6-10-1991

An Introduction to the RESearch Queueing Package for Modeling Computer Systems and Communication Networks

Robert F. Gordon Ph.D.
Molloy College, rfgordon@molloy.edu

Edward A. MacNair

RA 208 (#75028) 6/10/91
Computer Science 71 pages

Research Report

An Introduction to the RESearch Queueing Package for Modeling Computer Systems and Communication Networks

E. A. MacNair and R. F. Gordon

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

An Introduction to The RESearch Queueing Package for Modeling Computer Systems and Communication Networks

E.A. MacNair and R.F. Gordon
Dept.581, Loc.Hawthorne
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract: A queueing network is an important tool for modeling systems where performance is principally affected by contention for resources. Such systems include computer systems, communication networks and manufacturing lines. In order to effectively use queuing networks as performance models, appropriate software is necessary for definition of the networks to be solved, for solution of the networks and for examination of the performance measures obtained.

The RESearch Queueing Package (RESQ) and the RESearch Queueing Package Modeling Environment (RESQME) form a system for constructing, solving and analyzing extended queueing network models. We refer to the class of RESQ networks as "extended" because of characteristics which allow effective representation of system detail. RESQ incorporates a high level language to concisely describe the structure of the model and to specify constraints on the solution. A main feature of the language is the capability to describe models in a hierarchical fashion, allowing an analyst to define submodels to be used analogously to use of macros in programming languages. RESQ also provides a variety of methods for estimating the accuracy of simulation results and for determining simulation run lengths. RESQME is a graphical interface for RESQ.

In this introduction, we limit our examples to computer systems and communication networks.

Acknowledgement: The authors wish to thank their co-developers of RESQME: Jim Kurose and Kurt Gordon. We also want to thank Ben Antanaitis, Howard Jachter, Jack Servier, Daniel Souday and Peter Welch for their many suggestions which helped improve the RESQME package and Anil Aggarwal, Al Blum, Gary Burkland, Rocky Chang, Janet Chen, Diana Coles, Prakash Deka, Paul Loewner, and Geoff Parker for their work in implementing RESQME. We would also like to thank our users for their ideas and feedback that we tried to incorporate in RESQ and RESQME. We remain indebted to Charlie Sauer for his design, guidance, inspiration, and development of the RESQ language.

Preface

A queueing network is an important tool for modeling systems where performance is principally affected by contention for resources. Such systems include computer systems, communication networks and manufacturing lines. The RESearch Queueing Package (RESQ) is a system for constructing and solving queueing network models. The RESearch Queueing Package Modeling Environment (RESQME) is a graphical interface for RESQ. Simulation methods, including state of the art statistical analysis, are provided for the full class of queueing networks allowed in the RESQ language. Numerical methods are provided for a subset of the queueing networks allowed by the RESQ language.

This document introduces usage of RESQ and gives examples of simple models of computer systems and communication networks constructed and solved using RESQ. A companion introduction for modeling manufacturing systems is E.A. MacNair and R.F. Gordon, "An Introduction to the RESearch Queueing Package for Modeling Manufacturing Systems," IBM Research Report RA-206, Yorktown Heights, New York, November 1990. The RESQ user should also be familiar with

> C.H. Sauer, A.M. Blum, P.G. Loewner, E.A. MacNair and J.F. Kurose, "The Research Queueing Package Version 2: CMS Reference Manual," IBM Research Report RA-139, Yorktown Heights, New York, November 1986.

The RESQME user should also be familiar with

> R.F. Gordon, P.G. Loewner, E.A. MacNair, K.J. Gordon and J.F. Kurose, "The RESearch Queueing Package Modeling Environment (RESQME) OS/2 Guide for Version 4.1," IBM Research Report, Yorktown Heights, New York, April 1991.

This document has the following sections:

- "Section 1: Introduction" introduces some of the features and capabilities of RESQ.
- "Section 2: A Simple Computer System Model" illustrates the construction and solution of a simple model.
- "Section 3: A Starter Set of Modeling Elements" presents a simple subset of the RESQ modeling elements.
- "Section 4: Advanced Set of Modeling Elements" discusses a more advanced subset of the modeling elements.
- "Section 5: Submodels and Invocations" describes how to structure the model hierarchically.
- "Section 6: Distributions of Queueing Time, Queue Length, and Tokens" illustrates how to obtain distributions of performance measures.
- "Section 7: Confidence Interval Methods" discusses the three methods available in RESQ for statistical analysis of simulation results and automated control of run length.
- "Section 8: Trace and Animation" presents information related to trace output and animation.
- "Section 9: Performance Measures" describes the performance measures available from RESQ evaluations.
- "Section 10: Some Computer and Communication Related Submodels" is a collection of useful submodels of some types of resources found in computer systems and communication networks.
- "Section 11: The RESearch Queueing Package Modeling Environment (RESQME)" is an introduction to the graphical interface.

Table of Contents

1. Introduction 1
2. A Simple Computer System Model 5
3. A Starter Set of Modeling Elements 9
   3.1. Numeric Parameters and Identifiers 9
   3.2. Jobs and Job Attributes 9
   3.3. Simple Service Centers 9
   3.4. Set Nodes 11
   3.5. Simple Passive Resources 11
   3.6. Job Flow 12
   3.7. Sources and Sinks 12
   3.8. Global Variables 13
   3.9. Run Length Control 13
   3.10. Evaluation and Output Analysis 14
4. Advanced Set of Modeling Elements 16
   4.1. Distribution Parameters and Identifiers 16
   4.2. Chain and Node Arrays 16
   4.3. Maximum Number of Job and Chain Variables 16
   4.4. Expressions and Probability Distributions 17
      4.4.1. Arithmetic Expressions 17
      4.4.2. Boolean Expressions 18
      4.4.3. Probability Distributions 19
      4.4.4. Status Functions 19
   4.5. Service Centers 20
   4.6. Passive Resources 21
   4.7. Set Nodes with Multiple Assignment Statements 22
   4.8. Job Flow 22
   4.9. Split, Fission and Fusion Nodes 22
      4.9.1. Split Nodes 22
      4.9.2. Fission and Fusion Nodes 24
   4.10. Wait Nodes 25
   4.11. Dummy Nodes 25
5. Submodels and Invocations 26
6. Distributions of Queueing Time, Queue Length, and Tokens 30
7. Confidence Interval Methods 32
   7.1. Independent Replications 32
   7.2. Regenerative Method and Spectral Method 33
      7.2.1. Regenerative Method 34
      7.2.2. Spectral Method 35
8. Trace and Animation 37
9. Performance Measures 42
10. Some Computer and Communication Related Submodels 44
   10.1. Batch Service - BATCHPRO 44
      10.1.1. Description 44
      10.1.2. Numeric Parameters 44
      10.1.3. Distribution Parameters 44
      10.1.4. Chain Parameters 44
      10.1.5. Global Variables 44
      10.1.6. Submodel Listing and Diagram 44
      10.1.7. Model Which Invokes BATCHPRO 46
   10.2. Bulk Arrivals - BULKARR 46
      10.2.1. Description 46
      10.2.2. Numeric Parameters 46

10.2.3. Chain Parameters 47
10.2.4. Global Variables 47
10.2.5. Submodel Listing and Diagram 47
10.2.6. Model Which Invokes BULKARR 48
10.3. Conditional Arrivals - CARRSUB 49
10.3.1. Description 49
10.3.2. Numeric Parameters 49
10.3.3. Node Parameters 49
10.3.4. Chain Parameters 49
10.3.5. Chain Variables 49
10.3.6. Initial State 49
10.3.7. Submodel listing and Diagram 49
10.3.8. Model Which Invokes CARRSUB 50
10.4. Failures - FAIL1 51
10.4.1. Description 51
10.4.2. Distribution Parameters 51
10.4.3. Chain Parameters 51
10.4.4. Initial State 51
10.4.5. Submodel Listing and Diagram 51
10.4.6. Model Which Invokes FAIL1 52
10.5. Finite Capacity Buffers - FINBUFW 52
10.5.1. Description 52
10.5.2. Numeric Parameters 53
10.5.3. Distribution Parameters 53
10.5.4. Chain Parameters 53
10.5.5. Global Variables 53
10.5.6. Submodel Listing and Diagram 53
10.5.7. Model Which Invokes FINBUFW 54
10.6. Round Robin Scheduling - RRQUEUE 55
10.6.1. Description 55
10.6.2. Numeric Parameters 55
10.6.3. Chain Parameters 55
10.6.4. Job Variables 55
10.6.5. Submodel Listing and Diagram 55
10.6.6. Model Which Invokes RRQUEUE 56
10.7. Cyclic Server - SUBQ 57
10.7.1. Description 57
10.7.2. Chain Parameters 57
10.7.3. Global Variables 57
10.7.4. Job Variables 57
10.7.6. Submodel Listing and Diagram 57
10.7.7. Model Which Invokes SUBQ 58
10.8. Unlimited Receiving Area - UNLIMSB2 59
10.8.1. Description 59
10.8.2. Chain Parameters 59
10.8.3. Submodel Listing and Diagram 59
10.8.4. Model with example 60
11. The RESearch Queueing Package Modeling Environment (RESQME) 61
Bibliography 64

Index **65**

List of Illustrations

Figure 1. Queueing Network Model 2
Figure 2. Network with Passive Queue 3
Figure 3. Results for Simulation Model 15
Figure 4. Service Center with breakdowns 21
Figure 5. Bulk arrivals 23
Figure 6. Packetize and Reassemble Messages 24
Figure 7. Network with Submodel Invocations 26
Figure 8. Host Submodel 27
Figure 9. Queueing Time Distribution 31
Figure 10. Queue Length Distribution 31
Figure 11. Initial Animation Display 40
Figure 12. BATCHPRO submodel 45
Figure 13. BULKARR submodel 47
Figure 14. CARRSUB submodel 50
Figure 15. FAIL1 submodel 52
Figure 16. FINBUFW submodel 54
Figure 17. RRQUEUE submodel 56
Figure 18. SUBQ submodel 58
Figure 19. UNLIMSB2 submodel 60
Figure 20. Assembly Line 63

1. Introduction

Models are used to estimate the performance of systems when measurement of system performance is impossible (e.g., because the system is not yet operational) or impractical (e.g., because of the human and machine resources required). Queueing networks have become important as performance models of a variety of systems where system performance is usually significantly affected by contention for resources. Queueing network models can be used from the early design stages of a system or throughout the life of the system to estimate system performance and to evaluate alternatives.

We will not attempt a general discussion of queueing networks here, but will try to make our discussion self-contained. The reader seeking additional background may wish to refer to some of the books and papers listed in the Bibliography. Our examples will be of queueing network models of computer systems and communication networks, however, queueing models have been used for decades in examining a wide variety of other systems. Much of our discussion applies directly to performance issues in manufacturing lines, office systems, etc.. Our emphasis will be on performance, but the modeling techniques we present also apply to analysis of other issues such as reliability and correctness (e.g., deadlock analysis).

The objective often is to determine how to improve a system. The basic steps in using queueing network models to solve such a problem are (1) determine the resources and their characteristics which will most affect performance, (2) formulate a model representing these resources and characteristics and (3) determine (algebraically, numerically or by simulation) the resulting values for performance measures (e.g., average queue length at a resource) in the model. Typically, the modeler will iteratively go through these steps to examine an existing or proposed system, revising the assumptions and the model until satisfied the results meet the objective.

The first of these steps, though often difficult, is highly system specific. We will not address this problem directly. The RESearch Queueing package (RESQ) is a software **tool** for building queueing network models. We emphasize "tool" because RESQ is not a model in itself. As a tool, it can be of great value in dealing with the second and third basic steps just cited and in the iterative process involving examination of alternatives.

This document introduces some of the features of RESQ and their usage. For a thorough discussion of RESQ, see "The Research Queueing Package Version 2: CMS Reference Manual".

Figure 1 on page 2 illustrates a very simple queueing network of a computer system. The model considers contention for three resources of the system, a cpu and two I/O devices. Users' commands upon the system are represented by **jobs** in the queueing network. The queueing network model which represents the system consists of a collection of nodes corresponding to the resources in the system that the jobs visit. A user spends part of his or her time thinking at the terminal and keying in commands. The time it takes to perform this think time is represented by the service time of a job at the terminal node. The model assumes there are as many terminals as users, so there is no waiting for a terminal. We will still refer to the model representation of the terminals as a queue. After keying in a command, the user waits for a response. The job representing the user command alternates between computation and I/O activities until the command processing is finished and the user receives the response. In this simple model, the routing decisions will be made by specifying probabilities. The user then begins another think time.

We have made this model simple because it is our first example and so that a numerical solution of the model will be feasible. For an exact numerical solution to be feasible, we must make a number of assumptions. One of these assumptions is that multiple resources are not required simultaneously by a job. This may not be the case in a system where memory may be required before the jobs can begin computation. A second assumption is that scheduling is limited to a fairly restricted set of algorithms. In particular, priority scheduling is excluded. A third assumption is that routing decisions are based on probabilities. Other restrictive assumptions will be considered as we discuss and expand upon this model below.

Without RESQ one would likely have two choices with regard to this model and these assumptions: (1) Accept the model and its results without knowing how much impact the simplifying assump-

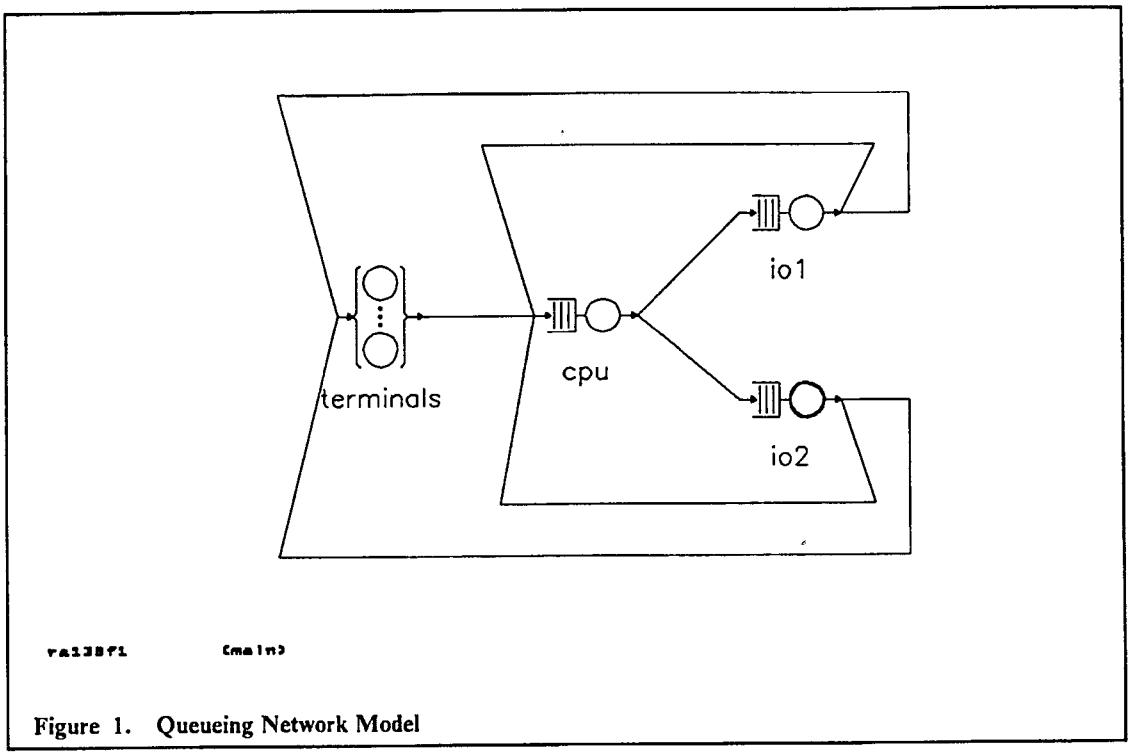

Figure 1. Queueing Network Model

tions have on the results. (2) Reject the model and build a detailed simulation model in a conventional discrete event simulation language. This second choice would entail new problems, most notably (a) expense of building and running the model and (b) doubt about the accuracy of the results (due to the statistical variability of simulation). In the past there has been very little middle ground between these choices.

Two of the principle objectives of RESQ have been (1) to bridge this gap between numerical and simulation methods and (2) to encourage analysts to use a solution method appropriate to the case at hand. RESQ has succeeded at these objectives partly because of the solution methods it provides and partly because of its characterizations of queueing networks. RESQ is effective because of its solution methods, because of its characterizations of queueing networks, and because of its user interfaces, which have been engineered to maximize user productivity.

RESQ provides the "state of the art" in numerical solution methods, so that restrictive assumptions can be avoided where possible. RESQ provides simulation solutions with special features not found in many simulation languages. Most important of these are statistical output analysis techniques which provide error estimates (in the form of confidence intervals) for simulation results and stopping rules for determining when the simulation should end. (Statistical output analysis techniques are discussed in Chapter 7 of Sauer and Chandy 1981, Chapter 4 of Kobayashi 1978 and Chapter 6 of Lavenberg 1982.) The presence of multiple solution methods in one tool makes it possible to use the method most appropriate to a given model and to test the impact of model assumptions such as the ones discussed above. As the model becomes more complex, the modeler can change solution methods. The presence of multiple solution methods also makes feasible the use of several methods in a hybrid solution of one model.

We refer to the networks of RESQ as "extended" because of characteristics absent from most queueing models. Perhaps the most important of the extensions is the "passive" resource, which allows convenient representation of simultaneous resource possession as in the discussion above. Traditional service centers are called "active" queues in the RESQ terminology. A job's activity is typically focused on the resources of active queues. A job typically has no interaction with other model elements while at an active queue. A job typically acquires units of a passive resource and holds onto them while visiting other queues (either active or passive) and model elements. The job

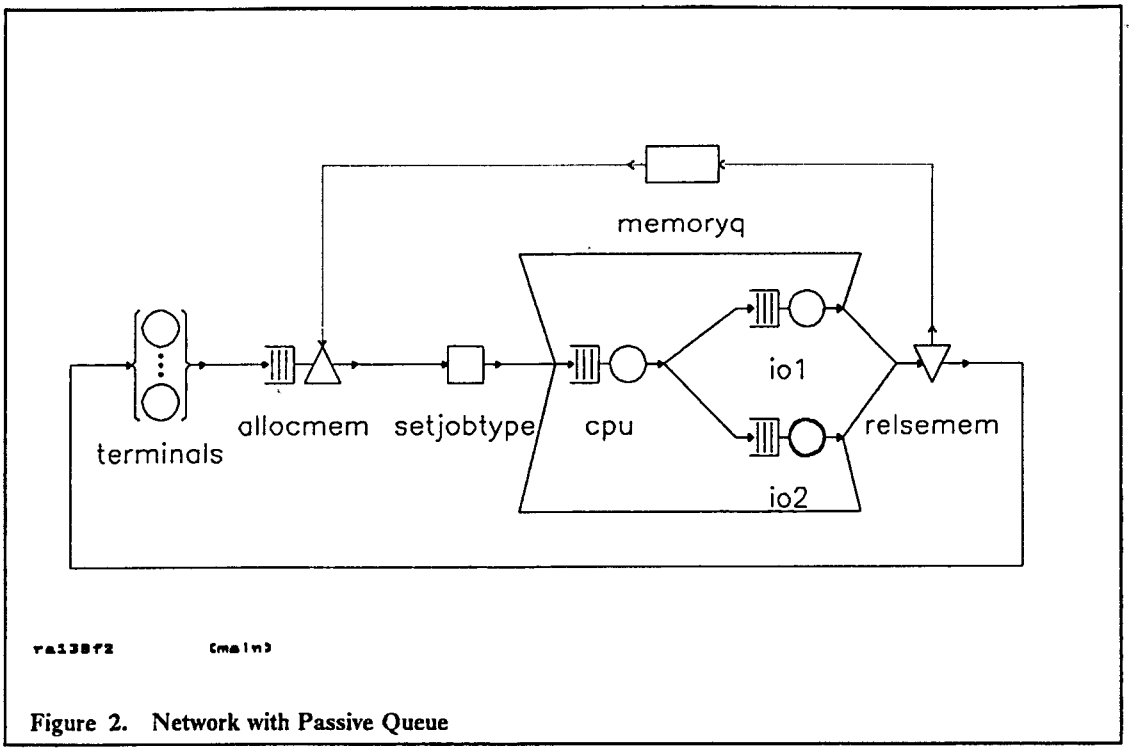

Figure 2. Network with Passive Queue

explicitly releases the units of resource when it no longer needs them. Figure 2 on page 3 shows the addition of a passive queue representing memory to the network of Figure 1. Inclusion of the passive queue allows us to avoid making the assumption that command processing does not require more than one resource at a time. This assumption was made with the model of Figure 1 on page 2 to make the numerical solution feasible; relaxation of this assumption precludes an exact numerical solution. Models with passive queues are solved either by simulation or by approximate numerical methods. (Exact numerical solution for networks with passive queues is possible, in principle, but usually not practical.)

