a full-duplex multidrop communications line linking several terminals to a central computer. Since terminals along a multidrop line must share the line, a queueing problem will develop for both input and output messages. Of particular interest is the average response time of messages sent from the terminals (ie. time interval from the operator's pressing the last key of the input to the terminal's displaying of the last character of the response). In this model, two types of messages are allowed, each having its own distribution of input/output character lengths and each having the ability to be assigned its own queueing priority.

Figure 10-1 displays the network diagram of this system. Note how service queues and activities are used

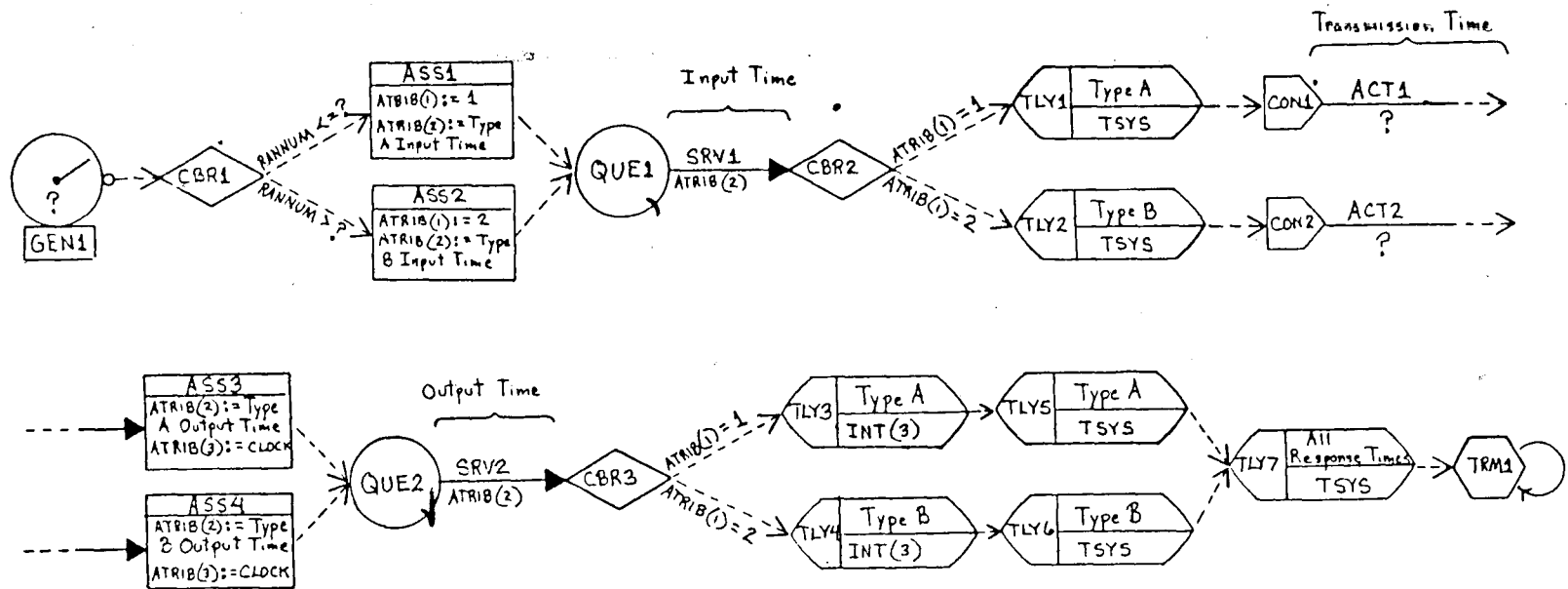


Figure 10-1. Network Diagram of Communications Line Model

to simulate the input and output lines respectively. Only one GENERATE node is used to represent the arrival of all input messages to the system. Also note the use of a CONDITIONAL BRANCH node and ASSIGNMENT nodes to characterize the different percentage and type of each input message. Regular activities are used to model the time spent in the computer by each message. TALLY nodes preceded by CONDITIONAL BRANCH nodes are used to collect statistics throughout the network.

10.1.2 Execution of Program

The script file of operations for this package is named COMLINE.NDF and also resides in the Lehigh University Computer Center tape library under Volume Serial Number JCW002. Like the general purpose NBDS script file, COMLINE.NDF is loaded into a Nial workspace using the "loaddefs" command. Once loaded, program execution begins by typing GO. At this point the initialization steps are performed by the INITIALIZE operation and operation PRINT_INTRO prints a header and message describing the purpose of the simulation package. Next follows a lengthy dialog containing a mixture of question-answer statements and menu listings. This interface is carried out by READ_INPUT and allows the user to input all the variables required by the program.

Figure 10-2 displays a sample of this input session.

6. Enter input line preparation time (seconds): 0.5

7. Enter the line speed (characters/second): 600

8. From the following, what probability distribution describes the length of time (seconds) Type A messages spend in the computer?

1. UNIFORM
2. EXPONENTIAL
3. NORMAL
4. ERLANG

Enter number of choice: 3
Enter expected value and standard deviation: 1.0 0.5

9. From the following, what probability distribution describes the length of time (seconds) Type B messages spend in the computer?

1. UNIFORM
2. EXPONENTIAL
3. NORMAL
4. ERLANG

Enter number of choice: 1
Enter minimum and maximum values: 0.2 2.0

10. From the following, what probability distribution describes the output message character lengths of Type A transactions?