Note that in the figure the passive resource is held by the job during cpu processing and I/O activity. Additional passive queues could be added to the model to represent the channel and /or controller contention. As well as representing simultaneous resource possession, passive queues are particularly useful for representing complex control mechanisms in a simple manner. For example, contention for a channel may cause I/O devices to experience extra revolutions prior to transfer. Communication network protocols and algorithms are additional examples of mechanisms conveniently represented by passive queues. A third use of passive queues is in measuring response times in subnetworks. The "queueing time" (response time) for a passive queue is defined as the time between a job's request for units of the passive resource and that jobs' freeing of the units of resource. Thus in Figure 2 the queueing time for the passive queue corresponds to the response time seen by the terminal users.

The recommended user interface for RESQ is the RESearch Queueing Modeling Environment (RESQME) graphical interface. On one hand, it serves to educate new users. Context-sensitive help is available throughout the modeling process, and a dynamic tutorial facility can be used to both illustrate all of the modeling elements and to build models. Interactive error checking assists the user while building the model. On the other hand, it is designed to accommodate sophisticated users. It gives the user access to all the functionality of RESQ to handle large, realistic models.

RESQME provides the modeler with complete control of the experimental process of creating/editing a model, executing the model, and examining the results. The modeler can iteratively and interactively move through these three tasks in RESQME until he or she solves the problem.

The user interacts with RESQME on two levels. The main level is graphical: the direct manipulation of icons to form a picture of the network, the control of the tasks through menu selections, and the viewing of performance measure charts and animation. Underlying this level is attribute information associated with each of the visual elements. The attribute information may consist of the name, solution method, parameters for the model being displayed; the name, queueing discipline, service time for an active queue in the network; or the colors and scaling for an output chart. This textual information is provided in attribute pop-up windows consisting of templates of prompts and default replies. The specific template depends on the graphical object being defined, and the prompts change dynamically with the user's replies. The interactive syntax checker provides for immediate correction of syntactic errors for this textual input.

In the Create/Edit task, portions of the model can be constructed and corrected in any desired order, and the graphics picture along with the textual attributes of the model is saved for future reference. Models may be defined with parameters so that solutions of several related versions of the model may be obtained in a single evaluation, without changing the model. A model (or submodel) with a generalized set of parameters may be conveniently used by people other than the model developer to study the performance of the system without having to become experts in RESQ. In addition to creating and editing the model graphically, RESQME allows the user to enter values for model parameters, execute the model, produce plots of results, and run animations of the model to observe jobs flowing through the model.

Our examples will display model diagrams, textual information from attribute windows, formatted results, and charts of performance measures produced by RESQME. The model diagrams in Figure 1 on page 2 and Figure 2 on page 3 were produced by RESQME.

2. A Simple Computer System Model

In this section we will consider the numerical solution of the example in Figure 1 on page 2. In doing so we will have to make further assumptions in addition to the ones already discussed.

Let us consider the cpu service center. Notice the icon used in the figure to represent this type of service center. It depicts a waiting line attached to a circle representing a server. We assume the queueing discipline for the cpu is Processor Sharing (PS). (We will usually refer to scheduling algorithms as queueing disciplines.) The PS discipline is defined as the limiting case of a Round-Robin discipline with no overhead as the quantum (time slice) goes to zero. With PS and n jobs in the queue, each job gets $1/n^{th}$ of the server, i.e., the server is shared equally among all of the jobs in the queue. Further, we assume that a job's service time at the cpu has an exponential distribution independent of the current state of the cpu. The following is a possible RESQME description of the cpu service center:

```
QUEUE: cpuq
TYPE: PS
CLASS LIST: cpu
   WORK DEMANDS: .05
```

In this definition we use upper case for RESQME prompts and for responses selected from a list of possible choices and lower case for user typed information. The prompts are terminated by a colon (":").

A queue in RESQ is a definition of a service center or a passive resource. The first prompt is asking for the name of the queue. We use the name "cpuq" for the name of the entire service center. The name we use may be any legal RESQ unused identifier (see Appendix 2 of the Reference Manual). (Legal names must start with an alphabetic character and be no more than ten characters.)

The second prompt is asking for the type of queue. The type specified may be a general type, i.e., "ACTIVE" or "PASSIVE," or a specialized type, e.g., "PS" as in the example. A general type allows specification of all queue characteristics, while a specialized type assumes certain default specifications and thus allows an abbreviated queue definition. The specialized type PS results in a single server active queue with the Processor Sharing queueing discipline. Further, the server is assumed to have a fixed service rate of one (1). The service time at a queue is equal to the work demanded divided by the service rate. The service rate is the rate at which the server performs the work. (If we want the server to have a different fixed rate, we can divide the mean service time by that rate. If we want to explicitly define server rates, we must use the general ACTIVE type of queue. The general ACTIVE type of queue is described in Section 4 of the Reference Manual.) We will defer until later discussion of some of the other characteristics assumed by the specialized PS type. Generally, the assumptions result in a simpler specification than might otherwise be made.

The third prompt is asking for a (job) "class" at the queue. In general, an active queue may have many classes. The classes of a queue serve as "nodes" in the routing description of the network; having multiple classes at a queue allows specification of different routing paths for different types of jobs. A node in the model represents a place a job visits. The routing represents the paths the jobs follow as they flow through the resources in the system. Different classes at a queue may also have different service requirements, priorities and other characteristics we will describe later. In this case there is only one class, to which we give the name "cpu." (A class name may be any legal identifier.)

The fourth prompt is for the service time distribution of jobs at the queue. Remember that the service time is the work demand divided by the service rate, and that the service rate for a PS queue is one. Therefore, the service time is equal to the work demand in this case. The service time of .05 is taken as an exponential distribution with mean .05.

In this example the values given for times are in units of seconds. This is only implicitly defined, however. As far as RESQ is concerned, the meaning of the time unit is up to the user; the numerical values produced by RESQ would be the same whether we intended the time units to be microseconds, seconds or hours. It is up to the user to decide upon the appropriate time unit for

the problem (e.g., the user may choose the time units to be seconds) and to consistently provide all input and to interpret all RESQ output times in the same units.

The definitions for two I/O devices are similar, the differences being in the names, the queueing discipline (First-Come-First-Served abbreviated FCFS), and the mean service times.

```
QUEUE: io1q
TYPE: FCFS
CLASS LIST: io1
   WORK DEMANDS: .22

QUEUE: io2q
TYPE: FCFS
CLASS LIST: io2
   WORK DEMANDS: .019
```

The final queue definition is similar to the other definitions except that we use the "IS" special type of queue, which gives an active queue with an infinite server discipline. This type of queue always has a server available for each job in the queue. There are never any jobs waiting for service. Notice that we use a different icon to represent the IS queue. The dots represent an infinite number of servers.

```
QUEUE: terminalsq
TYPE: IS
CLASS LIST: terminals
   WORK DEMANDS: 5
```

This concludes the queue definitions for this model.

The other principal part of the model definition is the routing. The routing is defined in terms of transitions between nodes; in this model the only nodes are the classes. In general, nodes of a model may be partitioned into "chains" to efficiently represent different types of jobs with different service times, such that a job at a node in one chain can never get to a node in another chain. Chains are disconnected paths that the jobs follow. In this model there is only one chain. The following is the routing definition for this model:

```
CHAIN: interactiv
   TYPE: CLOSED
   POPULATION: 15 /* users */
   : terminals -> cpu ;1.0
   : cpu -> io1 ;.10
   : cpu -> io2 ;.90
   : io1 -> cpu ;1-1/cpiocycles
   : io1 -> terminals ;1/cpiocycles
   : io2 -> cpu ;1-1/cpiocycles
   : io2 -> terminals ;1/cpiocycles
```

The first prompt is for the name of the chain. The second prompt is for the type of chain, "open" or "closed." Open chains have "sources" of jobs to enter the model and "sinks" for jobs to leave the model. Closed chains do not have sources or sinks. Usually the number of jobs in a closed chain is fixed, though there are possible exceptions to this rule. The chain for this model is closed. The third prompt is for the number of jobs in the closed chain, i.e., its "population." The population of fifteen represents the number of active users in the model.

The remainder of the routing lines starting with a colon (":") are for the routing transitions, i.e., descriptions of where a job can go when it leaves a node and how it decides where to go. The first routing transition means that a job leaving the terminals node always goes to the cpu node (with probability 1.0). The next two lines indicate that 10% of the jobs leaving cpu go to io1 and 90% go to io2. From each I/O device there is a probability of going back to the cpu and 1 minus this probability of going to the terminals. This probability is calculated using a numeric identifier which will be defined shortly.

The only other part of the model that we have not looked at is the model header.

```
MODEL: csmt
   METHOD: NUMERICAL
   NUMERIC PARAMETERS:
   DISTRIBUTION PARAMETERS:
   NUMERIC IDENTIFIERS: cpiocycles
   CPIOCYCLES: 8
   DISTRIBUTION IDENTIFIERS:
   CHAIN ARRAYS:
   NODE ARRAYS:
```

The model name can be any legal RESQ name. We are using the NUMERICAL solution method to solve this model. Parameters, identifiers, and arrays are optional modeling elements which will be discussed in detail later. For now, we define an identifier called cpiocycles which represents the average number of times a job cycles through the cpu and I/O subsystem. This identifier is assign a value of eight.

Since this is a model we are solving numerically, the above definitions complete all the information necessary for this model. At this point in the model lifecycle, we would do a global consistency check to make sure the model is complete and correct. Our next step is to evaluate the model. Since we have no parameters in the model, we can proceed right to the execution. (The current workstation version of RESQME does not permit the numerical solution to be computed. Until this restriction is removed, this model must be solved on the host using the PL/I version of RESQ, or changed to use the simulation solution.) The following are all of the formatted results produced for this model.

```
ELEMENT        UTILIZATION
CPUQ           0.86234
IO1Q           0.37943
IO2Q           0.29492
TERMINALSQ     0.00000

ELEMENT        THROUGHPUT
CPUQ           17.24686
IO1Q           1.72469
IO2Q           15.52218
TERMINALSQ     2.15586

ELEMENT        MEAN QUEUE LENGTH
CPUQ           3.22555
IO1Q           0.58712
IO2Q           0.40804
TERMINALSQ     10.77929

ELEMENT        MEAN QUEUEING TIME
CPUQ           0.18702
IO1Q           0.34042
IO2Q           0.02629
TERMINALSQ     5.00000
```

The utilization is the fraction of time a server is busy. The utilization for an infinite server is always reported as zero. The cpu has the highest utilization and is therefore the bottleneck.

The throughput is the number of jobs that completed per unit time. The routing probabilities and the service times determine the throughputs. The throughput at the cpu is the highest because of all the jobs that cycle back to the cpu.

The mean queue length includes the jobs waiting and in service. If we add up all of the queue lengths we obtain the 15 jobs in the closed chain population.

The mean queueing time also includes the time the jobs wait and are in service. We can obtain estimates of the waiting times by subtracting the mean service times from the mean queueing times. We can also obtain an estimate of the mean response time. (We use "estimate" here to emphasize that we are dealing with a model and usually do not obtain the values for the actual system. For

this model RESQ provides exact values for the performance measures within the limits of numerical error.) By response time we mean the time that begins after a job leaves the terminals until it returns to the terminals. This performance measure in a closed chain is not calculated by RESQ when using the numerical solution. Later we will see how we can obtain it in a simulation model.

The response time is simple to calculate using Little's Rule: mean number of jobs = throughput x mean response time. The mean number of jobs in the computer system (not at the terminals) is 15 - 10.77929 = 4.22071, so the mean response time is 4.22071 / 2.15586 = 1.958 seconds.

3. A Starter Set of Modeling Elements

This section will discuss a subset of the RESQ modeling elements which are sufficient for new users to get started with for building simple models. The more advanced modeling elements are discussed in Section 4. We will take the model we solved in Section 2 and modify it. The solution method will be changed from numerical to simulation and some other modeling elements will be added.

3.1. Numeric Parameters and Identifiers

Suppose we wish to evaluate the model for different think times, number of users, and number of memory partitions. Numeric parameters, which are defined in the model header, are a convenient way to do this. We will define three numeric parameters called thinktime, users, and partitions and use them during the remainder of the model definition.

As in Section 2, we define a numeric identifier called cpiocycles which will be used to calculate a branching probability. We also want to define a symbolic name called jobtype which we will use to improve the readability of the model. To do this we will define another numeric identifier in the model header. The following prompts show the revised model header with the model name changed to csmtm and the solution method to SIMULATION.

```
MODEL: csmtm
   METHOD: SIMULATION
   NUMERIC PARAMETERS: thinktime users partitions
   DISTRIBUTION PARAMETERS:
   NUMERIC IDENTIFIERS: cpiocycles jobtype
      CPIOCYCLES:8
      JOBTYPE: 0
   DISTRIBUTION IDENTIFIERS:
   GLOBAL VARIABLES:
   CHAIN ARRAYS:
   NODE ARRAYS:
   MAX JV:
   MAX CV:
```

Distribution parameters, distribution identifiers, chain and node arrays, and max jv and cv will be explained in Section 4. Global variables will be discussed in Section 3.8.

Numeric parameters and numeric identifiers are symbolic names that can be used in the model definition. They are assigned arithmetic expressions. Parameters receive their values when the model is executed. Identifiers are assigned values immediately after they are defined. In the above example, the numeric identifier cpiocycles is assigned the value eight (8) and jobtype is assigned the value zero (0).

3.2. Jobs and Job Attributes

Recall that the jobs represent commands to be processed which flow through the model and visit the nodes which depict the system resources. Every job has its own set of job attributes. The job attributes are stored in job variables identified by the keyword jv. For each job, jv is a vector which is subscripted to address the appropriate attribute. The jv subscripts start with index zero (0). In this example we will define a job variable to represent the type of job. We will have two types of jobs in the model. We will use jv(0) to represent the job type. Since we do not want to remember that the zeroth job variable is the job type, we will use the numeric identifier previously defined to subscript the job variable, jv(jobtype). Job variables can be used to store any type of information that the user wants related to a job, and their values can be used throughout the model, for example here we will use them to make routing decisions.

3.3. Simple Service Centers

Service centers have three components: one or more servers actively engaged in taking time to provide service to jobs, a queueing discipline or scheduling algorithm to decide which job to put into service next, and one or more classes of jobs to be served. Service centers are called active

queues in RESQ. They can be used to represent many different types of processing encountered in a computer system or a communication network, e.g., cpu, I/O device, bus, network link, terminals, and many more.

Of the three components, the purpose of the servers and the queueing disciplines is straight forward. Let us briefly discuss the meaning of classes. A class is normally associated with a local waiting line at a queue. The class represents a particular type of job that is being served at the queue. The name of the class is a node in the routing, so different types of jobs can follow different paths. Each class can have its own service time distribution, so different types of jobs can receive different amounts of service. Classes can also have different priority levels. Different work loads can be easily represented by different classes. Alternatively, transactions at different stages of service can be represented by different classes when they revisit service centers.

In this section we will only discuss First-Come-First-Served (FCFS), Infinite Server (IS), and ACTIVE queues. The ACTIVE queues discussed here will be a subset of more complicated versions. Queues with other types of queueing disciplines, more complicated ACTIVE queues, and queues with multiple classes will be discussed in Section 4.

The queues for the I/O devices that we discussed in Section 2 are all simple FCFS queues. A FCFS queue has a single server with a service rate of one. Recall that in this case the service time is equal to the work demand. The queue definition for io1 is repeated here for easy reference.

```
QUEUE: iolq
TYPE: FCFS
CLASS LIST: io1
   WORK DEMANDS: .22
```

Every queue has a unique name. The type selected from the list of types is FCFS. The name of the class is io1, and the service time distribution is exponential with a mean of .22 time units. The exponential distribution is the default distribution. Other probability distributions will be discussed in Section 4. The other FCFS queue definition, plus the Processor Sharing queue for the cpu, for the simulation version of the model in Figure 2 on page 3 are the same as the ones that were displayed in Section 2.

The other simple type of queue that we find in this model is the IS queue for the terminals. An IS queue has no waiting line. As soon as a job arrives at the queue it is placed in service. Therefore, it can be used to represent a pure delay (advance time) with no waiting. IS queues still have classes to distinguish different types of jobs. The only difference in the specification between the FCFS and IS queue definitions is the queue type as shown again in the following queue definition. Notice in the following example that we have used the numeric parameter called thinktime to be the service time.

```
QUEUE: terminalsq
TYPE: IS
CLASS LIST: terminals
   WORK DEMANDS: thinktime
```

The ACTIVE queue type can also be used in constructing simple models. It has a prompt for the number of servers. When the number of servers is greater than one, this type of queue is referred to as a multi-server queue. The model in Figure 2 on page 3 does not contain a multi-server queue. We will therefore give an example of the use of a multi-server queue by showing how it can represent a multiprocessor. Here we are assuming that 2 jobs can be processed simultaneously. The queueing discipline (DSPL:) can also be specified. It is selected from a list of several queueing disciplines. The other possible choices, besides FCFS, will be discussed in Section 4. ACTIVE queues also have classes and work demands similar to FCFS queues. A difference is that we can specify information related to the servers. In this section, we will assume all servers are identical. Therefore only one server definition is necessary. Here we are only interested in the service rate (RATES:) information. Recall that the service time of a job at a queue is equal to the work demand divided by the service rate. With the ACTIVE queue, we can specify the work demand and the service rate separately. This type of server is called a fixed rate server. A fixed rate server serves with the same rate independent of the number of customers at the queue. Queue dependent servers,

which change their service rate, and servers who select jobs from specified classes (ACCEPTS:) will be discussed in Section 4.

```
QUEUE: multiprocq
TYPE: ACTIVE
SERVERS: 2
DSPL: FCFS
CLASS LIST: cpu
   WORK DEMANDS: .005
SERVER-
   RATES: .9
   ACCEPTS: all
```

3.4. Set Nodes

A set node is a modeling element used to execute assignment statements. Whenever a job visits a set node, the assignment statements associated with that set node will be executed. In Figure 2 on page 3 the rectangular box with the name setjobtype illustrates the icon used for a set node. In this section we will only discuss set nodes with a single assignment statement. Set nodes with multiple assignment statements will be discussed in Section 4. The set node named setjobtype will be used to assign the type of job to a job variable so that later on a routing decision can be based on the job type. The following set node definition shows the syntax associated with a set node. Every set node has a unique name attached to it. The assignment list is used to specify the assignment statement. Recall that we previously defined a numeric identifier called jobtype to have a value of 0. In this case we are assigning to job variable jv(jobtype) a sample from a discrete probability distribution. This particular discrete distribution has two values, 1 and 2. The probability associated with value 1 is 0.1 and the probability associated to value 2 is 0.9.

```
SET NODE: setjobtype
   ASSIGNMENT LIST: jv(jobtype)=discrete(1,.1;2,.9)
```

In general, any type of RESQ variable can be assigned values at set nodes. The other types of variables and the types of expressions that can be assigned will be discussed in Section 4.

3.5. Simple Passive Resources

One of the principal limitations of the model in Section 2 is that it ignores simultaneous resource possession, where a job acquires and holds onto elements of a resource while it receives service at a service center. As an example of simultaneous resource possession, we will include a simple representation of memory being allocated to a job before it can undergo any processing in the cpu-I/O subsystem. A passive queue will be used to model memory. A passive queue has a pool of tokens which represent the finite number of elements of the passive resource which are to be allocated to the jobs. It also has one or more places (nodes) where the tokens are allocated and one or more nodes where tokens are released from the jobs and returned to the pool for reallocation. The following queue definition shows an example of a passive queue. A passive queue has a name and the type is PASSIVE. This passive queue uses the numeric parameter partitions to specify the number of tokens. The queueing discipline here is FCFS. In Section 4 we will see examples of other possible queueing disciplines. Allocate nodes provide a place where jobs can allocate tokens if they are available or a place to wait if the number of tokens requested cannot be satisfied. In this case the allocate node is given the name allocmem and we are allocating exactly 1 token (memory partition) to each job as it leaves the allocate node. It is important to remember that jobs hold onto the tokens as they visit other nodes in the model. The release node is given the name relsemem. All tokens associated with this passive queue are released by the job when it visits a release node.