1. UNIFORM
2. EXPONENTIAL
3. NORMAL
4. ERLANG

Enter number of choice: 3
Enter expected value and standard deviation: 750. 530.

11. From the following, what probability distribution describes the output message character lengths of Type B transactions?

1. UNIFORM
2. EXPONENTIAL
3. NORMAL
4. ERLANG

Enter number of choice: 3
Enter expected value and standard deviation: 395. 200.

12. How do output transactions contend for the line?

1. FIFO
2. LIFO
3. Type A First
4. Type B First

Enter number of choice: 4

13. Enter output line preparation time (seconds): 0.7

14. Enter desired length of time (minutes) for simulation: 60

15. A histogram of response times for all transactions is provided.
Enter best estimate of minimum and maximum range (seconds): 2.0 7.5

16. Enter seed stream number (from 1 to 10) for random number generator: 5

Program executing. Please wait.....

Figure 10-2. (continued)

Note the menus of probability distributions and queuing priorities to choose from in most of the queries. Input variables requested by the program include: interarrival times of input messages, message size distributions, fashion in which messages contend for the input and output lines, line speed, line preparation times, and length of time for the simulation run. The user is also asked to supply a minimum and maximum estimate of all response times to set the limits of a histogram summarizing those results.

Upon supplying all the necessary information, execution of the simulation run begins followed by a printout of the summary results (shown in Figure 10-3). The summary results contain queue length and utilization statistics for the input/output lines, transmission times of input/output messages, and response times for each and all message types. Counts of each message type are also given. The summary results conclude with a histogram illustrating the distribution of all response times and is shown in Figure 10-4.

10.1.3 Initialization and Input of Operating Rules

Since the user of an NBDS prototype only supplies some of the operating rules to the simulation program, the basic framework within which they

***** SUMMARY RESULTS *****

Simulation Started @ Minute : 0
 Simulation Ended @ Minute : 60.
 Seed Stream Number : 5

LINE STATISTICS *

Line	Avg Queue Length	Max Queue Length	Avg Wait For Line	Avg Line Utilization
Input	10.0042384 +/- 0.064965		110.0089492 +/- 0.047076	10.27254 +/- 0.44526
Output	0.49657 +/- 0.81315		611.0486 +/- 1.9142	10.75546 +/- 0.42981

INPUT TIME STATISTICS *

Message Type	Mean Value	Min Value	Max Value	No. Obsvrs
Type A	10.59714 +/- 0.10509	0.50021	1.2756	471
Type B	10.57953 +/- 0.088684	0.50003	1.2435	1234
All Transactions	10.58439 +/- 0.093805	0.50003	1.2756	1705

OUTPUT TIME STATISTICS *

Message Type	Mean Value	Min Value	Max Value	No. Obsvrs
Type A	14.0603 +/- 3.2648	0.72864	23.503	471
Type B	12.1044 +/- 0.95296	0.70837	6.8136	1232
All Transactions	12.6453 +/- 2.0895	0.70837	23.503	1703

RESPONSE TIME STATISTICS *

Message Type	Mean Value	Min Value	Max Value	No. Obsvrs
Type A	15.686 +/- 3.3028	1.609	24.979	471
Type B	13.7814 +/- 1.0241	1.7495	8.72	1232
All Response Times	14.3082 +/- 2.1207	1.609	24.979	1703

* All times in seconds

Figure 10-3. Summary Results for Communications Line Prototype

- 161 -

HISTOGRAM FOR All Response Times

Count	Rel Frea	Cum Frea	Cell Limit	20	40	60	80	100
16	0.0093952	0.0093952	2.1	C				
67	0.039342	0.048738	2.3667	*C				
111	0.065179	0.11392	2.7333	*** C				
187	0.10981	0.22372	3.1	***** C				
231	0.13564	0.35937	3.4667	*****	C			
230	0.13506	0.49442	3.8333	*****		C		
209	0.12272	0.61715	4.2	*****			C	
155	0.091016	0.70816	4.5667	****				C
124	0.072813	0.78097	4.9333	***				C
87	0.051086	0.83206	5.3	**				C
73	0.042866	0.87493	5.6667	**				C
50	0.02936	0.90429	6.0333	*				C
28	0.016442	0.92073	6.4					C
25	0.01468	0.93541	6.7667					C
10	0.005872	0.94128	7.1333					C
9	0.0052848	0.94656	7.5					C
91	0.053435	1.	Infinity	**				C
1703				20	40	60	80	100

Figure 10-4. Histogram for Communications Line Prototype

reside must already exist by the time the READ_INPUT operation is executed. Therefore, it is the job of the one who designs the simulation package to define each set of operating rules used by the model before execution begins. Furthermore, where member rules exist that must be supplied by the user, a position must be reserved for them in their parent array so as to maintain the critical ordering of rules within that set.