From Figure 2 on page 3 we see that after the think time is completed the job requests a memory partition. If one is available, it is acquired, and the job proceeds to set its job type and then to the cpu. If no memory partitions are available, the job waits at allocmem until one is released. After cycling through the cpu-I/O subsystem several times, the job finishes and releases the memory partition at relsemem and sends a response back to the terminal.

```
QUEUE: memoryq
   TYPE: PASSIVE
   TOKENS: partitions
   DSPL: FCFS
   ALLOCATE NODE LIST: allocmem
      NUMBER OF TOKENS TO ALLOCATE: 1
   RELEASE NODE LIST: relsemem
```

3.6. Job Flow

The flow of jobs through the resources in the system is described in the routing definition. Job routing is given in chain definitions. A chain is a path that certain types of jobs follow. Each chain has a name and a type. The chain name in this example model is interactiv and the type is CLOSED. OPEN chains will be discussed in the next subsection. CLOSED chains have a fixed population of jobs that continue to circulate through the nodes of the chain. In the following chain definition, the population is specified by using the numeric parameter called users which we previously defined in the header. Using a numeric parameter here will allow us to evaluate the model for several different number of users without having to change the model. The remaining lines in the chain definition are for the routing transitions. These routing transitions are similar to the ones we discussed in Section 2. The prompt is just a colon, followed by a FROM node name, an "arrow" (->), a TO node name, a semicolon, and a probability or a predicate for determining whether this branch is taken. All of the lines but two in this example use probabilities to make the routing decisions. The two lines from the cpu use a predicate to test the value of job variable jv(jobtype). Type 1 jobs go to io1. Type 2 jobs go to io2.

```
CHAIN: interactiv
   TYPE: CLOSED
   POPULATION: users
   : terminals -> allocmem ;1.0
   : allocmem -> setjobtype ;1.0
   : setjobtype -> cpu ;1.0
   : cpu -> io1 ;if(jv(jobtype)=1)
   : cpu -> io2 ;if(jv(jobtype)=2)
   : io1 -> relsemem ;1/cpiocycles
   : io1 -> setjobtype ;1-1/cpiocycles
   : io2 -> relsemem ;1/cpiocycles
   : io2 -> setjobtype ;1-1/cpiocycles
   : relsemem -> terminals ;1.0
```

This textual version of the routing is created for you automatically by RESQME as you graphically specify the paths the jobs follow. The default predicate to make the routing decision is a probability of 1.0. You can type over this to use other probabilities or conditions to make the routing decision as is illustrated in the above example.

3.7. Sources and Sinks

The other major type of chain is an open chain. The example model we are working with in this section does not contain an open chain. For this reason we will define a simple example of an open chain for illustrative purposes. An open chain has one or more sources where jobs can enter the model and one or more SINKs where jobs can leave the model. Each source is assumed to have an infinite population associated with it from which it selects jobs to enter the model according to an interarrival time distribution (ARRIVAL TIMES:). The following open chain has name jobpath and type OPEN. The name of the source (SOURCE LIST:) is start, and the interarrival time distribution is an exponential distribution with a mean of 1 time unit. The routing transitions have the same format as the ones we discussed in the above subsection.

```
CHAIN: jobpath
   TYPE: OPEN
   SOURCE LIST: start
      ARRIVAL TIMES: 1
   : start -> node1 ;1.0
```

One concept you should keep in mind related to open chains, is that the chain population varies as the model is running. This is not the case with closed chains.

3.8. Global Variables

Global variables are symbolic names which can change value as the run proceeds. They are defined in the model header in a similar way to the definition of identifiers and can then be used throughout the model. Since the example model we are discussing in this section does not contain any global variables, we will describe an example where a global variable might be used. If we are accumulating jobs to form a batch before an operation where we want to process batches of jobs together, we could use a global variable to count the number of jobs in the next batch. The following shows the syntax associated with the definition of a global variable. The name of the global variable is n_in_btch, and its initial value is 0.

```
GLOBAL VARIABLES: n_in_btch
   n_in_btch: 0
```

Every time another job comes to join the next batch we send it to a set node to increment the number in the batch.

```
SET NODE: inc_batch
   ASSIGNMENT LIST: n_in_btch = n_in_btch + 1
```

When the batch is completed, we then put all of the jobs in the batch in the batch process and reset the number in the batch to zero. A submodel illustrating these other details will be discussed in Section 10.

This completes the definition of the major modeling elements necessary to construct simple models of computer systems and communication networks. The next subsection will discuss the information necessary to specify the length of the run.

3.9. Run Length Control

An important concept to remember related to using simulation to solve models is that the results produced by the simulation are random variables. The stochastic nature of the results is based on the random numbers that are used to generate samples from the probability distributions while the simulation is executing.

The simulation run length template begins with a prompt for the confidence interval method. Section 7 will discuss confidence intervals. Here we will discuss running simulations without producing confidence intervals. As shown in the following template, the response of NONE is selected for the confidence interval method. Then there is a section for the initial state definition. This describes where jobs are to be placed initially when the simulation begins. For each chain we specify the initial population and the nodes where we want jobs to start. For chain interactiv initially all of the jobs (users) will be place at the terminals. The prompt for initial portion discarded lets you specified how much of the beginning of the run to throw away. Very often, the beginning of a simulation is not representative of the long run behavior after the simulation reaches steady state, and you can remove this data from the simulation results. Further discussions of these points are made in Section 7. A null response keeps all data for the entire run. The run limits section lets us define several ways that the simulation could terminate. The first of these conditions that is detected will stop the run. In the example, the run will stop on the number of events, where an event is a service completion at an active queue or an arrival at an open chain. The other possible stopping conditions are the simulated time and the number of departures from specified queues or nodes. As an overall limit, we specify 100 cpu seconds for each run. The seed is used to start the random number streams. Leaving it blank uses a default value of 1.

```
CONFIDENCE INTERVAL METHOD: NONE
INITIAL STATE DEFINITION-
CHAIN: interactiv
   NODE LIST: terminals
   INIT POP: users
INITIAL PORTION DISCARDED:
RUN LIMITS-
   SIMULATED TIME:
   EVENTS: 10000
   QUEUES FOR DEPARTURE COUNTS:
      DEPARTURES:
   NODES FOR DEPARTURE COUNTS:
      DEPARTURES:
LIMIT - CP SECONDS: 100
SEED:
```

3.10. Evaluation and Output Analysis

Now we are ready to do the evaluation and to analyze the results. The first step in the Evaluation task is to specify the parameter values. Remember that the model has one parameter representing the number of users. We can examine three scenarios by making three runs of the model with the number of users set at 20, 25, and 30, respectively. The following template shows the specification of the parameter values for the three runs.

```
Model Solution Parameters-
  thinktime: 10
      users: 20
 partitions: 4
Model Solution Parameters-
  thinktime: 10
      users: 25
 partitions: 4
Model Solution Parameters-
  thinktime: 10
      users: 30
 partitions: 4
Specify parameters for another run: NO
```

After executing the model, we can view the performance results and the animation. We will only look at a few of the possible results here. Additional results are discussed in Section 9 and in the Reference Manual. Figure 3 on page 15 shows plots of the utilizations of memory, the cpu and the I/O devices for the three runs plotted against the number of users. The increasing number of users puts an increasing load on memory usage (second curve, solid line) and the cpu (top curve, dashed line). These resources appear to be the bottlenecks in this model.

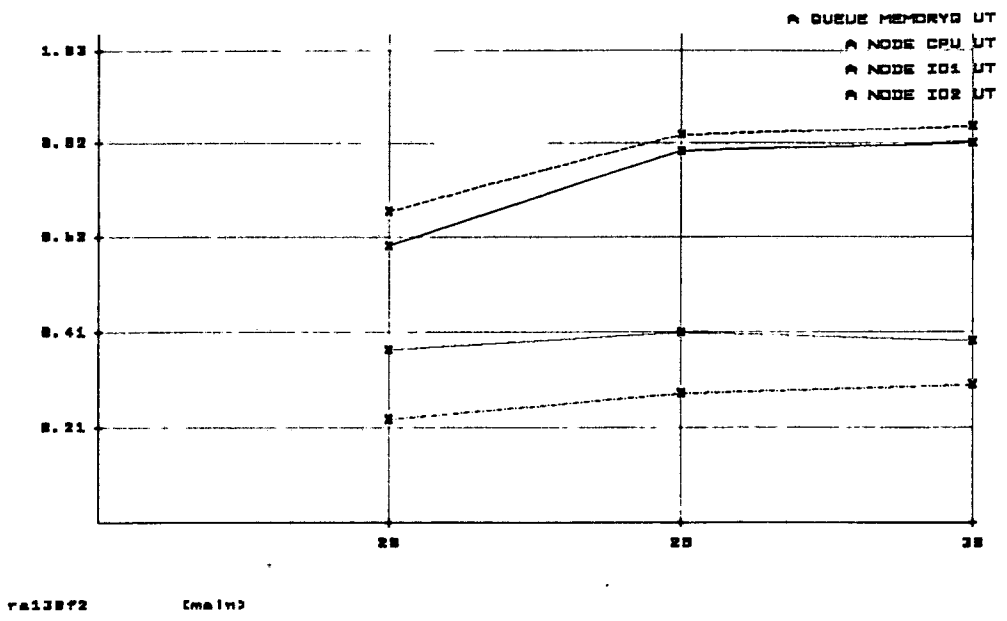

Figure 3. Results for Simulation Model

4. Advanced Set of Modeling Elements

In the previous section, we discussed a relatively simple subset of the available modeling elements for RESQ. This section will discuss the remainder of the modeling elements. A new user should be familiar with the starter set before trying to use these more advanced building blocks.

4.1. Distribution Parameters and Identifiers

Distribution parameters and distribution identifiers are symbolic names to which probability distributions are assigned. Similar to numeric parameters and identifiers, distribution parameters are assigned values at execution time, and distribution identifiers are assigned values in the model header. The values of parameters and identifiers are fixed throughout the simulation run.

Parameters and identifiers can be scalars, vectors, and matrices. The indexing of parameter and identifier arrays begins at one (1). The following example illustrates definitions for these types of symbolic names.

```
NUMERIC PARAMETERS: batchsize
DISTRIBUTION PARAMETERS: mlservtime(3)
NUMERIC IDENTIFIERS: rtgarray(2;3)
   rtgarray: .3 .4 .3 ++
             .2 .5 .3
DISTRIBUTION IDENTIFIERS: srcliat
   srcliat: uniform(1,2,1)
```

In the above example, the numeric parameter batchsize is a scalar parameter, mlservtime is a vector with three elements, and rtgarray is a matrix with two rows and three columns. The values assigned to mlservtime will be probability distributions. The distribution identifier srcliat will be used to sample from a uniform distribution.

4.2. Chain and Node Arrays

There are times when it is convenient to use vectors for names of chains and nodes. Chain and node arrays are defined in the model header. The following shows a simple abstract example of this.

```
NUMERIC PARAMETERS: nc cpop(nc) p2(nc)
CHAIN ARRAYS: chn(nc)
NODE ARRAYS: c1(nc) c2(nc) c3(nc)
...
CHAIN: chn(*)
   TYPE: closed
   POPULATION: cpop(*)
   : c1(*) -> c2(*) ;p2(*)
   : c1(*) -> c3(*) ;1-p2(*)
   : c2(*) -> c1(*)
   : c3(*) -> c1(*)
```

There are 3 numeric parameters. Nc will be used for the number of chains. Cpop is a vector with the chain populations. P2 is a vector with branching probabilities by chain. Chn is a chain array, a symbolic name for nc chains. C1, c2, and c3 are node arrays with their elements in the different chains.

4.3. Maximum Number of Job and Chain Variables

This information is provided in the model header. Each job has a vector of job variables with the number of elements being equal to one plus the maximum number of jv's specified. Each chain has a vector of chain variables with the number of elements being equal to one plus the maximum number of cv's specified.

```
MAX JV:5
MAX CV:1
```

In the above example you can use indices from zero to five to index the jv's and zero to one to index the cv's.

Chain variables are associated with chains. Chain variable zero (cv(0)) has a special meaning. It is used to change the arrival rate of jobs in an open chain. Cv(0) is initially one. When it is set to some value other than one, the arrival rate for that chain will change. Other chain variables do not have a special meaning for RESQ. See the Reference Manual for further information about chain variables.

4.4. Expressions and Probability Distributions

RESQ expressions correspond to those in programming languages, with essentially the same rules as languages such as C, Fortran, Pascal, and PL/I. A formal definition of RESQ expressions is given in the grammar in Appendix 4 of the Reference Manual. We will give an informal discussion in this section. Except for expressions used in routing predicates, any RESQ expression must evaluate to a scalar numeric value, a vector of numeric values or a matrix of numeric values.

"Simulation dependent" expressions are those that depend on job variables, chain variables, global variables, probability distributions, the USER function, status functions or the PRINT function. Simulation dependent expressions can be used most places in the definition of a simulation model. See the Reference Manual for information about when simulation dependent expressions cannot be used.

4.4.1. Arithmetic Expressions

An expression is built from primitive elements called factors. A factor may be an unsigned number,

```
3  45.625  3.2E-10  4.356E+02,
```

a parenthesized signed factor,

```
(-1),
```

an identifier or global variable,

```
tool1  yield(2)  array1(1;*),
```

a job or chain variable,

```
jv(jobtype)  cv(0),
```

a probability distribution,

```
standard(10,2)  discrete(1,.4;3,.6)  uniform(1,3,1),
```

a status function reference,

```
ta(robot)  th(buffer)  ql(machine1)  rj,
```

a parenthesized expression,

```
(4+2),
```

a call to a USER defined function,

```
user(2;6;uniform(0,1,1)),
```

a call to the PRINT function,

```
print(ta(robot)),
```

or a numeric function call,

```
min(1,2)   max(3,alpha)  ceil(10.3)  floor(9.99)
abs(beta)  exp(-3)       ln(exp(-3)).
```

These numeric functions are evaluated using the corresponding functions provided by C.

Using the exp and ln functions appropriately allows you to raise an expression to a power. The following expression can be used to raise some expression x to the power p:

```
exp(p*ln(x))
```

A special case of this is taking the square root of x:

```
exp(0.5*ln(x))
```

In RESQME, numeric values are generally treated as if they were single precision floating point, i.e., they are usually truncated to roughly six decimal digits, even if given more precisely by the user. Global variables, job variables, chain variables and temporaries used in expression evaluation are maintained as double precision floating point during the simulation, i.e., they have roughly 16 decimal digits of precision.

Factors are combined with multiplying operators "*", "/" and "mod" to form terms, e.g.,

```
1*print(w)      alpha mod 2     jv(3)/jv(10)
```

(MOD is the modulo function, performed by the C fmod function.) A single factor may itself be considered a term if it is not to be used in an operation with a multiplying operator.

Terms are combined with adding operators "+" and "-" to form expressions, e.g.,

```
1*print(w)+5      alpha mod 2+1   5-jv(3)/jv(10)
```

Note that the multiplying operators are applied first before the adding operators. A single term may itself be considered an expression if it is not to be used in an operation with a adding operator. As suggested before, expressions may be parenthesized to force adding operators to be used before multiplying operators.

4.4.2. Boolean Expressions

Predicates are used in routing definitions (Sections 3.6 and 4.8) and at wait nodes (Section 4.10). A predicate is a Boolean expression preceded by "if(" and followed by ")". A Boolean expression must evaluate to either T (true) or F (false).

The primitive elements of Boolean expressions are called Boolean factors. A Boolean factor may be a Boolean constant, e.g.,

```
T     F
```

a relational expression, e.g.,

```
v<2  j*k<=v  print(w)>j*k  abs(jv(0)) mod n>=print(w) i=j  w¬=3.4
```

the logical negation of a Boolean factor, e.g.,

```
not v>3
```

or a predicate, e.g.,

```
if(1<=v and v<=10)
```

Note the use of "if" before the parenthesized Boolean expression. The following is incorrect:

```
if((1<=v and v<=10))
```

Boolean factors are combined with the logical "and" operation to form Boolean terms, e.g.,

```
1<=v and v<=10
```

A Boolean factor by itself may be considered a Boolean term if it is not to be used in an "and" operation.

Boolean terms are combined with the logical "or" operation to form Boolean expressions, e.g.,

```
1<=v and v<=10 or ta(windowq)=7
```

Note that the "and" is performed before the "or". If the reverse order is desired, the predicate notation should be used, e.g.,

```
1<=v and if(v<=10 or ta(windowq)=7)
```

A Boolean term by itself may be considered a Boolean expression if it is not to be used in an "or" operation.

Note that all of the above Boolean expressions must be enclosed in "if(" and ")" before they can be used in routing definitions, e.g.,

```
host->link;if(1<=v and if(v<=10 or ta(windowq)=7))
```

4.4.3. Probability Distributions

Often one has little information about random values in the model other than the mean values. For this reason, RESQ uses the exponential distribution as the default distribution. When RESQ expects a probability distribution and the expression given evaluates to a scalar value, the exponential distribution is assumed. It can be specified explicitly with the syntax

```
exponential(mean)
```

If the standard deviation is known, a distribution with two parameters can be used. The standard distribution has a mean and a coefficient of variation (cv). (The cv is defined as the standard deviation divided by the mean.) The syntax is

```
standard(mean,cv)
```

Depending on the value of the cv, RESQ will choose an appropriate probability distribution.

- If the cv is zero, the constant distribution is used.
- If $0 < cv < 0.5$, the uniform distribution is employed.
- If $0.5 <= cv < 1$, a sample is taken from an Erlang distribution.
- If the cv is one, the exponential distribution is the choice.
- If the $cv > 1$, the hyperexponential distribution is used.

You can explicitly request a uniform distribution. The syntax is

```
uniform(l1,u1,p1; ... ;ln,un,pn)
```

where l1 is the lower limit of the first range of values, u1 is the upper limit of the first range of values, p1 is the probability of being in the first range of values, ln is the lower limit of the nth range of values, un is the upper limit of the nth range of values, and pn is the probability of being in the nth range of values.

The discrete distribution has a list of values and their associated probabilities

```
discrete(v1,p1; ... ;vn,pn)
```

When you have a large number of values associated with a discrete distribution, the above syntax can be tedious. In the special case where the values are consecutive integers and have equal probabilities, using the uniform distribution and converting the sample to an integer is easier to specify. For example, if we want integer values between 1 and 100 with equal probabilities we could do the following:

```
ceil(uniform(0,100,1))
```

Probability distributions can be used in arithmetic expressions. For more information related to these distributions and others, see Appendix 3 of the Reference Manual.

4.4.4. Status Functions

There are six functions which may be used in arithmetic and boolean expressions which indicate the current status of the network. These can be used for example to determine to which service center to send a job, perhaps to the service center with the shortest queue length. The functions have an argument specifying a node or queue name. When used in routing predicates, the argument is optional under the circumstances described with a specific function. These functions are:

SA(queue name) - Servers Available. SA returns the number of servers currently available at an active queue, i.e., the number not in use. The queue name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is a class of the queue.

TA(queue name) - Tokens Available. TA returns the number of tokens currently available at a passive queue, i.e., the number not in use. The queue name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is an allocate node of the queue.

TH(queue name) - Tokens Held. TH returns the number of tokens of the specified passive queue held by the job causing the function to be invoked.

QL(node name) - Queue Length. QL returns the current number of jobs (counting both true jobs and job copies) at a class or an allocate node. The node name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is the desired node.

TQ(queue name) - Total Queue. TQ returns the current number of jobs (counting both true jobs and job copies) at a queue. The queue name and parentheses may be omitted if the function is used in a routing predicate and the corresponding destination is a node of the desired queue.

RJ(node name) - Related Jobs. RJ returns the number of jobs related to the job causing the function to be invoked. If the node name is given, only jobs at that node are counted. Otherwise all of the job's relatives are counted. (Related jobs are produced by fission nodes. See Section 4.9.) If the node name and parentheses are omitted, RJ returns the total number of jobs related to the job.

4.5. Service Centers

Recall that service centers are composed of one or more classes, one or more servers, and a queueing discipline for deciding which job to put into service when a server is free. The classes are local waiting lines at the service center representing different types of jobs with specified work demand distributions. The class names are used in the routing and can be assigned different priority levels.

Servers can serve with a fixed rate of service as illustrated in Section 3.3 or with queue dependent service rates. Queue dependent service rates are given by a vector, where each element applies to the number of customers at the queue. If there are multiple classes and multiple servers, you can specify an arbitrary mapping between which servers will serve which classes.