As an example of how this is carried out in COMLINE, NDF, the code for the initialization operation INIT_RULES is illustrated in Figure 10-5. Each set of operating rules used by the program is defined in that operation. Note how the first position of each class of rules is a blank character string. This is to maintain the correspondence between the integer number of each set of rules and its position in the array at the top level (ie. there can be no set of Qrules identified as QUE0). Also note the use of asterisks in many of the rule sets. These represent rule elements within a character string or individual rules themselves that must be supplied by the user during the query session. For instance, observe the set of Genrules as it appears after the initialization operation is executed:

```

INIT_RULES IS (
GENRULES:= ' ' LINK SOLITARY ('GEN1' 'CBR1' '* *' 0 ' ');
CBRRULES:= ' ' LINK SOLITARY ('CBR1' ('RANNUM <=*' 'ASS1')
('RANNUM>*' 'ASS2')) LINK SOLITARY ('CBR2' ('ATRIB(1)=1' 'TLY1')
('ATRIB(1)=2' 'TLY2')) LINK SOLITARY ('CBR3' ('ATRIB(1)=1' 'TLY3')
('ATRIB(1)=2' 'TLY4')));
ASSNRULES:= ' ' LINK SOLITARY ('ASS1' 'QUE1' 'ATRIB(1)=1'
'ATRIB(2)=(( * *)/*)+*') LINK SOLITARY ('ASS2' 'QUE1' 'ATRIB(1)=2'
'ATRIB(2)=(( * *)/*)+*') LINK SOLITARY ('ASS3' 'QUE2' 'ATRIB(3)=CLOCK'
'ATRIB(2)=(( * *)/*)+*') LINK SOLITARY ('ASS4' 'QUE2' 'ATRIB(3)=CLOCK'
'ATRIB(2)=(( * *)/*)+*');
QRULES:= ' ' LINK SOLITARY ('QUE1' '* 0 ' ' ' ' 'SRV1')
LINK SOLITARY ('QUE2' '* 0 ' ' ' ' 'SRV2');
SRVRULES:= ' ' LINK SOLITARY ('SRV1' 'CBR2' 'ATRIB(2)' 1 ' ')
LINK SOLITARY ('SRV2' 'CBR3' 'ATRIB(2)' 1 ' ');
TALLYRULES:= ' ' LINK SOLITARY ('TLY1' 'CON1' 'Type A' 'TSYS')
LINK SOLITARY ('TLY2' 'CON2' 'Type B' 'TSYS')
LINK SOLITARY ('TLY3' 'TLY5' 'Type A' 'INT(3)')
LINK SOLITARY ('TLY4' 'TLY6' 'Type B' 'INT(3)')
LINK SOLITARY ('TLY5' 'TLY7' 'Type A' 'TSYS' ' ')
LINK SOLITARY ('TLY6' 'TLY7' 'Type B' 'TSYS' ' ')
LINK SOLITARY ('TLY7' 'TRM1' 'All Response Times' 'TSYS'
('HIST * * * * *'));
CONTRULES:= ' ' LINK SOLITARY ('CON1' 'ACT1') LINK SOLITARY
('CON2' 'ACT2');
ACTRULES:= ' ' LINK SOLITARY ('ACT1' 'ASS3' '* *' 'N/S' )
LINK SOLITARY ('ACT2' 'ASS4' '* *' 'N/S');
TERMRULES:= ' ' LINK SOLITARY ('TRM1' ' ');
QSFRULES:= VOYD);

```

Figure 10-5. Nial Code for INIT_Rules of Communications Line Prototype

```

+-----+
|+-----+-----+-----+-----+|
|:GEN1:CBR1:* *:0:|
|+-----+-----+-----+-----+|
+-----+

```

rf

The character string '* *' represents the time interval between generations where the individual asterisks represent the probability distribution and its associated set of parameters requested by query No. 1 of the input session. The more complicated sets of asterisks in the Assnrules represent the probability distributions and their associated parameters describing the individual message lengths, the line speed, and the line preparation times respectively. They compute to the service times used by each entity in downstream service activities and are stored in ATRIB(2).

Once the operating rules have been defined in the initialization step, it's the job of the READ_INPUT operation to replace each variable represented by a symbol with a real value prompted from the user. This is done by inserting the value returned by a "read" operation into its corresponding position in the designated set of operating rules. For instance, consider the following segment of code which executes the first query shown in Figure 10-2:


```

EACH WRITESCREEN '
'1. From the following, what probability distribution describes the
' time between arrivals (in seconds) of all input transactions?';
GENRULES:= ((LINK ((CHOOSE_DIST CHOOSE_PARAMS) (0 2) PLACEALL
                (THIRD SECOND GENRULES))) 2 PLACE SECOND GENRULES)
                1 PLACE GENRULES;

```

The operation CHOOSE_DIST supplies the menu of probability distributions, reads the choice of the user, and returns as its value the distribution code corresponding to the user's choice. Likewise, CHOOSE_PARAMS prints the statement requesting the parameters associated with the given distribution and returns as its value the parameters read by it. Each value is then inserted into the first set of Genrules at the positions held by the asterisks. The result is a completed set of Genrules pictured below:

```

+-----+
| +-----+ |
| : GEN1:CBR1:ERLNG 0.7 3:0: |
| +-----+ |
+-----+

```

This same technique was used in all the queries of READ_INPUT where dummy rules needed to be replaced with real values.

10.1.4 Access to Statistical Arrays for Summary Report

Section 4.4 detailed the organization of

the statistical arrays created for each class of modeling elements in an NBDS simulation run. Those classes include both types of queues, both types of activities, resource banks, designated time dependent variables, and designated time independent variables. Each time a class of modeling elements is represented in an NBDS simulation run, a two-dimensional array is created for that set with each row but the initial dummy one belonging to a given modeling element within that class. It is the task of one developing a prototype NBDS package to selectively pick from these arrays the data he wishes to process and report at the end of a simulation run. To aid in this procedure, Appendix C details the components maintained by every statistical array in NBDS. The components are listed in order of their column position in the array and are identified by the variable names used within the NBDS program. Each component is also accompanied by a brief description of its role in the array.

In some cases, the statistical components of interest need no further processing before reporting (eg. maximum queue length, number of entities served, etc.). These data are simply selected from the given array using the appropriate address and placed in another array designated for output. However, in instances where sample means and standard deviations are required, additional

processing is required. Sample means based on observations are simply calculated by dividing the array member holding the accumulated sum of observations by the array member holding the count for that variable. Sample means for time-persistent variables are calculated by dividing the array member holding the accumulated sum of the $x(t) \cdot dt$ statistic by the program variable CLOCK (total time interval). Standard deviation calculations are more complicated and require the use of the specially built operations STDV and GRPSTDV. STDV is used for statistics based upon observations and requires as its arguments the previously estimated mean, the array member holding the accumulated sum of the x^2 statistic, and the array member holding the observation count for that variable. GRPSTDV is used for time-persistent variables and requires as its arguments the accumulated $x^2(t) \cdot dt$ statistic, the program variable CLOCK, and the previously calculated mean.

To demonstrate how some of these calculations are carried out at the end of a simulation run, consider the array of accumulated statistics for the queue node QUE1 as it existed at the end of the COMLINE simulation just presented:

```
o 0 1 3405.6 15.258 15.258 15.258 3.9129 0.50864 93 1705 0 0 0 0 0
```

If the above array is assigned to the variable Qdat, the following expression would estimate the average waiting time, Avgw, of all entities receiving service at QUE1:

$$\text{Avgw} = (6 \text{ pick Qdat}) / (10 \text{ pick Qdat})$$

where the accumulated sum of waiting times (15.258) is held at address 6 and the count of entities passing through the queue (1705) held at address 10. Furthermore, the standard deviation of waiting times, Stdw, is estimated with:

$$\text{Stdw} = \text{STDV Avgw} (7 \text{ pick Qdat}) (10 \text{ pick Qdat})$$

where the accumulated sum of squares of waiting times (3.9129) is held at address 7.

10.1.5 Output of Summary Results

Nial provides two useful operations for outputting information to a screen or printer--the WRITSCREEN and WRITE operations. WRITSCREEN displays the value of its character string argument and was used to generate the introductory script and table headings in all the NBDS examples illustrated in this thesis. The WRITE operation displays the value of its argument and was used to print all the summary statistics. As with all data in Nial, the results are expressed as arrays which can contain a mixture of data types. Each row in the tables of statistical results represents a single array or list of data objects. For instance, the array pictured

below is a list of character strings containing all the column titles for the table Response Time Statistics of COMLINE:

```

+-----+-----+-----+-----+
|Message Type|Mean Value|Min Value|Max Value|No. Obsrvs|
+-----+-----+-----+-----+

```

The next array contains the data of the second row of Response Time Statistics; note the mixture of data types:

```

+-----+-----+-----+-----+
|Type A|5.686 +/- 3.3028|1.609|24.979|471|
+-----+-----+-----+-----+

```

After all the necessary calculations have been made and the chosen data arranged into arrays like those above, the results are conveniently arranged into tables using the primitive "mix" operation. A "mix" of a list of lists of the same length results in a table with the lists as rows. To illustrate this, assume the two arrays presented earlier are assigned to the variables A and B respectively. Observe the effect of the next assignment:

RESULTS:= solitary A link solitary B

```

+-----+-----+-----+-----+-----+-----+
|Message Type|Mean Value|Min Value|Max Value|No. Obsrvs||Type A|5.686 +/- 3.3028|1.609|24.979|471|
+-----+-----+-----+-----+-----+-----+

```

Here the variable RESULTS becomes a list of the lists A and B. Now observe the effect of the "mix"

operation on RESULTS:

mix RESULTS

Message Type	Mean Value	Min Value	Max Value	No. Obsrvs
Type A	5.686 +/- 3.3028	1.609	24.979	471

This technique was used to generate all the tables of summary statistics displayed in NBDS examples throughout this thesis. Histograms also begin as a list of solitary arrays with each array representing an individual cell of the histogram. The "mix" operation is used to create its final form.

10.1.6 Integration of Operations into a Working Program

Once all the input and output operations of an NBDS prototype have been defined, the simplest way to create a working program is to edit those operations into the original script file of baseoperations for the general purpose NBDS package (NBDS.NDF). In many cases operations designed for a special purpose simulation have a similar function to ones in the general purpose package. When this occurs, the easiest thing to do is replace the general purpose operation with the new one while retaining its original name. For example, the INIT_RULES operation of COMLINE.NDF shown in Figure 10-5 replaced the generalized operation by the same name in

NBDS.NDF. Likewise, the READ_INPUT, SUMMARY_Q, and SUMMARY_IND_STATS operations of COMLINE.NDF all replaced operations by the same name and with a similar function in NBDS.NDF. The advantage of this procedure lies in not having to redefine these operation names in the top level GO control operation. It also helps maintain the logical sequence of input/output operations. In some instances, new operations (like PRINT_INTRO of COMLINE.NDF) need to be added to the original script file. When this is done, a reference to it must be added to the GO operation in its logical position.