In addition to the three types of queues discussed in Section 3.3, there are types for Last-Come-First-Served (LCFS), Processor Sharing (PS), and non-preemptive and preemptive priority (PRTY and PRTYPR). For the ACTIVE queue, other possible queueing disciplines are Shortest-Remaining-Time-First (SRTF) and Longest-Remaining-Time-First (LRTF). LCFS is a pushdown stack. The last job to arrive preempts the current job in service. When a job completes service, the last job preempted resumes service. PS is a limiting version of Round Robin scheduling where the time slice is allowed to approach zero. In this limiting case, all jobs at the service center are served in parallel, each receiving an equal share of the servers. With the priority disciplines, different classes can have different priority levels. Priority one is the highest priority. Additional details about these types of queues can be found in the Reference Manual.

The following example shows a priority queue with two classes which might be used to represent a resource with breakdowns. The class named process is for jobs which are worked on when the resource is up and running. The class named down is for a control job which takes over the server and makes the resource appear as if it has a breakdown. The control job is initiated either at the class down or up at the start of the simulation, and moves from one class to the other based on their respective service times. Since the class down has a higher priority than process, the control job will be put into service ahead of any jobs waiting in the process class.

```
QUEUE: processq
TYPE: prty
CLASS LIST: process
   WORK DEMANDS: proctime
   PRIORITIES: 2
CLASS LIST: down
   WORK DEMANDS: downtime
   PRIORITIES: 1
```

Figure 4 on page 21 shows a diagram with the process queue. Classes beyond the one defined with the queue icon are added with a class icon which looks like a waiting line attached to the queue icon

with a dotted line. This type of modeling construct will be explained further in Section 10 where a submodel for representing breakdowns is presented.

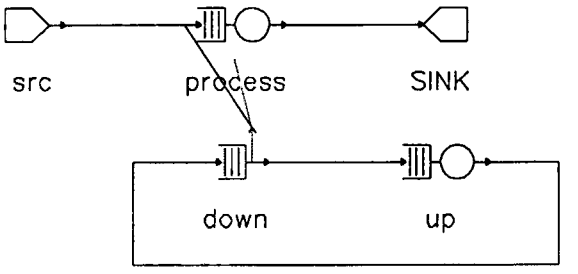

Figure 4. Service Center with breakdowns

4.6. Passive Resources

Recall that passive queues represent resources whose elements are held while the job uses other resources. No service time is associated with the passive queue. The elements of the resources are called tokens and are explicitly acquired and freed. As an example a passive queue can be used to represent a buffer before a communication link.

The passive queue definition contains a pool of tokens to be allocated to jobs. The queue definition also includes a queueing discipline for determining the order in which jobs receive the tokens. Allocate nodes are defined with a distribution for the number of tokens to allocate to the job as it leaves the allocate node. A waiting area and a priority level can be associated with each allocate node. When a job visits a release node, all of the tokens for the associated passive queue are released and returned to the pool of tokens. The token path from a passive queue to an allocate node and from a release node to a passive queue is denoted by a dotted line in model diagrams.

There are times when you would like to change the number of tokens associated with a passive queue. Create and destroy nodes allow you to accomplish this. When a job visits a create node, a specified number of tokens are created and placed in the pool for further allocation. When a job passes through a destroy node, the tokens held from the passive queue are destroyed, instead of being returned to the pool. The following passive queue definition illustrates one example of using create and destroy nodes.

```
QUEUE: batchpq
   TYPE: PASSIVE
   TOKENS: 0
   DSPL: FCFS
   ALLOCATE NODE LIST: batchwait
      NUMBER OF TOKENS TO ALLOCATE: 1
   DESTROY NODE LIST: desbatch
   CREATE NODE LIST: startbatch
      NUMBER OF TOKENS TO CREATE: batchsize
```

We want to collect jobs into batches. Initially, there are zero tokens in the pool. Jobs in the batch are routed to the allocate node and wait until the last job in the batch appears. The last job is routed to the create node to create enough tokens for the entire batch before proceeding to the allocate node. Each job in the batch then obtains one of the tokens and destroys it so the next batch can be accumulated. A create node is represented by a triangle with base on the bottom and an altitude line, and a destroy node is shown as the opposite-oriented icon.

Three other types of nodes can be defined at passive queues: AND allocate nodes, OR allocate nodes, and transfer nodes. In model diagrams, these nodes are represented by an allocate node icon with an A, an allocate node icon with an O, and a triangle with a T, respectively. See the Reference Manual for a discussion of these node types.

4.7. Set Nodes with Multiple Assignment Statements

As discussed previously in Section 3.4, set nodes provide for the execution of assignment statements. In addition to the simple examples presented previously, the assignment lists are permitted to have multiple assignment statements. The following set node illustrates two assignments which are executed for every job passing through this set node.

```
SET NODE: reset
   ASSIGNMENT LIST: numinbatch=0 batchtime=processtim
```

4.8. Job Flow

Section 3.6 discussed some simple forms of routing using probabilities and predicates. The syntax related to predicates was discussed in Section 4.4.2. Below we see two routing lines with multiple conditions tested and the use of a numeric function.

```
: crarr2 -> setbtch2 ;if(ct2=1 or numpart2=1)
: setbtch2 -> SINK ;if(btch2¬=max(btch1,btch2,btch3))
```

The first line contains a predicate testing the values of two global variables using an "or" operator. The second line uses the maximum function.

4.9. Split, Fission and Fusion Nodes

Split, fission and fusion nodes provide the capability of producing copies of jobs and merging related jobs back together again.

4.9.1. Split Nodes

Split nodes allow a job to produce additional independent jobs. A split node has one entrance, an exit for the job that entered, and an additional exit for each new job to be created. The created jobs are given the same job variable values as the creating job. The created jobs do not possess tokens, even though the creating job might be holding tokens. The split node icon is represented by a triangle split in the middle with one entry branch and two exit branches.

As an abstract example of using a split node, let us consider a system with bulk arrivals. In Figure 5 on page 23 a single job arrives from the source and goes to a set node to take a sample from a probability distribution to determine how many jobs are in this batch of arrivals. The set node definition might be

```
SET NODE: setcount
   ASSIGNMENT LIST: numjobs=ceil(uniform(minjobs-1,maxjobs,1)).
```

Minjobs and maxjobs are numeric parameters which specify the minimum and maximum number of jobs in a batch. We use a uniform distribution to create random sized batches and the ceiling (ceil) function to convert the sample from the continuous distribution into an integer. Numjobs is a global variable that keeps track of the number of jobs in the batch. It is initially zero.

If the batch contains only one job, no additional jobs are to be created and the following branch is chosen.

```
: setcount -> queue1 ;if(numjobs=1)
```

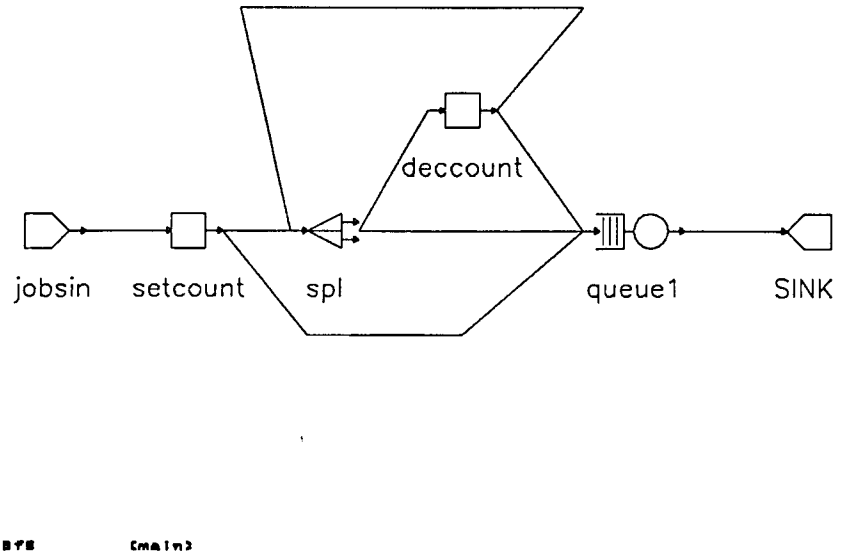

Figure 5. Bulk arrivals

If the batch contains more than one job, the arrival job goes to the split node.

```
: setcount -> spl ;if(numjobs>1)
```

The job passing through the split node in this example always creates one new job. The original job goes to the set node deccount to subtract one from the remaining number of jobs to be created, and the new job goes to queue1.

```
SET NODE: deccount
   ASSIGNMENT LIST: numjobs=numjobs-1
...
   : spl -> deccount ;SPLIT
   : spl -> queue1 ;SPLIT
```

Notice the use of the keyword SPLIT in place of a probability or predicate. This is the syntax RESQ requires when the FROM node is a split node.

After subtracting one from the remaining number of jobs in the batch, the original job goes back to the split node if more jobs are to be created or else goes to queue1.

```
: deccount -> spl ;if(numjobs>1)
: deccount -> queue1 ;if(numjobs=1)
```

In Section 10 we will see that it is easy to build a submodel that incorporates the above logic to do bulk arrivals. This submodel can be used anywhere in a model where one job is supposed to represent the arrival of a group of jobs.

4.9.2. Fission and Fusion Nodes

Fission nodes allow a job to create additional jobs that are dependent on the creating job. Fusion nodes allow for the destruction of the created jobs in a coordinated manner. Fission and fusion nodes are usually used together in pairs. They are useful for representing synchronized processes (tasks) occurring in operating systems. Similarly, fission and fusion nodes are useful for representing parallel physical activities representing a single logical activity, for example transmission of a message across a communication network as a collection of packets.

A fission node has one entrance, an exit for the job that entered (referred to as the "parent"), and an additional exit for each new job to be created. The created jobs are referred to as "children." Children may themselves enter fission nodes, thus creating hierarchies of jobs (see Section 8 of the Reference Manual). Children are given the same job variable values as the parent. The children do not get any of their parent's tokens at the fission node. *Jobs are not allowed to go to a SINK as long as they have relatives (parents or children).* If this rule is violated, the simulation terminates.

In the model diagrams we represent a fission node by a triangle with the entrance at one vertex and two exits on the opposite side. This corresponds to the split node icon except that the triangle is not divided into separate sub-triangles for the parent and children exits. In the dialogue syntax, fission nodes are treated exactly the same as split nodes, except that the keyword "FISSION" is used instead of the keyword "SPLIT."

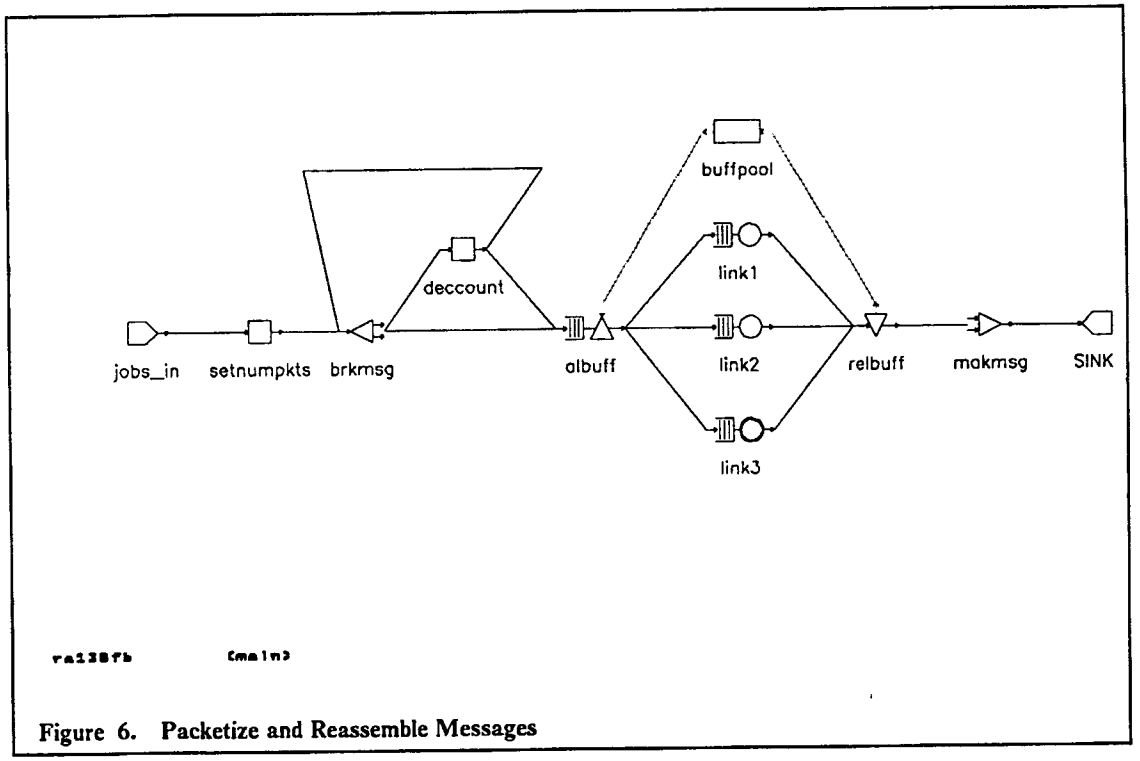

Figure 6. Packetize and Reassemble Messages

A fusion node provides a place for jobs to wait for related jobs (a parent or children). (A fusion node acts as a dummy node for jobs without relatives, i.e., such jobs pass through a fusion node without delay or other effect.) No more than one job of a "family" can stay at a fusion node. If a job arrives at a fusion node and it has relatives, but none of its relatives are at this particular fusion node, it waits at the fusion node. When a job arrives at a fusion node and it has a relative at this particular fusion node, two things can happen, depending on the relationship between the jobs. If one is the parent and the other is a child, then the offspring is destroyed. If both are children, the one that was created last is destroyed. Before a child is destroyed, any tokens it holds are released. After destruction of one job, if the other job has no remaining relatives, it proceeds from the exit of the fusion node. If the other job still has relatives, it waits at the fusion node for another relative

to arrive. This provides the synchronization mechanism required before allowing the job to proceed.

The icon for a fusion node is a triangle with the exit at one vertex and the entrance(s) on the opposite side. Fusion nodes appear in the routing without further distinction, i.e., there is no need for a keyword as in the case of split and fission nodes.

Figure 6 on page 24 shows a model which breaks messages down into packets which are transmitted on separate links and then reassembled into messages again after the transmission is completed. Breaking the messages down into separate jobs is done exactly as was done for the bulk arrives, but uses a fission node in place of a split node. Then there is an allocate node where packages wait until a link is free. A job to be transmitted goes to the first available link, notifies a waiting packet that a link is free and goes to a fusion node to reassemble the message. The routing from the allocate node to a free link is slightly different from previous examples, so we show it here for illustrative purposes.

```
: albuff -> link1 ;if(ql=0)
: albuff -> link2 ;if(ql=0)
: albuff -> link3 ;if(ql=0)
```

This routing displays the searching for the first available link.

4.10. Wait Nodes

Wait nodes provide a place for a job to wait until a specified condition occurs. The syntax of the wait node is shown in the following example.

```
WAIT NODE: waitforbuf
   PREDICATE LIST: until(tabufferq(nextlink)>0)
```

A wait node has a name and an until condition that follows the rules associated with routing predicates. In the above example, jobs will wait at node waitforbuf until there are buffers available at the next link. Tabufferq is a global variable which contains the number of buffers available for a list of links, and nextlink is an index for the next link the job is to go to.

4.11. Dummy Nodes

Dummy nodes are places where nothing happens to a job. They are most frequently used to make routing decisions for jobs leaving split and fission nodes when there are multiple destinations. The following example, although an abstract illustration, will demonstrate this case.

```
DUMMY NODE: d1
 ...
   : fis1 -> n1 ;FISSION
   : fis1 -> d1 ;FISSION
   : d1 -> n2 ;0.3
   : d1 -> n3 ;0.7
```

Here the dummy node d1 acts as the decision point for going to nodes n2 and n3 with probabilities 0.3 and 0.7. These probabilities could not have been specified from the fission node because of the syntax that uses the keyword FISSION.

5. Submodels and Invocations

Submodels provide a facility for macro definition of subnetworks. A submodel is a template for a subnetwork which the user wishes to explicitly encapsulate (1) because this clarifies the model structure, (2) because several such subnetworks (with parameterizable differences) appear in a model, and/or (3) because this submodel is to be (may be) used in other models.

When one uses ("invokes") a submodel with a set of parameter values, then a set of queues and nodes with the specified values and relationships is added to the network, just as the invocation of an assembly language macro causes a set of instructions to be added to a program. It is important for the user to think in terms of macros rather than procedures in properly understanding submodels and how they may be used.

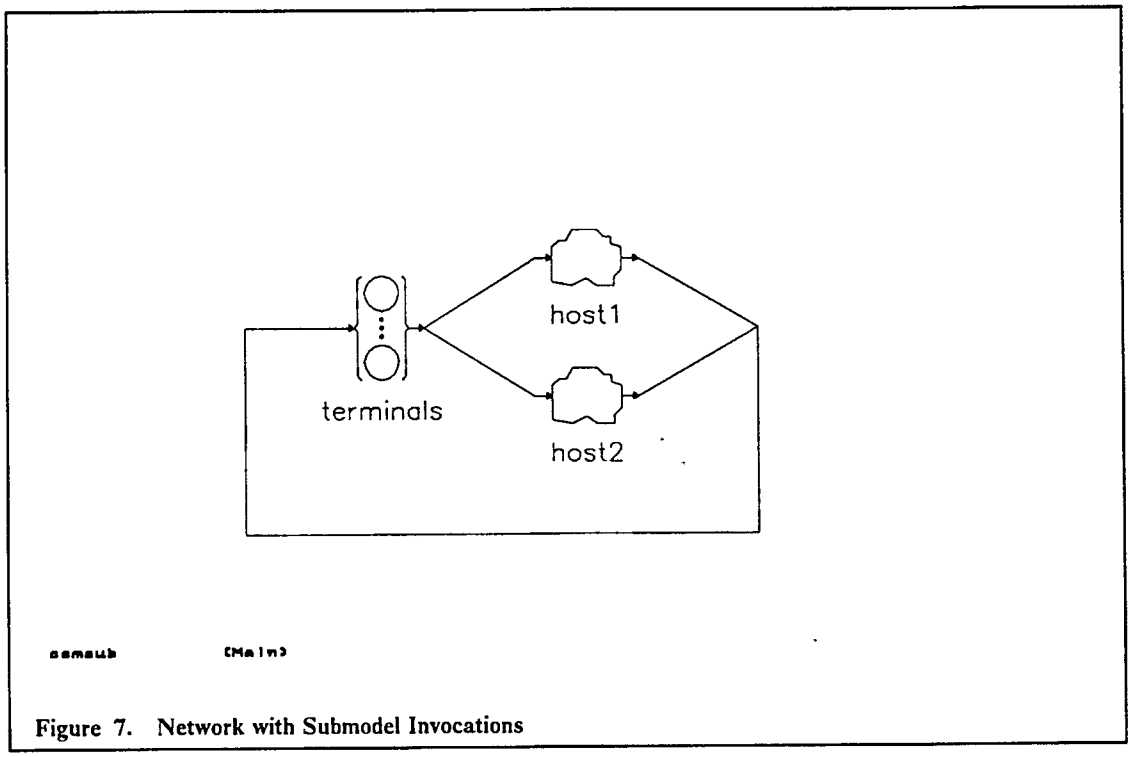

Figure 7. Network with Submodel Invocations

Many of the examples we have given are easily (and appropriately) restated using submodels. This will be further illustrated in Section 10. Here we consider a simple model with two host systems. Figure 7 shows the model level with two invocations of a computer system submodel. The cloud-shaped icon is used in RESQME to represent a generic invocation of a submodel. The specific submodel it invokes is specified, as we will show, in the attributes of that icon. User-created icons can also be drawn to invoke specific submodels and to pictorially represent them. The details contained in the submodel are shown in Figure 8 on page 27. The following example displays the textual information related to the submodel.

```
SUBMODEL: cssm
   NUMERIC PARAMETERS: pageframes floppytime disktime cputime
   CHAIN PARAMETERS: chn
   NUMERIC IDENTIFIERS: cpiocycles
      CPIOCYCLES: 8
QUEUE: cpuq
TYPE: PS
CLASS LIST: cpu
   WORK DEMANDS: cputime
```

```
QUEUE: diskq
TYPE: FCFS
CLASS LIST: disk
   WORK DEMANDS: disktime
QUEUE: floppyq
TYPE: FCFS
CLASS LIST: floppy
   WORK DEMANDS: floppytime
QUEUE: memory
   TYPE: PASSIVE
   TOKENS: pageframes
   DSPL: FCFS
   ALLOCATE NODE LIST: getmemory
      NUMBER OF TOKENS TO ALLOCATE: discrete(16,.25;32,.5;48,.25)
   RELEASE NODE LIST: freememory
CHAIN: chn
   TYPE: EXTERNAL
   INPUT: getmemory
   OUTPUT: freememory
   : getmemory -> cpu ;1.0
   : cpu -> disk ;.1
   : cpu -> floppy ;.9
   : disk -> freememory ;1/cpiocycles
   : floppy -> freememory ;1/1/cpiocycles
   : floppy -> cpu ;1-1/cpiocycles
   : disk -> cpu ;1-1/cpiocycles
END OF SUBMODEL cssm
```

Notice that the dialogue very closely parallels the dialogue for an entire model.