After all the new input/output operations have been edited into the original NBDS script file, the next task is to eliminate all unnecessary operations so that the specialized version can be loaded into a Nial workspace without exceeding its capacity. The hierarchical listing of baseoperations in Appendix A aids one in determining which operations are required to support the given elements in the prototype. Those that are deemed unnecessary can be culled from the main script file. For instance, since INIT_RULES of COMLINE.NDF already defines the operating rules in logical order, there is no need for the SORT_RULES operation. Therefore, it is edited from the new script file and, since SORT_RULES will no longer be defined, the reference to it in the higher

level operation RULE_SORT is removed as well. In some cases an unneeded operation is referenced many times throughout the script file which would result in extensive editing if it is eliminated. A simple remedy for these situations is to replace the code for that operation with the empty array NULL. This is the safest technique to use in all cases but, as one familiar with the evaluation mechanism of Nial can see, either method requires experimentation. That is, make the change, reload the script file, and check for any resultant errors; repeat this process until all operations are fully defined.

11. Conclusions

The work presented in this thesis demonstrates how the Nial language, with its unique approach to handling data and its rich pool of primitive operations, is ideally suited for programming discrete simulations on a digital computer. Discrete event simulations demand a great deal of recordkeeping in the form of maintaining ordered lists of records, searching and selecting records from those lists, and creating new records as well as destroying old ones. All of these programming tasks are conveniently handled with Nial due to its inherent array-as-data-object concepts and ability to operate on nested arrays with ease. The result is a greatly reduced programming effort compared to that required by other general purpose computer languages performing the same tasks.

Nial's ability to treat arrays as single data objects provided an efficient means for manipulating entity records in the NBDS simulations. Records containing an entity's entire list of attributes were transferred from one file to another with little programming effort. The filing of these records in ordered lists also required no need for a complex system of pointers--it's all embedded in the language itself. Likewise, operations on individual elements of a record

were easily carried out and, when used in combination with one of Nial's powerful transformers, provided a means for accessing a given attribute in several different records at once.

Nial's only drawback as a base language for computer simulations is its relatively slow execution time compared to compiled languages (even the simple verification runs of a single-server queueing system required 20-30 minutes of run-time under low load conditions). For this reason it would not be practical to use Nial as a production language for computer simulation. However, its functional design, combined with the programming features just presented, lend Nial as a useful tool for prototyping specialized discrete simulation packages. This was demonstrated through the design of the many functional program units that supported a variety of simulation modeling elements. A generalized simulation package was designed as a vehicle for experimenting with these modeling elements and ultimately serves as the framework for developing specialized prototypes. One simply has to design a problem-specific interface for inputting the various operating rules to the system and tailor the summary report to suit the specific needs of the prototype.

The work presented in this thesis demonstrates

Nial's usefulness as a prototyping tool in other applications as well. As shown here, the conciseness and power of its primitive operations greatly reduces the programming effort of complex operations. Furthermore, Nial's design allows one to decompose a problem into several functional units, thus, helping to clarify the program logic. However, from this researcher's own personal experience, Nial's main attraction as a prototyping tool stems from its interactive nature and its ability to display the results of an operation on an array as a picture. Each operation contained in the library of NBDS baseoperations is the result of several iterative sessions at a terminal. Typically, a command or expression was issued and its effect on the target array examined through its picture. This experimental process continued until the unit of code produced the desired result. Once an entire operation was completed and fully tested, it was copied into the permanent script file of NBDS baseoperations. Without this functional approach to problem solving and the interactive environment provided by Nial, the list of simulation elements resulting from this process would never have been as extensive.

List of References

1. Jenkins, A.M., and Naumann, J.D. "The Prototype Model as a MIS Design Technique," Discussion Paper No. 163, Graduate School of Business, Indiana University (September 1980), p. 1.
2. Canning Publications, Inc. "Developing Systems by Prototyping," EDP Analyzer, Vol. 19, No. 9 (September 1981), pp. 5-6.
3. Appleton, D.S. "Data Driven Prototyping," Datamation (November 1983), pp. 259-268.
4. Jenkins, M.A. The Q'Nial Reference Manual, Release 10, Queen's University, Kingston, Canada (1983).
5. More, T. "Notes on the Diagrams, Logic and Operations of Array Theory," Tech. Rep. G320-2137, IBM Scientific Center, Cambridge (September 1981).
6. Jenkins, M.A. "A Development System for Testing Array Theory Concepts," APL81, APL Quote Quad, Vol. 12, No. 1 (September 1981) pp. 152-159.
7. Hinden, H.J. "Lisp/APL Merger Supports Functional Programming Concepts," Computer Design (April 1984), pp 29-31.
8. Emschoff, J.R. and Sisson, R.L. Design and Use of Computer Simulation Models. New York: The MacMillan Co., 1970, p. 264.
9. Shannon, R.E. Systems Simulation: The Art and Science, Englewood Cliffs, NJ: Prentice Hall, 1975.
10. Krasnow, H.S. and Merikallio, R. "The Past, Present, and Future of General Simulation Languages," Management Science, Vol. XI, No. 2 (1964) p. 236-267.
11. Nicholls, J.E. The Structure and Design of Programming Languages, Reading, MA: Addison Wesley Publishing Co., 1975.

12. Mitrani, I. Simulation Techniques for Discrete Event Systems, Cambridge, Great Britain: Cambridge University Press, 1982, pp. 27-31.
13. Pritsker, A.B. and Pegden, C.D. Introduction to Simulation and SLAM, New York: Halsted Press, 1979.
14. Fishman, G.S. Concepts and Methods in Discrete Event Digital Simulation, New York: John Wiley & Sons, 1973, Chap. 7,8.
15. Deo, N. System Simulation With Digital Computer, Englewood, NJ: Prentice-Hall, Inc. 1983, pp. 43-46.
16. Gordon, G. System Simulation. 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978, Chap. 7.
17. Khintchine, A.Y. Mathematical Methods in the Theory of Queueing, New York: Hafner, 1960.
18. Mendenhall, W., Schaeffer, R.L. and Wackerly, D.D. Mathematical Statistics With Applications, Second Ed., Boston: Duxbury Press, 1981.

12. Mitrani, I. Simulation Techniques for Discrete Event Systems, Cambridge, Great Britain: Cambridge University Press, 1982, pp. 27-31.
13. Pritsker, A.B. and Pegden, C.D. Introduction to Simulation and SLAM, New York: Halsted Press, 1979.
14. Fishman, G.S. Concepts and Methods in Discrete Event Digital Simulation, New York: John Wiley & Sons, 1973, Chap. 7,8.
15. Deo, N. System Simulation With Digital Computer, Englewood, NJ: Prentice-Hall, Inc. 1983, pp. 43-46.
16. Gordon, G. System Simulation. 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978, Chap. 7.
17. Khintchine, A.Y. Mathematical Methods in the Theory of Queueing, New York: Hafner, 1960.
18. Mendenhall, W., Schaeffer, R.L. and Wackerly, D.D. Mathematical Statistics With Applications, Second Ed., Boston: Duxbury Press, 1981.

Appendix A. Hierarchical Organization of
NBDS Baseoperations

1. Simulation Control Program Operations

• GO

```
INITIALIZE
    INIT_CONSTS
    INIT_RULES
    INIT_VARS
READ_INPUT
    CHOOSE_SEED
RULE_SORT
    SORT_RULES
    SORT_QUEUES
    SORT_RSCRULES
    SORT_ASSNRULES
    SORT_TALLYRULES
LOAD_CAL
    CREATE_FIRST
    QUEUE_FIRST
    RQUEUE_FIRST
    SORTUP
ARRIVE
    CREATE
    CHOOSE_DEST
        (see top level operations of
        Arrival Event Operations)
    SORTUP
DEPART
    (See top level operations of
    Departure Event Operations)
All UPDATE operations
All SUMMARY operations
```

2. Arrival Event Operations

- ASIGN
 - COMPTRIBS
 - COMPGLOBS
 - SWITCH_DEST
- CBRANCH
 - SWITCH_DEST
- Q_SLCT_FWD
 - SWITCH_DEST
- ARV_QUE
 - ENTER_QUE
 - QSTATS_IN
 - ORDER_QUE
 - CHK_IF_BLKD
 - BALK
 - TERMINATE
 - CHECK_SRVRs
 - TALLY_SERVICE
 - CHK_IF_BLKD
 - SET_BLK_FLG
- ARV_RQ
 - ENTER_RQ
 - QSTATS_IN
 - ORDER_QUE
 - BALK
 - TERMINATE
 - TALLY_RSC
 - START_ACTIVITY
 - TALLY_ACT
- FREE_RSC
 - TALLY_RSC
 - START_ACTIVITY
 - TALLY_ACT
 - POLL_QUEs
 - PROCESS_RQF
 - START_ACTIVITY
 - TALLY_ACT
 - PROCESS_PRMPTE
 - TALLY_ACT
 - QSTATS_OUT

- ALTER_RSC
 - TALLY_RSC
 - START_ACTIVITY
 - POLL_QUES
 - (see POLL_QUES of FREE_RSC)

- PREEMPT
 - SWITCH_DEST
 - ATEMPT_PRMP
 - PREEMPTED
 - ATRIB
 - ALLOW_PRMP
 - QSTATS_IN
 - TALLY_ACT
 - START_ACTIVITY
 - TALLY_ACT
 - TALLY_RSC
 - POLL_QUES
 - (see POLL_QUES of FREE_RSC)
 - TALLY_RSC
 - START_ACTIVITY
 - TALLY_ACT

- MULTIPLY

- MBRANCH

- KONTINUE
 - START_ACTIVITY
 - TALLY_ACT

- CLOSE_Q
 - Q_CLOSED
 - QSTATS_IN
 - BALK
 - TERMINATE
 - START_ACTIVITY
 - TALLY_ACT

- OPEN_Q
 - START_ACTIVITY
 - TALLY_ACT
 - CHECK_SRVRS
 - TALLY_SERVICE
 - CHECK_QUE
 - (see CHECK_QUE in
Departure Event Operations)

TALLY_RSC
POLL_QUES
(see POLL_QUES of FREE_RSC)