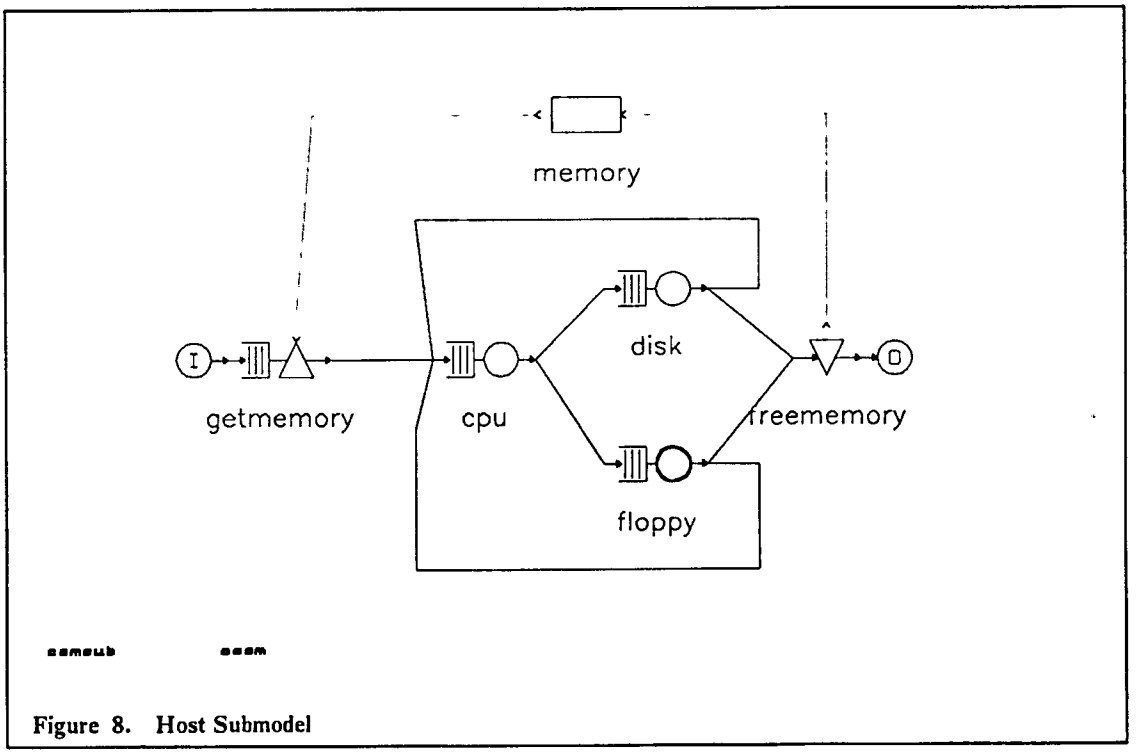

Figure 8. Host Submodel

The submodel name, parameters and identifiers are defined in the submodel header. A submodel is only part of a network. For a network including a submodel to be meaningful, there must be at least one chain which is partially defined inside the submodel and partially defined outside the

submodel. The chain parameter is necessary because the chain defined in the submodel will be attached to a chain at the model level. Here the four queue definitions are similar to the ones we discussed in Sections 2 and 3. The service times are specified by the numeric parameters which are given values when the submodel is invoked. The chain type is EXTERNAL because it will be connected to a chain outside of the submodel. The chain type of a chain parameter is defined as "external" because the usual type, open or closed, is not determined until the chain definition is completed outside of the submodel. **Be careful not to connect an external chain containing a source or a sink to a closed chain at a higher level.** An external chain has a primary input node and a primary output node. In many situations, submodels can be used with minimal knowledge of the contents of the submodel. To this end, it is possible to give exactly one node of each chain parameter the synonym "input" and to give exactly one node of each chain parameter the synonym "output." When the submodel is invoked, and the chain definition completed, these nodes may be referred to by these synonyms instead of the names used within the submodel. As we will see later, it is also possible for submodels to have any number of entrances and exits. The routing is straightforward, with an identifier being used for some branching probabilities.

The model header is displayed here because some of the symbolic names defined in the header will be used in the invocations that follow.

```
MODEL: csmsub
   METHOD: SIMULATION
   NUMERIC PARAMETER: thinktime
   NUMERIC PARAMETER: users
   NUMERIC PARAMETER: pageframes
   NUMERIC IDENTIFIER: floppytime disktime cputime1 cputime2
      FLOPPYTIME: .22
      DISKTIME: .019
      CPUTIME1: .05
      CPUTIME2: .075
```

An invocation is a specific instance of a submodel with its own parameter values (and its own nodes, queues and "global" variables which are defined locally within the submodel). The submodel is invoked two times in this example, and the following shows the textual information for these invocations.

```
INVOCATION: host1
   TYPE: cssm
   PAGEFRAMES: pageframes
   FLOPPYTIME: floppytime
   DISKTIME: disktime
   CPUTIME: cputime1
   CHN: interactiv
INVOCATION: host2
   TYPE: cssm
   PAGEFRAMES: pageframes
   FLOPPYTIME: floppytime
   DISKTIME: disktime
   CPUTIME: cputime2
   CHN: interactiv
```

The INVOCATION: prompt requests a name for the invocation. This name will be needed later for qualification of the names of elements of the submodel. The TYPE: prompt requests the name of the submodel being invoked. After giving the submodel name, the remaining prompts of the invocation are for the parameter values. For numeric and distribution parameters, the values given must be expressions consisting of constants and previously defined identifiers (possibly including model parameters). Each invocation is similar, the differences being in the values given for the numeric parameters. The value given for a chain parameter will be the first appearance of that chain name, unless it is a previously defined chain, has previously been used in another invocation or is a chain array. The chain name is assigned to the submodel chain parameter. This will be the name of the chain at the model level as shown below. Every invocation will require an invocation name, submodel name (TYPE:), and submodel chain parameter.

```
CHAIN: interactiv
   TYPE: CLOSED
   POPULATION: users
   : terminals -> host1.INPUT ;1.0/2.0
   : terminals -> host2.INPUT ;1.0/2.0
   : host1.OUTPUT -> terminals ;1.0
   : host2.OUTPUT -> terminals ;1.0
```

This is a closed chain with the numeric parameter users specified as the population. When it is necessary to refer to elements of the invoked submodel, these names are qualified by the invocation name in the form "invocation.element". Notice that in the above example the invocation names are used to qualify the submodel input and output nodes. The input node is the node named getmemory in the submodel, and the output node is the node named freememory. This correspondence is done automatically for you when you use the RESQME graphical interface. The invocation icon is shaped like a cloud. A user-created icon can be drawn to represent an invocation of a specific submodel.

Submodels allow the modeler to define once how a subsystem is to operate and then to use that definition, perhaps with different parameter values, many times in one model or in many models. People can develop submodels and put them in libraries to be shared by other modelers. The other modelers need only know the assumptions and meaning of the parameters to link together these submodels to form their own model. This provides for reuse of pre-tested code for faster, more accurate modeling.

Many more examples of using submodels will be presented in Section 10.

6. Distributions of Queueing Time, Queue Length, and Tokens

Performance measures for the queueing time distribution, the queue length distribution, the tokens in use distribution and the total number of tokens in the pool are gathered only for the queues and nodes specifically requested for by the user. In order for the simulation to estimate a distribution, it must reserve storage for each point on the distribution. All other performance measure are automatically produced.

The queueing time distribution is a continuous distribution which represents the probability that the queueing time is less than or equal to a certain queueing time value. Rather than attempt to guess which points of the distribution should be gathered for each queue and node and reserve a large amount of storage for information that may not be of interest to the user, the simulation requires that the user specify which distributions are to be gathered and what points of the distributions are to be considered. The following prompts reflect information requested for all types of distributions.

```
QUEUES FOR QUEUEING TIME DIST: memoryq
   VALUES: 1 2 3 4 5 6 7 8
QUEUES FOR QUEUE LENGTH DIST: memoryq
   MAX VALUE: users/2
QUEUES FOR TOKEN USE DIST:
   MAX VALUE:
QUEUES FOR TOTAL TOKEN DIST:
   MAX VALUE:
NODES FOR QUEUEING TIME DIST:
   VALUES:
NODES FOR QUEUE LENGTH DIST:
   MAX VALUE:
```

The first prompt is asking for a list of queue names for which the user wants the queueing time distribution (qtd) and a list of values.

The queue length distribution (qld) gives the probability of having a queue length of each value from zero up to some maximum value. The information related to token use (tud) and total number of tokens in the pool (ttd) is similar to the queue length distribution, but for tokens rather than number of jobs. The distribution of the total number of tokens is of interest only for passive queues which have create or destroy nodes where the number of tokens can change.

The prompts related to nodes for qtd and qld are the same as the ones for queues, except the information is gathered on a node basis.

Sample plots for the queueing time distribution and the queue length distribution are shown below in Figure 9 on page 31 and Figure 10 on page 31.

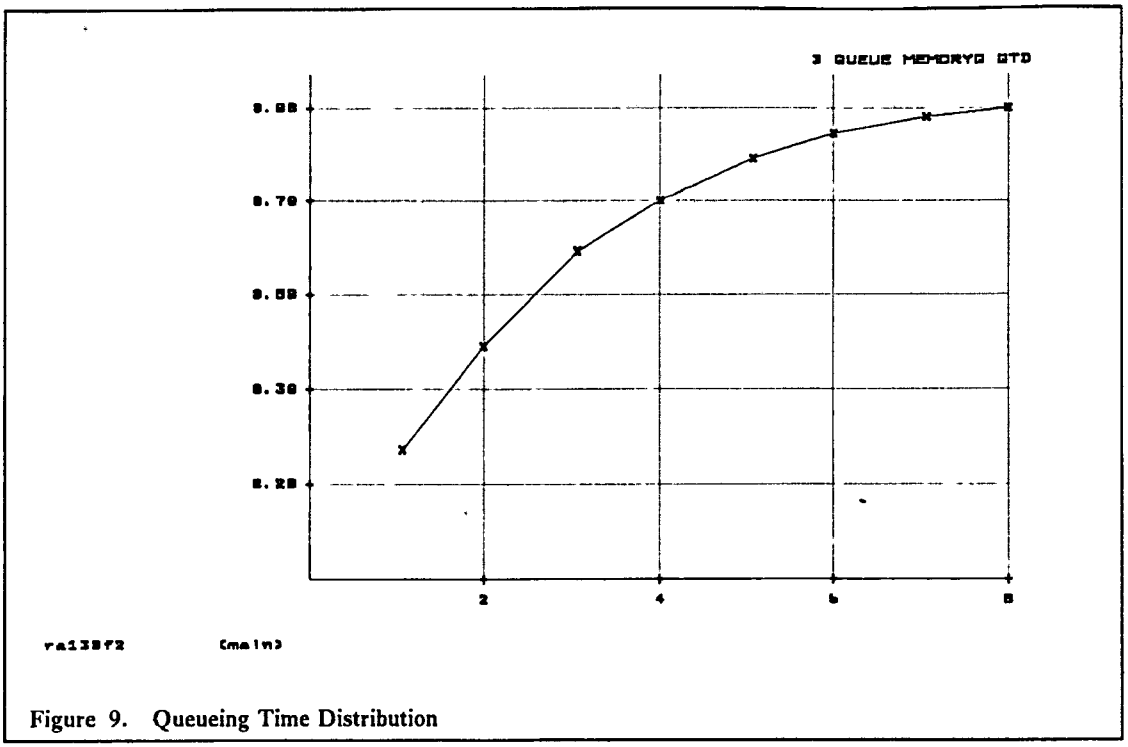

Figure 9. Queueing Time Distribution

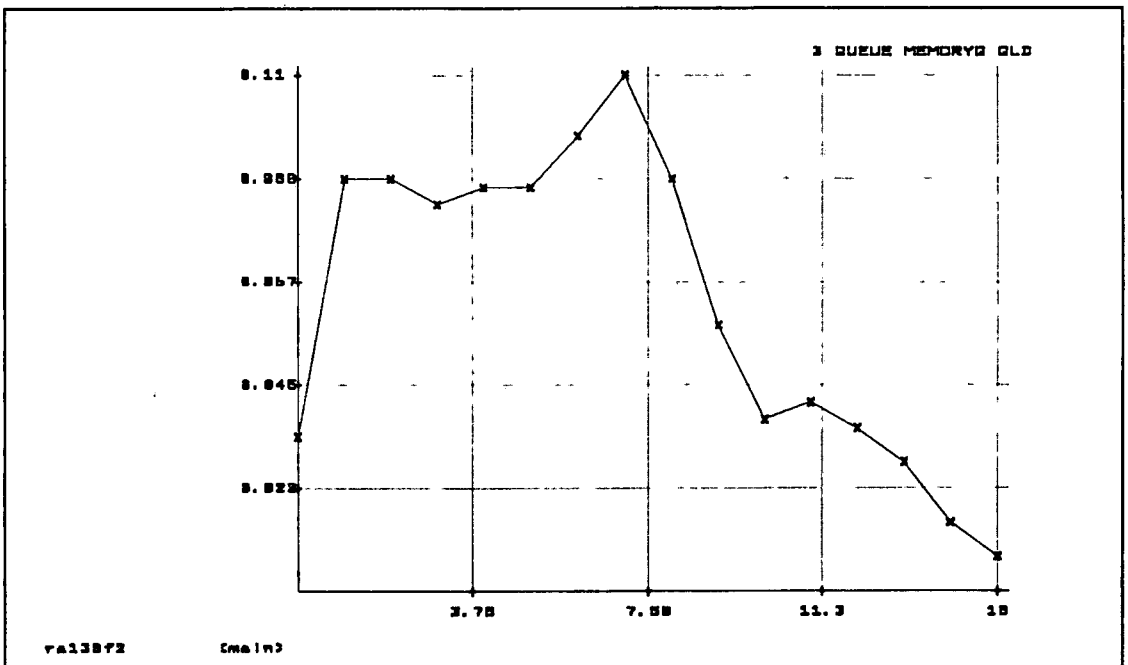

Figure 10. Queue Length Distribution

7. Confidence Interval Methods

We have previously referred to one of the most troublesome problems with simulation: *We need some indication of the accuracy of simulation estimates because of the statistical variability of simulation estimates.* This statistical variability is due to the use of random number streams to drive the simulation. Assuming numerical errors are small, there is no corresponding problem with numerical solution. For example, when we obtained the mean response time estimate of 1.958 seconds in Section 2 that was an exact value *for the model.* The difference between that value and the mean response time of the modeled system is due to inaccuracies of the model and of parameter estimation, not to inaccuracy of solution. When we looked at the results of the simulation model in Section 3, we had no idea of how accurate these estimates were *for the results for the model,* much less the modeled system. Though we usually expect the inaccuracies of our models to be the principal source of error in model estimates, it behooves us to attempt some estimate of the error introduced by statistical variability.

The usual method of estimating variability of simulation results is to produce "confidence interval" estimates: given some point estimate p of a result and information related to the statistical variability around p, we produce a confidence interval estimate around p $(p - \delta, p + \delta)$ and estimate the "true" value (for the model) is contained within the interval with some chosen probability, say 0.9. This probability, expressed in percent, e.g., 90%, is known as the "confidence level." The quantity δ depends on the confidence level; the higher the confidence level is, the larger δ is. We will use the term "confidence interval" to avoid the mouthful "confidence interval estimate," but it should be remembered that the confidence intervals are only estimates. Note that the true value may lie outside of the confidence interval, but this happens only with a small probability (e.g., 1 - .9 = .1). If a simulation is not run long enough, or if the performance measure considered is highly variable, then δ may be greater than p and $p - \delta$ may be negative even though the performance measure must be non-negative. Similarly, for performance measures known to be no greater than 1, e.g., utilizations, p and δ may be such that $p + \delta > 1$.

RESQ provides three methods for confidence interval estimation. The methods are implemented to be as transparent to the user as is practical, i.e., to minimize user decision making and to minimize required user understanding of the statistical bases of the methods. No one method is best for all applications.

- The method of independent replications is the preferred method for estimation of transient characteristics. Independent replications may be applied to estimation of equilibrium characteristics, but one of the following two methods may be preferable for estimating equilibrium characteristics.
- The regenerative method is the preferred method for estimation of equilibrium behavior in models with regenerative characteristics. Some models constructed with RESQ will have regenerative characteristics, but many other models will not.
- The spectral method is the preferred method for estimation of equilibrium behavior in models without regenerative characteristics. The regenerative method requires more user sophistication than the spectral method in that the user must be able to define "regeneration states." Definition of a model to use the spectral method is no more difficult than definition of a model to be simulated without confidence intervals.

The regenerative method and the spectral method allow automated run length control based on achieving confidence intervals of a prespecified width. All three methods are discussed from a statistical point of view in Chapter 6 of Lavenberg 1983. Other references in the Bibliography discuss the statistical aspects of the regenerative method and the spectral method in more detail.

7.1. Independent Replications

A classical method for obtaining confidence intervals is the method of independent replications. With independent replications we repeat the simulation run several times with everything except the random number stream reset to the original initial state for each replication after the first. By a state of the model we mean the location of jobs at specified nodes, e.g., all jobs at the terminals. The

random number streams for the second replication begin where the streams for the first replication ended, the streams for the third replication begin where the streams for the second replication ended, etc..

The dialogue for specifying independent replications for the model in Figure 2 on page 3 is shown below.

```
CONFIDENCE INTERVAL METHOD: REPLICATIONS
INITIAL STATE DEFINITION-
CHAIN: interactiv
   NODE LIST: terminals
   INIT POP: users
CONFIDENCE LEVEL: 90
NUMBER OF REPLICATIONS: 5
INITIAL PORTION DISCARDED: 10
REPLIC LIMITS-
   SIMULATED TIME:
   EVENTS: 10000
   QUEUES FOR DEPARTURE COUNTS:
      DEPARTURES:
   NODES FOR DEPARTURE COUNTS:
      DEPARTURES:
LIMIT - CP SECONDS: 1500
SEED:
```

The confidence interval method is specified as REPLICATIONS by selecting it from the list of available methods. The initialization state definition uses the numeric parameter users for placing all the jobs at the terminals. The confidence level is 90(%). This will be used to calculate the confidence interval widths. We want to make five independent replications and discard 10(%) of the beginning of each replication. Then there is a section to specify how each replication will end. We chose to end each replication after 10,000 events. The other limits that could be specified are the internal clock (simulated) time and queue and node departure counts. When multiple conditions are specified, the first one detected would end the replication. We specify an overall cpu limit and use the default seed to start the random number streams. The CP SECONDS limit is the total for all replications.

It is often advisable to discard results from the initial transient phase of a replication or run. Results from the remainder of the run are, presumably, more representative of the equilibrium behavior to be studied if the effects of the initial system state can be masked. For a formal discussion, see Chapter 6 of Lavenberg 1983. The expression for the INITIAL PORTION DISCARDED: gives the fraction, in percent, of each replication or run that will be discarded. This fraction applies only to the limits in the REPLIC LIMITS and not to the CP SECONDS limit. For each limit of the section, a temporary limit is established by multiplying the given limit by the fraction. Once one of these temporary limits is reached, the variables used to accumulate performance measures are reset, the original limits are put in effect, and the replication or run continues.

Usually we are interested in equilibrium behavior of the modeled system. In this case we wish to have the replications long so the effects of our choice of initial state will not be noticeable. *We prefer a few longer replications to many shorter replications.* Usually we choose the number of replications to be between 5 and 10. The only significant exception is when we want the replications short because we want to notice the effects of our choice of initial state, i.e., we are interested in transient behavior rather than equilibrium behavior. In that case it may be quite reasonable to have many (20 or more) replications.