- TALLIE
 BTWNTALLY
 TALLY_HISTS
 NUMTALLY
 TALLY_HISTS
 SWITCH_DEST

- TERMINATE

3. Departure Event Operations

- END_SERVICE
 SUM_SERVICE
 TERMINATE
 TALLY_SERVICE
 Q_SLCT_BHND
 RANNUM
 CHECK_QUE
 TALLY_SERVICE
 SERVICE_QUE
 RANNUM
 QSTATS_OUT
 CHECK_FOR_BLKs
 RESET_BLK_FLG
 TALLY_SERVICE
 FIND_HOME_QUE
 Q_SLCT_BHND
 RANNUM
 CHECK_QUE
- END_ACTIVITY
 TALLY_ACT
 TERMINATE

Appendix B. Location of Operations in
NBDS.NDF Script File

<u>Operation Name</u>	<u>Page</u>	<u>Line No.</u>
ACT_ADDR5	1	48900
ACTDAT	1	49100
ALLOW_PRMP	2	34200
ALTER_RSC	2	27000
ARRIVE	2	69900
ARV_QUE	2	12100
ARV_RQ	2	22000
ASIGN	2	8200
AEMPT_PRMP	2	31900
ATRI5	1	30000
BALK	1	58800
BTWNTALLY	2	60000
CBRANCH	2	10100
CHECK_FOR_BLK5	1	92200
CHECK_QUE	1	97100
CHECK_SRVR5	1	83000
CHK_IF_BLKD	1	74200
CHOOSE_DEST	2	67400
CHOOSE_SEED	1	22000
CLOSE_Q	2	49600
COMPGLOB5	1	7700
COMPTRIB5	1	21200
CREATE	1	36600
CREATE_FIRST	2	73400
DEPART	2	4700
END_ACTIVITY	1	50800
END_SERVICE	2	1300

<u>Operation Name</u>	<u>Page</u>	<u>Line No.</u>
ENTER_QUE	1	75900
ENTER_RQ	1	70000
ERLNG	1	32600
EXPON	1	31500
FIND_HOME_QUE	1	90000
FREE_RSC	2	24700
GO	3	27100
GRPSTDV	1	34300
GVAR	1	30200
INIT_CONSTS	1	2100
INIT_RULES	1	3700
INIT_VARS	1	2800
INITIALIZE	1	4500
KONTINUE	2	43800
LOAD_CAL	2	83800
MBRANCH	2	42600
MULTIPLY	2	41100
NORML	1	31700
NUMTALLY	2	62300
ORDER_QUE	1	60400
OPEN_Q	2	51400
POLL_QUE	1	64800
PREEMPT	2	38100
PREEMPTED	2	29600
PRINT_HISTOS	3	18600
PROCESS_PRMPF	1	63600
PROCESS_RQF	1	62300
Q_CLOSED	2	44600
Q_SLCT_BHND	1	84400
Q_SLCT_FWD	2	15600
QSTATS_IN	1	54200
QSTATS_OUT	1	56100
QUEUE_FIRST	2	75200

<u>Operation Name</u>	<u>Page</u>	<u>Line No.</u>
RANNUM	1	27700
READ_INPUT	1	22800
RESET_BLK_FLG	1	73800
RQUEUE_FIRST	2	79700
RULE_SORT	1	17900
SELECT_SRVR	1	7900
SERVICE_QUE	1	95400
SET_BLK_FLG	1	73400
SORT_ASSNRULES	1	10100
SORT_QRULES	1	16700
SORT_RSCRULES	1	15100
SORT_RULES	1	4700
SORT_TALLYRULES	1	12300
SORTDOWN	1	36200
SORTUP	1	35900
SRV_ADDRS	1	45700
SRVDAT	1	45900
START_ACTIVITY	1	49300
STDV	1	34600
SUM_SERVICE	1	43400
SUMMARY_ACT	3	11800
SUMMARY_IND_STATS	3	14600
SUMMARY_Q	2	87300
SUMMARY_RQ	3	500
SUMMARY_RSC	3	7700
SUMMARY_SRVS	2	94600
SUMMARY_TDP_STATS	3	24800
SWITCH_DEST	2	7000
TALLIE	2	64200
TALLY_ACT	1	46100
TALLY_HISTS	2	57800
TALLY_RSC	1	52300
TALLY_SERVICE	1	39500
TALLY_TIMED	1	5700
TERMINATE	1	30400
TGEN	1	38200
UNFRM	1	31300
UPDATE_ACT	3	9900
UPDATE_Q	2	85500
UPDATE_RQ	2	98600
UPDATE_RSC	3	5900
UPDATE_SRVS	2	92700
UPDATE_TDP_STATS	3	23100
VALID_NUMS	2	72600

Appendix C. Components of NBDS Statistical Arrays

● Service Queues (Qstats) and Resource Queues (Rqststs)

<u>Address</u>	<u>Variable Name</u>	<u>Description</u>
0	FLG	Boolean flag indicating when queue is blocked; used only by service queues
1	QN	number of entities waiting in queue
2	QMAX	maximum queue length
3	QT	time of last state change in queue
4	SUMFQ	cumulative sum of $(\text{CLOCK}-\text{QT}) \cdot \text{QN}$; divided by CLOCK, yields average number of entities in queue at any one time
5	SUMFQ2	cumulative sum of $(\text{CLOCK}-\text{QT}) \cdot \text{QN}^2$; passed as first argument to GRPSTDV operation to estimate standard deviation of queue length over time
6	SUMQT	cumulative sum of waiting times in queue
7	SUMQT2	cumulative sum of squares of waiting times in queue; passed as second argument to STDV operation to estimate standard deviation of waiting times
8	MAXQT	maximum waiting time in queue
9	QDEPART	number of entities departing queue that had to wait for service

Appendix C (continued)

10	NTHRU	total number of entities departing from queue
11	TB	time of last balk from queue
12	SUMBT	cumulative sum of (CLOCK-TB) or time between last balk
13	SUMBT	cumulative sum of squares of times between last balk; passed as second argument to STDV operation to estimate standard deviation of times between balks
14	NBALK	number of entities balking from queue
15	GTFLG	flag indicating whether queue open (0) or closed (1)

• Service Activities (Srvstats)

<u>Address</u>	<u>Variable Name</u>	<u>Description</u>
0	SN	number of entities served
1	SUMSRVT	cumulative sum of service times
2	SUMSRVT2	cumulative sum of squares of service times; passed as second argument to STDV operation to estimate standard deviation of service times
3	MINT	minimum service time
4	MAXT	maximum service time
5	ST	time of last state change in service activity

Appendix C (continued)

6	SUMFT	cumulative sum of (CLOCK-ST)* utilization where utilization is equal to the number of "busy" servers/total number of available servers; divided by CLOCK, yields average server utilization
7	SUMFT2	cumulative sum of (CLOCK-ST)* (Utilization) ² ; passed as first argument to GRPSTDV operation to estimate standard deviation of server utilization over time
8	NBUSY	number of servers engaged in an activity
9	MAXIDL	maximum idle time for one server; where there is more than one server, this holds the maximum number of servers idle at one time
10	MAXBSY	maximum busy time for one server; where there is more than one server, this holds the maximum number of servers busy at one time
11	NSRVS	designated number of servers for activity
12	SUMBLKT	cumulative sum of (number of blocked servers/total number of servers)*(CLOCK-ST); divided by CLOCK, yields average blocking time
13	NBLKS	number of blocked servers at a given time
14	BLKFLG	flag indicating whether destination queue is blocked (1) or free to receive entities (0)

● Regular Activities (Actstats)

Appendix C (continued)

<u>Address</u>	<u>Variable Name</u>	<u>Description</u>
0	AN	number of entities currently engaged in activity
1	AT	time of last state change in activity
2	SUMFT	cumulative sum of (CLOCK-AT) * AN; divided by CLOCK, yields average number of entities in activity at one time
3	SUMFT2	cumulative sum of (CLOCK-AT) * AN ² ; passed as first argument to GRPSTDV operation to calculate standard deviation of number of entities in activity over time
4	UMAX	maximum number of entities in activity at one time
5	CNT	number of entities routed over activity

● Resource Banks (Rscstats)

<u>Address</u>	<u>Variable Name</u>	<u>Description</u>
0	INIT	capacity of resource bank
1	REMAIN	current number of available resources
2	RT	time of last state change in resource bank
3	SUMFT	cumulative sum of (CLOCK-RT) * utilization where utilization equals INIT-REMAIN; divided by CLOCK, yields average utilization of resource bank

Appendix C (continued)

4	SUMFT2	cumulative sum of (CLOCK-RT) * (utilization) ² ; passed as first argument to GRPSTDV operation to estimate standard deviation of resource bank utilization over time
5	UMAX	maximum utilization of resource bank at one time

● Time Independent Variables (Tallystats)

<u>Address</u>	<u>Variable Name</u>	<u>Description</u>
0	NUMS	number of observations
1	SUMX	cumulative sum of observations
2	SUMX2	cumulative sum of squares of observations; passed as second argument to STDV operation to estimate standard deviation of all observations
3	MINX	minimum observed value
4	MAXX	maximum observed value
5	LAST_T	time of last observation; used only with BTWN option of TALLY nodes

● Time Dependent Variables (Glbstats)

0	LST_VAL	value of last observation
1	LT	time of last state change in variable
2	FX	cumulative sum of (CLOCK-LT) * LST_VAL; divided by CLOCK, yields average value over time

Appendix C (continued).

3	FX2	cumulative sum of (CLOCK-LT) * (LST_VAL) ² ; passed as second argument to GRPSTDV operation to estimate standard deviation of variable over time
4	MINX	minimum observed value
5	MAXX	maximum observed value

Biography

Rick Sell was born in Allentown, PA on July 23, 1952 to Douglas and Marjorie Sell of Emmaus, PA. After graduating from Emmaus High School in 1970, Rick entered Pennsylvania State University where he majored in Biology. In May, 1974 he received a Bachelor of Science degree from PSU and graduated with Distinction.

Upon graduating from PSU, Rick joined Air Products and Chemicals, Inc. of Trexlertown, PA as a microbiologist in their wastewater treatment research laboratory. Over the years he advanced to the title of Research Biologist and assumed responsibilities as a supervisor of the wastewater laboratory and pilot plant program.

Rick is married and currently resides in Allentown. He and his wife, Kathryn, have one child, Rebecca, who is almost 3 years old.

It was during his experience at Air Products that Rick developed an interest in working with computers.

After completing several programming courses offered at area colleges, he entered Lehigh University in the spring of 1982 as a part-time candidate for an Master of Science degree in the Industrial Engineering program. Rick's particular area of interest is information systems. After a series of job assignments interrupted his studies, Rick decided to take an educational leave of absense from Air Products to pursue the degree on a full-time basis. That endeavor began in August 1983 and continues to the present.