7.2. Regenerative Method and Spectral Method

For most models of computer systems and communication networks, the method of independent replications is the only method of the three described here that will be used. The following two methods have disadvantages that normally limit their use. The major problems which limit the use of these methods is finding a regeneration state for the regenerative method and the small number of performance measures for which confidence intervals are produced by the spectral method.

7.2.1. Regenerative Method

The regenerative method can only be used to estimate confidence intervals for models that reach equilibrium. The main problem with the regenerative method is that we must select a regeneration state that has the properties that

- The model periodically returns to the regeneration state. The periods between occurrences of the regeneration state are called "cycles."
- When the model enters the regeneration state, the future behavior of the model depends only on the regeneration state, i.e., it is independent of the behavior that led to the entrance of that state. Another way of saying this is that there are random times at which the model probabilistically starts over again.

However, the regenerative method does have advantages over the method of independent replications. It operates on a single (long) run of the simulation instead of multiple (shorter) runs, and we do not need to be concerned about the effects of the choice of the initial state. Because of the single run, the regenerative method is normally more efficient in the amount of cpu time that is necessary to run the model. If we start the simulation in the regeneration state, the first regeneration cycle is statistically identical to every other cycle, so there is no need to discard any data from the run. This method lets us use a sequential stopping rule to automatically detect when a specified level of accuracy has been obtained. Using the stopping rule, the simulation run length is defined in terms of desired confidence interval widths. The simulation runs for a number of regeneration cycles and then confidence intervals are obtained. If the intervals do not meet the width criteria, the simulation continues for more cycles. Then new estimates are made and a new decision to terminate or continue is reached. This continues until the criteria are satisfied or the CPU limit is reached. The groups of regeneration cycles will be referred to as "sampling periods."

The principal practical consideration is that we would like the regeneration state to occur frequently during a simulation of reasonable length. By "frequently" we mean that there be at least some minimum number of cycles (say 20) during the simulation. If we do not have this property then we cannot reasonably use the regenerative method. This is what limits its use in computer and communication models. It is often very difficult to find a regeneration state.

We would also like the regeneration state to be one that is easily detected by the simulation program. For this reason, RESQ only allows regeneration states which are specified by the number of jobs at each node with the understanding that additional characteristics of the states are specified implicitly. These implicitly specified characteristics are (1) Where arrival and service time distributions are specified by the method of exponential stages (see Appendix 3 of the Reference Manual) any arrival and service times in progress are in the first stage in the regeneration state. (2) At active queues where different orderings of the jobs in the queue are important (e.g., FCFS queueing discipline) the ordering of jobs of different classes is the same as at the first occurrence of the required number of jobs at all nodes. (3) At passive queues the ordering of jobs of different allocate nodes and different numbers of tokens requested is the same as at the first occurrence of the required numbers of jobs at each node. (4) Chain variables of open chains have the value one. In addition to these checks, the simulation program issues warning conditions when an apparently correctly defined regeneration state is not actually a regeneration state.

For further discussion of the regenerative method in general, see Chapter 4 of Kobayashi 1978, Chapter 6 of Lavenberg 1982, and Chapter 7 of Sauer and Chandy 1981.

The following is a possible dialogue for using the regenerative method. We will not discuss all the various possible ways of using this method. The interested reader can find further details in the Reference Manual.

```
CONFIDENCE INTERVAL METHOD: REGENERATIVE
REGENERATIVE STATE DEFINITION-
CHAIN: memoryq
   NODE LIST: terminals
   REGEN POP: users
   INIT POP: users
CONFIDENCE LEVEL: 90
SEQUENTIAL STOPPING RULE: YES
   QUEUES TO BE CHECKED: memoryq
      MEASURES: qt
      ALLOWED WIDTHS: 10
   EXTRA SAMPLING PERIODS:
SAMPLING PERIOD GUIDELINES-
   SIMULATED TIME:
   CYCLES:
   EVENTS: 5000
   QUEUES FOR DEPARTURE COUNTS:
      DEPARTURES:
   NODES FOR DEPARTURE COUNTS:
      DEPARTURES:
LIMIT - CP SECONDS: 1500
SEED:
```

The regeneration state definition includes the initial state definition. Here we put all of the jobs at the terminals and look for this state as the regeneration state. Using the sequential stopping rule, the simulation program will check the confidence interval for the mean queueing time at the memoryq queue and stop when it is less than or equal to 10% wide. Extra sampling periods force the simulation to run longer and make sure that the accuracy criteria are satisfied multiple times. The sampling period guidelines specify how frequently the simulation program should check whether to stop. Here we check after every 5000 events.

7.2.2. Spectral Method

The spectral method can be used for models that reach equilibrium, but do not regenerate a sufficient number of times. Most methods in classical statistics for estimating confidence intervals depend on having data that are "independent and identically distributed." The method of independent replications achieves this "i.i.d." property by repeating the simulation with different random numbers. The regenerative method depends on being able to observe the i.i.d. property for the data in each regeneration cycle. The spectral method does not depend on the i.i.d. property. Rather, it explicitly takes into consideration the correlation between data items in the simulation, e.g., the dependencies between successive queueing times for a given queue. This is done without user awareness, other than the calculation of confidence intervals, so the dialogue for simulation using the spectral method is essentially the same as simulation without confidence intervals. A sequential stopping rule is available with the spectral method, but it is a slightly different rule than the one used with the regenerative method. A significant advantage of the spectral method over independent replications is that we can make a single (long) run instead of multiple (shorter) runs, and thus we need not be as concerned about the effects of the choice of initial state. A disadvantage of the way the spectral method is implemented in RESQ is that confidence intervals are only produced for the mean queueing time (QT) and the points on the queueing time distribution (QTD).

The following illustrates a possible dialogue for use with the spectral method.

```
CONFIDENCE INTERVAL METHOD: SPECTRAL
INITIAL STATE DEFINITION-
CHAIN: interactiv
   NODE LIST: terminals
   INIT POP: users
CONFIDENCE LEVEL: 90
SEQUENTIAL STOPPING RULE: YES
   CONFIDENCE INTERVAL QUEUES: memoryq
      MEASURES: qt
      ALLOWED WIDTHS: 10
   CONFIDENCE INTERVAL NODES:
      MEASURES:
      ALLOWED WIDTHS:
   EXTRA SAMPLING PERIODS:
INITIAL PORTION DISCARDED: 10
INITIAL PERIOD LIMITS-
   SIMULATED TIME:
   EVENTS: 5000
   QUEUES FOR DEPARTURE COUNTS:
      DEPARTURES:
   NODES FOR DEPARTURE COUNTS:
      DEPARTURES:
LIMIT - CP SECONDS: 1500
```

The initial state definition is exactly the same as with confidence interval method NONE. The sequential stopping rule is checking the accuracy of the confidence interval for the mean queueing time at the memoryq queue. We are discarding 10% of the initial period which has a length of 5000 events.

8. Trace and Animation

Trace is mainly used for model debugging and when used with animation, as a communication device. The trace provides detailed textual information related to events that are occurring while the simulation is running. Animation provides a moving picture of jobs and tokens flowing in the model diagram. The trace provides the necessary input to produce animation. Animation is useful in model debugging and also to communicate the model logic to others. Animation enhances the understanding of model interactions and the impact of changes. It therefore can serve as a central visual reference for a team of personnel working on a project.

The trace can be started and stopped at various times specified by the user and comes in five formats: job movement, queues, event handling, event list, snapshots. The following dialogue shows how to specify when the trace starts, when the trace stops, and the type of trace required.

```
TRACE: YES
INITIALLY ON: YES
TURN TRACE ON-
   SIMULATED TIME:
   CYCLES:
   EVENTS: 0
   QUEUES FOR DEPARTURE COUNTS:
      DEPARTURES:
   NODES FOR DEPARTURE COUNTS:
      DEPARTURES:
TURN TRACE OFF-
   SIMULATED TIME:
   CYCLES:
   EVENTS: 500
   QUEUES FOR DEPARTURE COUNTS:
      DEPARTURES:
   NODES FOR DEPARTURE COUNTS:
      DEPARTURES:
JOB MOVEMENT: YES
QUEUES: no
EVENT HANDLING: NO
EVENT LIST: NO
SNAPSHOTS: NO
```

In this example, we initially have the trace on and start it at event 0, which is the beginning of the simulation. We turn the trace off after 500 events. The following shows an example of job movement in a textual trace file.

```
ARRIVE -- JOB            12 DEPARTURE      1 FROM TERMINALS(CLASS)
          CURRENT TIME:  1.7631795E+000 NUMBER OF EVENTS:         1
          POPULATION:   15 JOBS,   0 JOB COPIES
          DESTINATION, CONDITION:
          ALLOCMEM(ALLOCATE), 0.992826 < 1.000000
```

The simulation routine ARRIVE shows that job number 12 (every RESQ job has a unique job number) is the first departure from the class with node name terminals. This departure occurs at time 1.763 time units and is the first event in this run. There are a total of 15 jobs in the model and no job copies. (A copy of a job is made by RESQ when a job is allocated tokens in order to keep track of the performance measures related to the passive queue.) The destination of job 12 is the allocate node named allocmem. The routing condition in the model is to take this branch with a probability of one, but RESQ still generates a random number (0.992826) to compare against the specified probability. The following shows some more job movement trace. After going through an allocate node, we can see how many tokens a job is holding.

```
ARRIVE -- JOB            12 DEPARTURE      0 FROM ALLOCMEM(ALLOCATE)
          CURRENT TIME:  1.7631795E+000 NUMBER OF EVENTS:        1
          POPULATION:   15 JOBS,   1 JOB COPIES
          TOKENS HELD:          1 AT ALLOCMEM
          DESTINATION, CONDITION:
          SETJOBTYPE(SET), 0.034472 < 1.000000
EXPRT --  = SUB1 JV JOBTYPE DISCRETE ; , 1 1.000E-001 ; , 2 9.000E-001 EOX
```

The line which begins with EXPRT -- is a reverse Polish notation of the assignment statement to be evaluated at the set node. In the model this assignment statement looks like:

```
SET NODE: setjobtype
   ASSIGNMENT LIST: jv(jobtype)=discrete(1,.1;2,.9)
```

The following trace segment shows a different type of node that the job is going to and that one of the job variables is different from zero.

```
ARRIVE -- JOB            12 DEPARTURE      1 FROM SETJOBTYPE(SET)
          CURRENT TIME:  1.7631795E+000 NUMBER OF EVENTS:        1
          POPULATION:   15 JOBS,   0 JOB COPIES
          JV'S¬=0:    0:  2.000E+000
          DESTINATION, CONDITION:
          CPU(CLASS), 0.434226 < 1.000000
```

If we turn on the queue trace we get the following type of information. The routine ALLCTE shows job number 12 at allocate node allocmem which belongs to passive queue memoryq requesting one token. Then routine PQTRAC shows the number of tokens requested and held by job 12, along with the number of tokens in the passive queue pool and the number of tokens available for allocation. Then job 12 goes to the cpu class at active queue cpuq and requests 0.2773 time units of service. The AQTRAC routine gives information about the jobs waiting and in service and about the servers. The COMPLT routine reports when a job completes service. After a while, routine RELEAS shows that job 12 releases the tokens it is holding.

```
ALLCTE -- JOB        12 AT NODE ALLOCMEM QUEUE MEMORYQ TOKEN REQUEST        1
PQTRAC --    JOB  NODE                   PRTY TOKNS HELD?     Q_PTR
              12  ALLOCMEM                 -1     1     1
PQTRAC -- MEMORYQ TOKENS:         4 TOKENS AVAILBLE:         3
SERARR -- JOB        12 AT CLASS CPU QUEUE CPUQ SERVICE REQUEST  2.773E-01
AQTRAC --    TIME       JOB  NODE      SERV PREEM  PRTY  DSTG     Q_PTR
        2.773E-01        12  CPU          1     0    -1     1
AQTRAC -- CPUQ SERVERS:         1 SERVERS AVAILBLE:         0
COMPLT -- JOB        12 AT CLASS CPU QUEUE CPUQ
 ...
RELEAS -- JOB        12 AT NODE RELSEMEM QUEUE MEMORYQ
```

Event handling trace gives us information related to the handling of every event. The following two lines illustrate event handling information related to activity at an active and passive queue.

```
SMULAT -- NO. EVENTS         6 TIME  4.5187236E+00(SERVER) JOB        12 QUEUE I01Q
SMULAT -- NO. EVENTS         6 TIME  4.5187236E+00(PRTYPQ) QUEUE MEMORYQ
```

If we turn on the event list trace we see information pertaining to all pending events. For each event we have the type of event, the job number, the node or queue that the job is at, and whether the event has just been added to the event list.

```
ADEVNT -- T (TYPE) 0:SERVER,1:PSEUDO,2:SOURCE,3:PRTYPQ; J (JOB) ; N|Q (NODE|QUEUE);
       (* => EVENT JUST ADDED)
 T   J N|Q          *  T   J N|Q          *  ...
 0  12 TERMINALSQ    0   8 TERMINALSQ    0   2 TERMINALSQ    0   5 TERMINALSQ
 1.7631807327271E+00 3.4429693222046E+00 3.6118545532227E+00 6.4211244583130E+00
 0  14 TERMINALSQ    0  10 TERMINALSQ    0   9 TERMINALSQ    0   7 TERMINALSQ
 7.2373218536377E+00 8.0025444030762E+00 9.9320631027222E+00 1.7208862304688E+01
 0  13 TERMINALSQ    0   4 TERMINALSQ    0  15 TERMINALSQ *  0   1 TERMINALSQ
 1.8178161621094E+01 1.8462860107422E+01 2.0273315429688E+01 2.0694839477539E+01
 0  11 TERMINALSQ    0   3 TERMINALSQ    0   6 TERMINALSQ
 4.8286651611328E+01 6.4015045166016E+01 1.0617042541504E+02
```

The snapshot trace gives us the queue length and the number of departures at every queue and node in the model.

```
SNPSHT -- QUEUE MEMORYQ LENGTH      4 DEPARTURES      3
SNPSHT -- QUEUE CPUQ LENGTH      1 DEPARTURES      6
SNPSHT -- QUEUE IO1Q LENGTH      2 DEPARTURES      1
SNPSHT -- QUEUE IO2Q LENGTH      0 DEPARTURES      3
SNPSHT -- QUEUE TERMINALSQ LENGTH     11 DEPARTURES      7
SNPSHT -- CLASS   CPU LENGTH      1 DEPARTURES      6
SNPSHT -- CLASS   IO1 LENGTH      2 DEPARTURES      1
SNPSHT -- CLASS   IO2 LENGTH      0 DEPARTURES      3
SNPSHT -- CLASS   TERMINALS LENGTH     11 DEPARTURES      7
SNPSHT -- ALLOCATEALLOCMEM LENGTH      4 DEPARTURES      3
SNPSHT -- RELEASE RELSEMEM DEPARTURES      3
SNPSHT -- SET     SETJOBTYPE DEPARTURES      7
```

As you can imagine if you had to pour through thousands of lines of trace output, animation of the simulation provides a powerful tool for visualizing massive amounts of data, thus reducing the modeler's need to translate from the syntax and semantics of the computer software to that of the real-world system. Models may be more easily debugged due to the modeler's enhanced ability to visually trace the movement of jobs and resource allocation in the model. By observing the evolution of the system, the modeler may also obtain a qualitative understanding of complex phenomena, which then complements the quantitative performance results provided by RESQME. The ability to observe the evolution of the system also offers the possibility of observing and discovering, "in the lab," phenomena which otherwise would have been "washed out" in the average, steady-state statistics. Finally, animation provides an enhanced ability to communicate the model design and the model results to others.

Some of the basic capabilities provided by the RESQME animation facility include:

- the animation of job and token movement in a hierarchical model.
- the ability to modify animation variables, such as speed, color and size of the animated objects or use of event versus linear time.
- the ability to interrupt the animation to modify the model diagram display (move, pan, zoom, layer) or to change animation options.
- selective animation of subsets of nodes, queues, or chains.
- the ability to trace an individual job as it moves through the model and its submodels.
- the ability to (re)start an animation at any point, step through the simulation based on checkpoints, and to stop an animation at breakpoints.

Animation provides a moving picture on the model diagram of some of the information given in the job movement and snapshot trace. As the jobs and tokens move, the queue length and token availability at the nodes and queues are updated and displayed textually and graphically. We can observe jobs moving from one node to another, and tokens being allocated, created, released and destroyed. Elements display the simulation time and event count and identify the job currently being moved.

Animation is implemented in an after-the-fact manner by creating a trace file, containing a partial event history of departure and arrival events. This file is created during the execution of the simulation. The innermost "loop" of the animation component of RESQME simply takes an event (such as, the departure of a job from one node in the model and its subsequent arrival at another node) from the trace file and animates that event on the graphical network. As a result, a given simulation can be animated numerous times without actually re-simulating a model. An advantage of this approach is that the simulation can be replayed with the same timing and sequencing of events. Furthermore, portions of the simulation can be skipped over, while others can be examined in slow motion. A disadvantage is that the modeler cannot interact directly with an on-going simulation via the animation facility.

When setting up the model for simulation, all that needs to be added to get animation is to ask that a trace file be created. To do this, the modeler selects the trace menu item and specifies in the pop-up window the period of simulated activity for which animation is desired. This period can be specified in terms of starting and stopping simulation time or events.

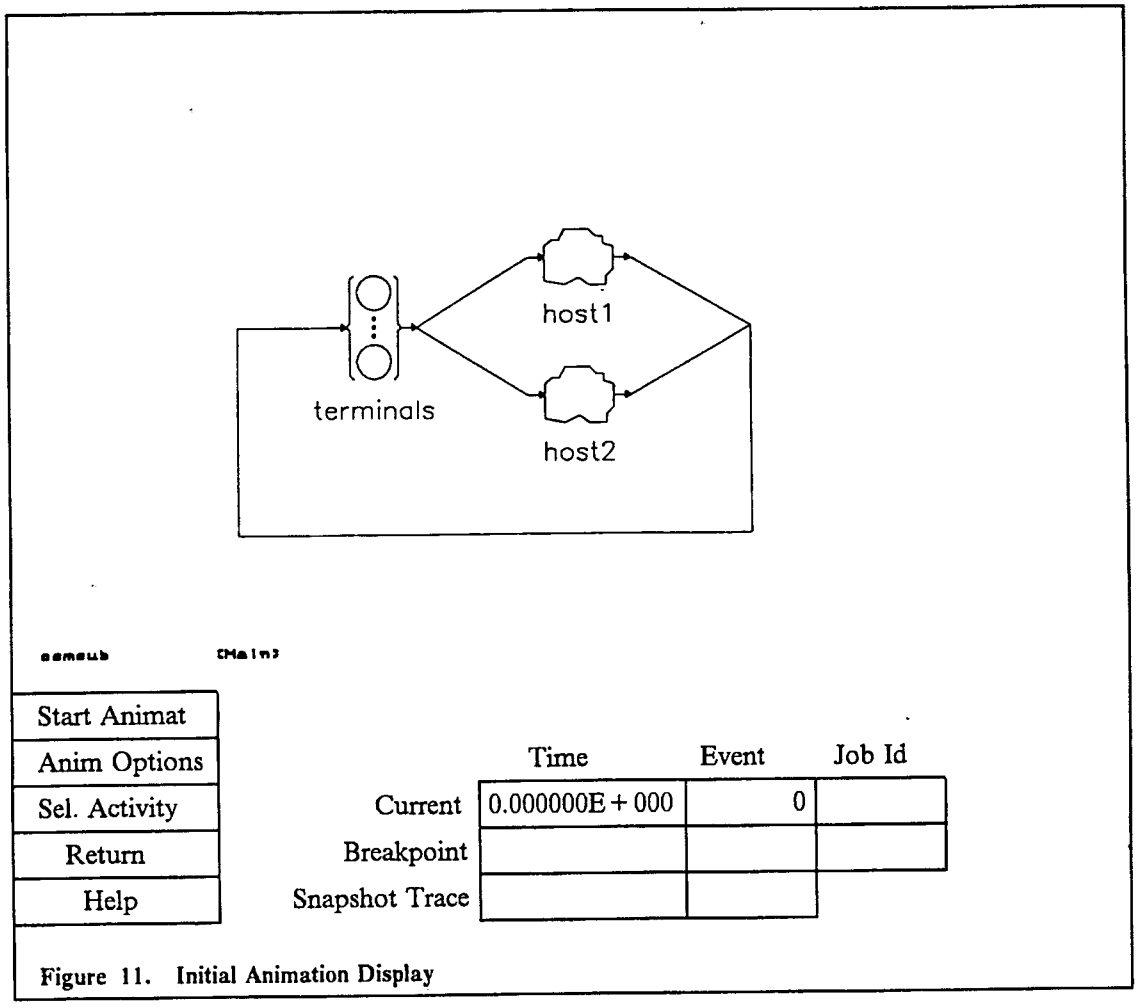

Figure 11. Initial Animation Display

During the simulation of the model, the trace file is created and when the simulation run is completed, the trace file can be played back. The modeler selects the animation menu item in the output analysis task to enter the animation facility. In order to be able to skip forward and backward in the animation a checkpoint file is created that contains snapshots of the system states. A pointer file is also produced that links the snapshots to the appropriate record in the trace file. This method provides an efficient and flexible way of processing what is typically a large trace file. We can then quickly restore the state using the snapshot information in the checkpoint file, and then through the use of the pointer file, we can begin animating events until the desired simulation time has been reached.

Figure 11 shows the resulting RESQME screen layout when the animation facility is entered.

Consistent with other tasks in RESQME, the model diagram is displayed in the main modeling area with the screen management menu on the right border. The model diagram has the initial queue lengths and token pool values displayed above the appropriate nodes, and the nodes are also shaded to reflect their initial queue lengths. As the animation proceeds, balls representing jobs move along the routing lines adjusting the queue lengths at the nodes they reach, and rectangles representing tokens move along the dotted token lines adjusting the passive queue numbers.

The screen management menu can be used throughout all tasks in RESQME to modify the view of the model by zooming, panning, centering on a node, or layering to a submodel view. Its functions are made available within the animation facility as well, since RESQME determines the job and token movements from the endpoint node names of each traversed arc and not from its graphical position. This late binding allows the modeler to change the positions of the nodes and arcs during animation, and RESQME will animate the jobs and tokens along the revised paths. For further details of using the animation facility, see the RESQME OS/2 Guide.

9. Performance Measures

While the trace and animation provide detailed information about the transient behavior of the model, the performance measures provide long run information averaged over the entire run. Performance measures are normally much more useful in understanding the behavior of the system.

The following provides a list of performance measure codes, their associated names, and a brief explanation of each.

- UT - utilization. The utilization is the fraction of time that a server is busy or a token is in use.
- TP - throughput. The throughput is the completion rate or the number of departures per unit time.
- QL - mean queue length. The mean queue length is the average number of jobs waiting and in service at a service station or the average number of jobs waiting or holding tokens at a passive queue.
- SDQL - standard deviation of queue length. The standard deviation of queue length is a measure of variability of the queue length.
- QLD - queue length distribution. For each occurring queue length up to a specified maximum value, the queue length distribution gives the probability of the correponding queue length.
- QT - mean queueing time. The mean queueing time is the average time waiting and in service at a service station or the average time waiting and holding tokens at a passive queue.
- SDQT - standard deviation of queueing time. The standard deviation of queueing time is a measure of variability of the queueing time.
- QTD - queueing time distribution. For a list of values specified by the user, the queueing time distribution gives the probability that the queueing time is less than or equal to each value.
- TU - mean tokens in use. The mean tokens in use is the average number of tokens of a passive queue that are allocated to jobs.
- TUD - distribution of tokens in use. For each occurring number of tokens in use up to a specified maximum value, the distribution of tokens in use gives the probability of the correponding number of tokens in use.
- TT - mean total tokens in pool. This is the same as the initial number of tokens in the pool unless tokens are created or destroyed. When tokens are created or destroyed, the mean total tokens in pool represents the average number of tokens in the pool over the entire run.
- TTD - distribution of tokens in pool. For each possible number of tokens in the pool up to a specified maximum value, the distribution of tokens in pool gives the probability of the correponding number of tokens in the pool.
- MXQL - maximum queue length. The maximum queue length is the largest value of the queue length observed during the run.
- MXQT - maximum queueing time. The maximum queueing time is the largest value of the queueing time observed during the run.
- ND - number of departures. The number of departures is the number of job completions associated with a node or queue. Note that for allocate nodes, a departure does not occur until the allocated tokens are released or destroyed.
- PO - open chain population. The open chain population is the average number of jobs in an open chain during the length of the run.
- RTM - open chain response time. The open chain response time is the average amount of time for a job to go from a source to the SINK.

- ST - mean service time. The mean service time is the average service time observed at a service center during the run. If this value is significantly different from the mean of the distribution specified in the model, the model was probably not run long enough.
- LNG - final queue lengths. The final queue lengths are the number of jobs remaining at each node and queue when the simulation stops.
- JV - final job variable values. The final job variable values are the values of the job variables for the jobs remaining in the model when the simulation stops.
- CV - final chain variable values. The final chain variable values are the values of the chain variables when the simulation stops.
- GV - final global variable values. The final global variable values are the values of the global variables when the simulation stops.

See the Reference Manual for more information about these performance measures.

10. Some Computer and Communication Related Submodels

In this section we will describe several submodels which could be useful in modeling computer systems and communication networks. The purpose is not to develop a comprehensive set of submodels, but rather to give some examples that users can build on and extend. Each subsection will include a description of the submodel, parameters, variables, intialization state if necessary, a submodel listing and diagram, and an example use in a simple model. In the sample model, an INCLUDE: statement identifies the place where the submodel is defined.

10.1. Batch Service - BATCHPRO

10.1.1. Description

The following submodel is for batching jobs in a manufacturing system, but this type of batching also occurs in computer systems and communication networks. This submodel is based on one received from Hans Fromm, IBM German Manufacturing Technology Center, Sindelfingen. It performs batch service, given maximum and minimum batch sizes and a processing time distribution. Three assumptions are: (1) Serves only a single type of process. Multiple processes with different batch sizes and different process time distributions are not handled. (2) There are no setup times and no failures. (3) This submodel represents a single tool and must be invoked multiple times for different tools.

10.1.2. Numeric Parameters

- BATCHMAX, the maximum batch size.
- BATCHMIN, the minimum batch size.

10.1.3. Distribution Parameters

- PROCESSTIME, the processing time distribution for each batch.

10.1.4. Chain Parameters

- CH, the chain paramter to attach to a chain at a higher level.

10.1.5. Global Variables

- BATCHTIME, the processing time for the next batch.

10.1.6. Submodel Listing and Diagram

```
SUBMODEL: batchpro
   NUMERIC PARAMETERS: batchmax batchmin
   DISTRIBUTION PARAMETERS: processtim
   CHAIN PARAMETERS: ch
   GLOBAL VARIABLES: batchtime
      batchtime: 0
QUEUE: processq
TYPE: ACTIVE
SERVERS: batchmax
DSPL: FCFS
CLASS LIST: process
   WORK DEMANDS: constant(batchtime)
SERVER-
   RATES: 1
   ACCEPTS: all
```

```
QUEUE: cycletimeq
   TYPE: PASSIVE
   TOKENS: 999999
   DSPL: FCFS
   ALLOCATE NODE LIST: cyclestart
      NUMBER OF TOKENS TO ALLOCATE: 1
   RELEASE NODE LIST: cycleend
QUEUE: batchpq
   TYPE: PASSIVE
   TOKENS: 0
   DSPL: FCFS
   ALLOCATE NODE LIST: batchwait
      NUMBER OF TOKENS TO ALLOCATE: 1
   DESTROY NODE LIST: desbatch
   CREATE NODE LIST: startbatc1
      NUMBER OF TOKENS TO CREATE: min(ql(batchwait)+1,batchmax)
   CREATE NODE LIST: startbatc2
      NUMBER OF TOKENS TO CREATE: min(ql(batchwait),batchmax)
SET NODE: settime
   ASSIGNMENT LIST: batchtime=processtim
```

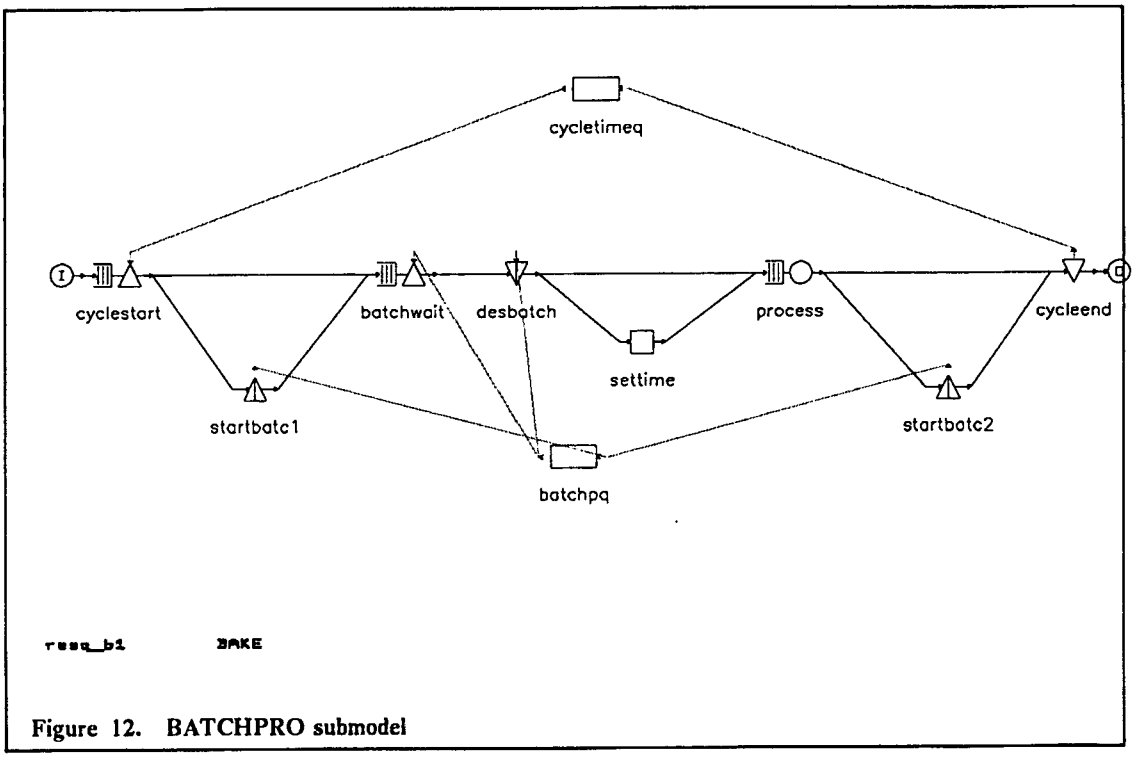

Figure 12. BATCHPRO submodel

```
CHAIN: ch
   TYPE: EXTERNAL
   INPUT: cyclestart
   OUTPUT: cycleend
   : cyclestart -> batchwait ; ++
     if(ql(process)>0 or ql(batchwait)<batchmin-1)
   : cyclestart -> startbatc1 ; ++
     if(ql(process)=0 and ql(batchwait)>=batchmin-1)
   : startbatc1 -> batchwait ;1.0
   : batchwait -> desbatch ;1.0
```

```
   : desbatch -> process ; ++
     if(ql(process)>0)
   : desbatch -> settime ; ++
     if(ql(process)=0)
   : settime -> process ;1.0
   : process -> cycleend ; ++
     if(ql(process)>0 or ql(batchwait)<batchmin)
   : process -> startbatc2 ; ++
     if(ql(process)=0 and ql(batchwait)>=batchmin)
   : startbatc2 -> cycleend ;1.0
END OF SUBMODEL batchpro
```

10.1.7. Model Which Invokes BATCHPRO

```
MODEL: resq_b1
   METHOD: SIMULATION
   NUMERIC PARAMETERS: iat batchmax batchmin
   DISTRIBUTION PARAMETERS: bakest
INCLUDE: batchpro
INVOCATION: bake
   TYPE: batchpro
batchmax: batchmax
batchmin: batchmin
processtim: bakest
ch: ch
CHAIN: ch
   TYPE: OPEN
   SOURCE LIST: src
      ARRIVAL TIMES: iat
   : src -> bake.INPUT ;1.0
   : bake.OUTPUT -> SINK ;1.0
QUEUES FOR QUEUEING TIME DIST: bake.cycletimeq
   VALUES: 1 1.5 2 2.5
CONFIDENCE INTERVAL METHOD: REGENERATIVE
REGENERATIVE STATE DEFINITION-
CONFIDENCE LEVEL: 90
SEQUENTIAL STOPPING RULE: NO
RUN GUIDELINES-
   EVENTS: 20000
LIMIT - CP SECONDS: 100
TRACE: NO
END
```

10.2. Bulk Arrivals - BULKARR

10.2.1. Description

Produces bulk arrivals which allow several jobs to arrive at the same simulated time. The number of jobs to arrive will be an integer between some minimum number of jobs and some maximum number of jobs, with each integer being equally likely.

10.2.2. Numeric Parameters

- MAXJOBS - maximum number of jobs which arrive at one time.
- MINJOBS - minimum number of jobs which arrive at one time.

10.2.3. Chain Parameters

- CH - chain parameter to be assigned to chain at higher level.

10.2.4. Global Variables

- NUMJOBS - the number of jobs generated by an arrival.

10.2.5. Submodel Listing and Diagram

```
SUBMODEL: bulkarr
   NUMERIC PARAMETERS: maxjobs minjobs
   /* maxjobs - maximum number of jobs which arrive at 1 time */
   /* minjobs - minimum number of jobs which arrive at 1 time */
   CHAIN PARAMETERS: ch
   /* ch - chain parameter assigned to chain at higher level */
   GLOBAL VARIABLES: numjobs
      NUMJOBS: 0
   /* numjobs - the number of jobs generated by an arrival */
SET NODE: setcount
   ASSIGNMENT LIST: numjobs=ceil(uniform(minjobs-1,maxjobs,1))
   /* The number of jobs to arrive will be an integer between
   minjobs and maxjobs, with each value being equally likely */
SET NODE: deccount
   ASSIGNMENT LIST: numjobs=numjobs-1
SPLIT NODE: spl
DUMMY NODE: dout
```

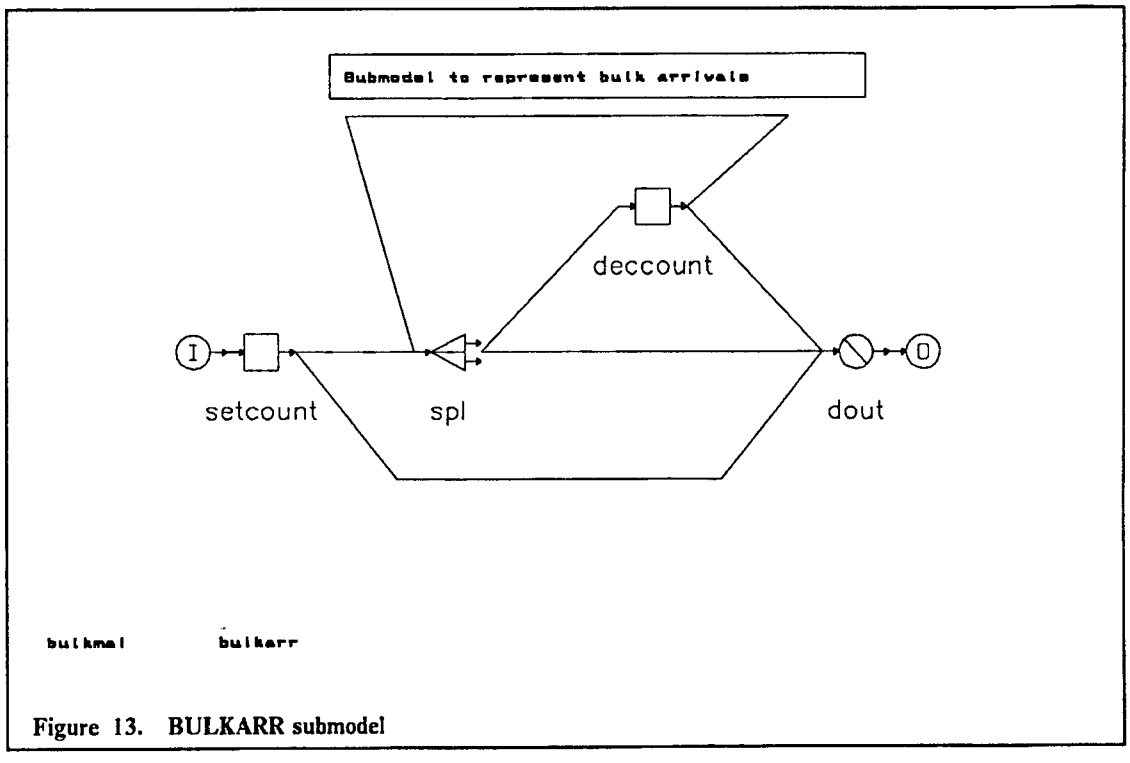

Figure 13. BULKARR submodel

```
CHAIN: ch
   TYPE: EXTERNAL
   INPUT: setcount
   OUTPUT: dout
   : setcount -> spl ;if(numjobs>1)
   : spl -> deccount ;SPLIT
   : spl -> dout ;SPLIT
   : setcount -> dout ;if(numjobs=1)
   : deccount -> dout ;if(numjobs=1)
   : deccount -> spl ;if(numjobs>1)
END OF SUBMODEL bulkarr
```

10.2.6. Model Which Invokes BULKARR

```
MODEL: bulkmai
   METHOD: SIMULATION
   NUMERIC PARAMETERS: iat
   NUMERIC PARAMETERS: st
   NUMERIC PARAMETERS: maxjobs
   NUMERIC PARAMETERS: minjobs
QUEUE: q1
TYPE:  IS
CLASS LIST: c1
   WORK DEMANDS: st
INCLUDE: bulkarr
INVOCATION: bulkinv
   TYPE: bulkarr
   MAXJOBS: maxjobs
   MINJOBS: minjobs
   CH: ch
CHAIN: ch
   TYPE: OPEN
   SOURCE LIST: src
      ARRIVAL TIMES: iat
   : src -> bulkinv.INPUT ;1.0
   : bulkinv.OUTPUT -> c1 ;1.0
   : c1 -> SINK ;1.0
CONFIDENCE INTERVAL METHOD: REGENERATIVE
REGENERATIVE STATE DEFINITION-
CONFIDENCE LEVEL: 90
SEQUENTIAL STOPPING RULE: NO
RUN GUIDELINES-
   EVENTS: 20000
LIMIT - CP SECONDS: 10
TRACE: NO
END
```

10.3. Conditional Arrivals - CARRSUB

10.3.1. Description

Submodel to represent conditional arrivals based on state-dependence.

10.3.2. Numeric Parameters

- LVL - level at which arrivals are turned off.

10.3.3. Node Parameters

- C1 - name of node at which to check QL.

10.3.4. Chain Parameters

- CH - chain parameter of open chain where arrival time is varied.

10.3.5. Chain Variables

- CV(0) - used to vary arrival rate.

10.3.6. Initial State

One job must be initialized at "invocation name".waitbelow.

10.3.7. Submodel listing and Diagram

```
SUBMODEL: carrsub
   NUMERIC PARAMETERS: lvl
   /* Level at which arrivals are turned off. */
   NODE PARAMETERS: c1
   /* Name of node at which to check QL. */
   CHAIN PARAMETERS: ch
   /* Open chain where arrival time is varied. */
SET NODE: setoff
   ASSIGNMENT LIST: cv(0)=0.000001 /* effectively 0 arrivals */
SET NODE: seton
   ASSIGNMENT LIST: cv(0)=1.0 /* original rate */
WAIT NODE: waitbelow
   PREDICATE LIST: until(ql(c1)>=lvl)
WAIT NODE: waitabove
   PREDICATE LIST: until(ql(c1)<lvl)
DUMMY NODE: d1 /* needed because an external chain requires */
               /* an input and output node. */
CHAIN: ch
   TYPE: EXTERNAL
   INPUT: d1
   OUTPUT: d1
   : waitbelow -> setoff ;1.0
   : setoff -> waitabove ;1.0
   : waitabove -> seton ;1.0
   : seton -> waitbelow ;1.0
END OF SUBMODEL carrsub
```

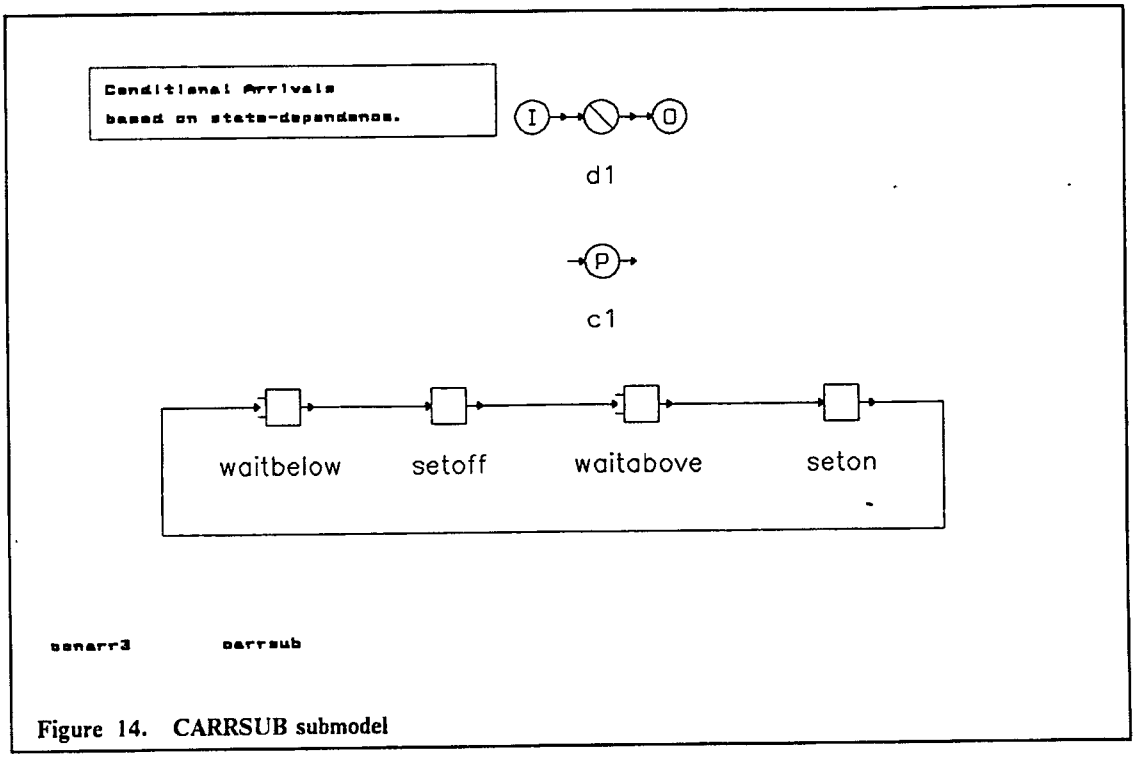

Figure 14. CARRSUB submodel

10.3.8. Model Which Invokes CARRSUB

```
MODEL: conarr3
   METHOD: SIMULATION
   NUMERIC PARAMETERS: lvl
QUEUE: q1
TYPE: FCFS
CLASS LIST: c1
   WORK DEMANDS: .8
INCLUDE: carrsub
INVOCATION: statedep
   TYPE: carrsub
   LVL: lvl
   C1: c1
   CH: ch
CHAIN: ch
   TYPE: OPEN
   SOURCE LIST: s
      ARRIVAL TIMES: 1
   : s -> c1 ;1.0
   : c1 -> SINK ;1.0
CONFIDENCE INTERVAL METHOD: NONE
INITIAL STATE DEFINITION-
CHAIN: ch
   NODE LIST: statedep.waitbelow
   INIT POP: 1
RUN LIMITS-
   EVENTS: 20000
LIMIT - CP SECONDS:  10
TRACE: NO
END
```

10.4. Failures - FAIL1

10.4.1. Description

Represents failures by a separate job and a PRTYPR queue.

10.4.2. Distribution Parameters

- PROCTIME - process time distribution when tool is up.
- UPTIME - distribution of uptime.
- DOWNTIME - distribution of downtime.

10.4.3. Chain Parameters

- CH - chain parameter to be assigned to chain at higher level.

10.4.4. Initial State

One job must be initialized in the chain which contains node "invocation name".UP.

10.4.5. Submodel Listing and Diagram

```
SUBMODEL: fail1
   DISTRIBUTION PARAMETERS: proctime uptime downtime
   /* proctime - process time distribution when tool is up. */
   /* uptime - distribution of uptime. */
   /* downtime - distribution of downtime. */
   CHAIN PARAMETERS: ch
   /* ch - chain parameter assigned to chain at higher level */
QUEUE: upq
TYPE: FCFS
CLASS LIST: up
   WORK DEMANDS: uptime
QUEUE: processq
TYPE: PRTYPR
PREEMPT DIST:  1
CLASS LIST: process
   WORK DEMANDS: proctime
   PRIORITIES: 2
CLASS LIST: down
   WORK DEMANDS: downtime
   PRIORITIES: 1
CHAIN: ch
   TYPE: EXTERNAL
   INPUT: process
   OUTPUT: process
   : down -> up ;1.0
   : up -> down ;1.0
END OF SUBMODEL fail1
```

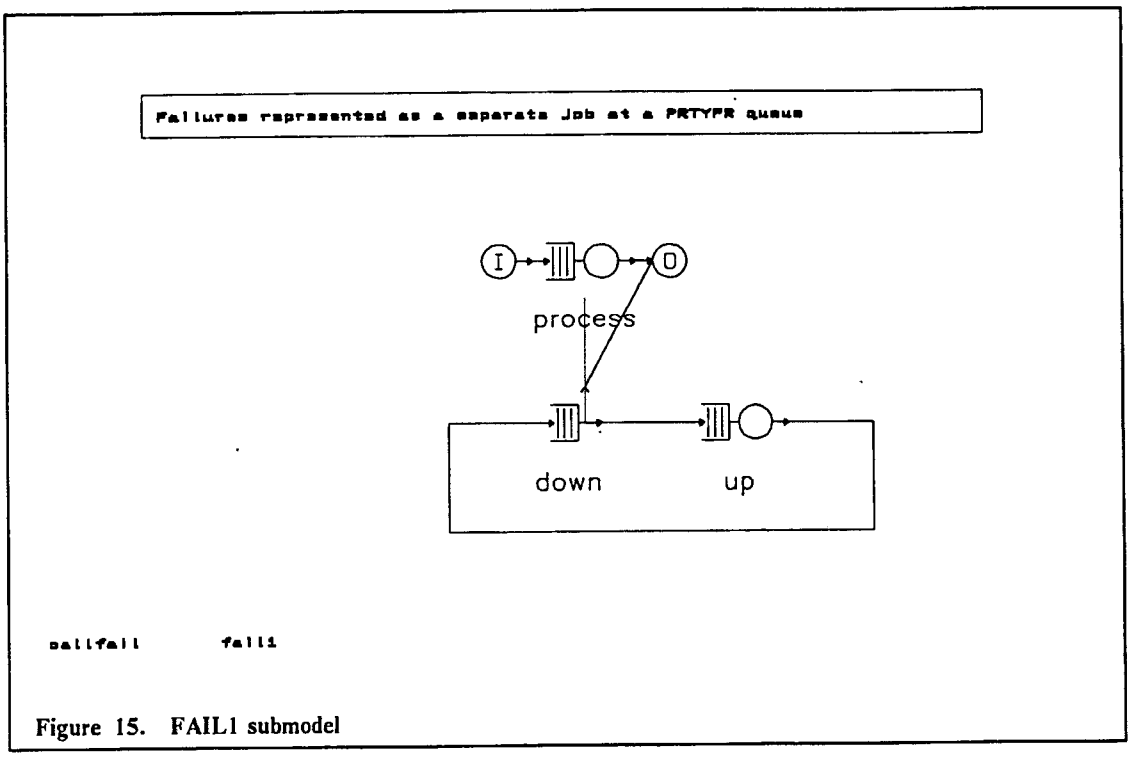

Figure 15. FAIL1 submodel

10.4.6. Model Which Invokes FAIL1

```
MODEL: callfail
   METHOD: SIMULATION
INCLUDE: fail1
INVOCATION: tool1
   TYPE: fail1
   PROCTIME: exponential(.2)
   UPTIME: exponential(20)
   DOWNTIME: exponential(.6)
   CH: ch
CHAIN: ch
   TYPE: OPEN
   SOURCE LIST: s
      ARRIVAL TIMES: 1
   : s -> tool1.INPUT;1.0
   : tool1.OUTPUT-> SINK ;1.0
CONFIDENCE INTERVAL METHOD: NONE
INITIAL STATE DEFINITION-
CHAIN: ch
   NODE LIST: tool1.up
   INIT POP: 1
RUN LIMITS-
   EVENTS: 100
LIMIT - CP SECONDS:  10
TRACE: NO
END
```

10.5. Finite Capacity Buffers - FINBUFW

10.5.1. Description

The following submodel represents a finite capacity queue with blocking using a wait node. This submodel can be used to model a pull system in a manufacturing line. This type of blocking also

occurs in computer systems and communication networks. It uses global variable TABUFFERQ, defined at a higher level, to communicate the number of buffers remaining at each machine which has a finite buffer.

10.5.2. Numeric Parameters

- NUMBUFS - number of buffer positions for an invocation of one workcell.
- NUMMACHS - number of machines for an invocation of one workcell.
- MACHINDEX - workcell index. Used to index TABUFFERQ.
- NEXTMACH - index of next workcell. A large value (999999) is used to indicate no buffer restriction at the next destination.

10.5.3. Distribution Parameters

- MACHST - service time distribution for machine processing.

10.5.4. Chain Parameters

- CH - chain parameter to be assigned to chain at higher level.

10.5.5. Global Variables

- TABUFFERQ is a vector defined at a higher level. It is used to keep track of the number of buffers remaining at each workcell in the model. Its dimension must be one greater than the number of workcells.

10.5.6. Submodel Listing and Diagram

```
SUBMODEL: finbufw
   NUMERIC PARAMETERS: numbufs nummachs machindex nextmach
   /* numbufs - no. of buffers for invocation of 1 workcell.*/
   /* nummachs - no. of machines for invocation of 1 workcell.*/
   /* machindex - workcell index. Used to index TABUFFERQ.*/
   /* nextmach - index of next workcell.                  */
   DISTRIBUTION PARAMETERS: machst
   /* machst - service time distrib. for machine processing.*/
   CHAIN PARAMETERS: ch
   /* ch - chain parameter to assign chain from higher level.*/
QUEUE: machineq
TYPE:  IS
CLASS LIST: machine
   WORK DEMANDS: machst
QUEUE: machinepq
   TYPE:  PASSIVE
   TOKENS: nummachs
   DSPL: FCFS
   ALLOCATE NODE LIST: almach
      NUMBER OF TOKENS TO ALLOCATE:  1
   RELEASE NODE LIST: relmach
QUEUE: bufferq
   TYPE:  PASSIVE
   TOKENS: numbufs+nummachs
   DSPL: FCFS
   ALLOCATE NODE LIST: albuf
      NUMBER OF TOKENS TO ALLOCATE:  1
   RELEASE NODE LIST: relbuf
SET NODE: settaa
   ASSIGNMENT LIST: tabufferq(machindex)=ta(bufferq)
SET NODE: settar
   ASSIGNMENT LIST: tabufferq(machindex)=ta(bufferq)
WAIT NODE: waitforbuf
   PREDICATE LIST: until(tabufferq(nextmach)>0)
```

```
CHAIN: ch
   TYPE: EXTERNAL
   INPUT: albuf
   OUTPUT: settar
   : almach -> machine ;1.0
   : machine -> waitforbuf ;1.0
   : waitforbuf -> relmach ;1.0
   : relmach -> relbuf ;1.0
   : albuf -> settaa ;1.0
   : settaa -> almach ;1.0
   : relbuf -> settar ;1.0
END OF SUBMODEL finbufw
```

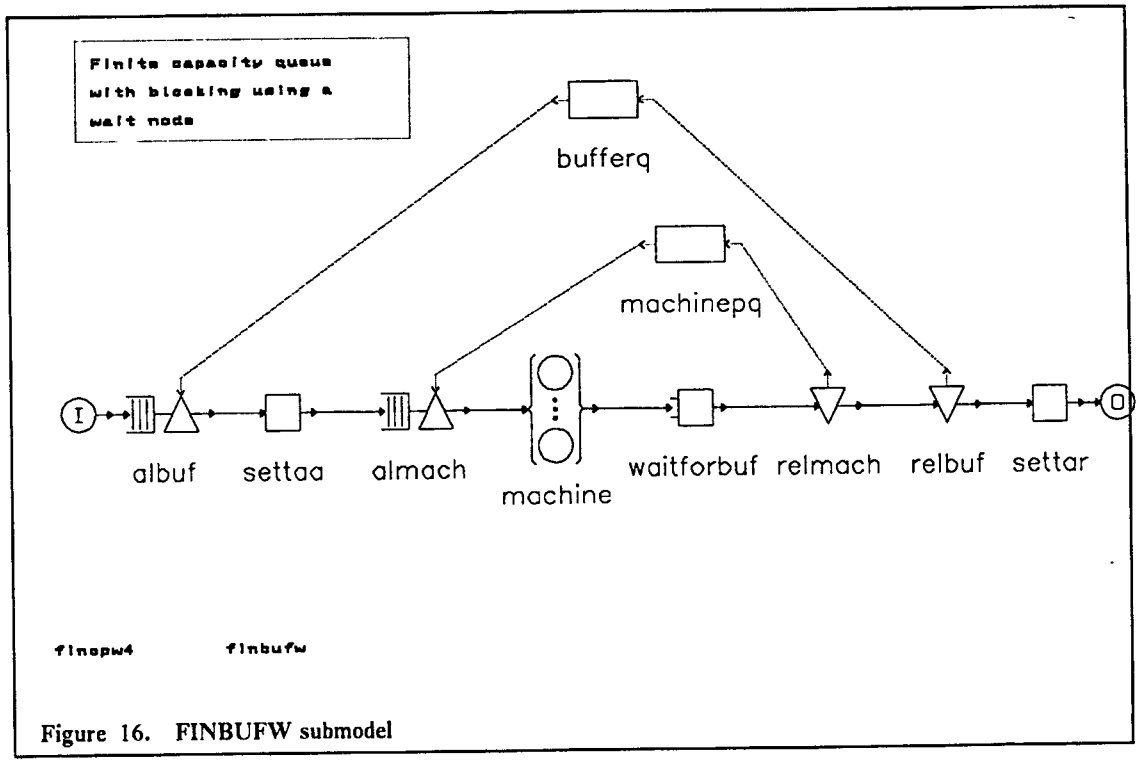

Figure 16. FINBUFW submodel

10.5.7. Model Which Invokes FINBUFW

```
MODEL: fincpw4
   METHOD: SIMULATION
   NUMERIC PARAMETERS: iat
   DISTRIBUTION PARAMETERS: mach1st
   DISTRIBUTION PARAMETERS: mach2st
   DISTRIBUTION PARAMETERS: mach3st
   GLOBAL VARIABLES: tabufferq(4)
      TABUFFERQ: 999999 2 2 999999
INCLUDE: finbufw
INVOCATION: machine1
   TYPE: finbufw
   NUMBUFS: 1
   NUMMACHS: 1
   MACHINDEX: 1
   NEXTMACH: 2
   MACHST: mach1st
   CH: ch
```

```
INVOCATION: machine2
   TYPE: finbufw
   NUMBUFS: 1
   NUMMACHS: 1
   MACHINDEX: 2
   NEXTMACH: 3
   MACHST: mach2st
   CH: ch
INVOCATION: machine3
   TYPE: finbufw
   NUMBUFS: 1
   NUMMACHS: 1
   MACHINDEX: 3
   NEXTMACH: 4
   MACHST: mach3st
   CH: ch
CHAIN: ch
   TYPE: OPEN
   SOURCE LIST: src
      ARRIVAL TIMES: iat
   : src -> machine1.INPUT ;1.0
   : machine1.OUTPUT -> machine2.INPUT ;1.0
   : machine2.OUTPUT -> machine3.INPUT ;1.0
   : machine3.OUTPUT -> SINK ;1.0
CONFIDENCE INTERVAL METHOD: REGENERATIVE
REGENERATIVE STATE DEFINITION-
CONFIDENCE LEVEL: 90
SEQUENTIAL STOPPING RULE: NO
RUN GUIDELINES-
   EVENTS: 20000
LIMIT - CP SECONDS: 30
TRACE: NO
END
```

10.6. Round Robin Scheduling - RRQUEUE

10.6.1. Description

This submodel represents Round Robin scheduling with overhead.

10.6.2. Numeric Parameters

- MEAN_SERVE - total mean cpu service time.
- QUANTUM - quantum or time slice.
- OVERHEAD - overhead for task switching.

10.6.3. Chain Parameters

- CHN - chain parameter to be assigned to chain at higher level.

10.6.4. Job Variables

- JV(0) is used to store the remaining cpu service time.

10.6.5. Submodel Listing and Diagram

```
SUBMODEL: rrqueue /*round robin queue*/
   NUMERIC PARAMETERS: mean_serve quantum overhead
   CHAIN PARAMETERS: chn
   QUEUE: q
      TYPE: fcfs
      CLASS LIST: cls
         SERVICE TIMES: constant(min(jv(0),quantum)+overhead)
   SET NODES: set_total
   ASSIGNMENT LIST: jv(0)=exponential(mean_serve)
   SET NODES: set_remain
   ASSIGNMENT LIST: jv(0)=jv(0)-min(jv(0),quantum)
   DUMMY NODES: dummy_out
   CHAIN: chn
      TYPE: external
      INPUT: set_total
      OUTPUT: dummy_out
      : set_total->cls->set_remain->cls dummy_out;if(jv(0)>0) if(t)
END OF SUBMODEL RRQUEUE
```

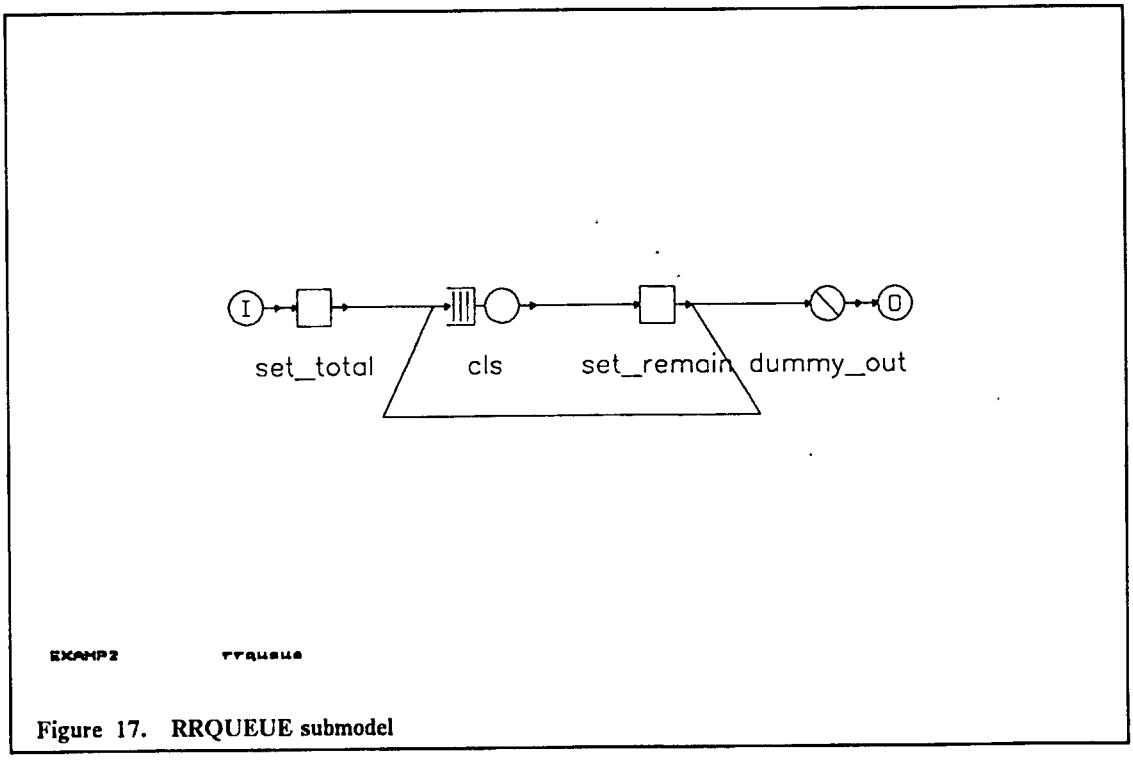

Figure 17. RRQUEUE submodel

10.6.6. Model Which Invokes RRQUEUE

```
MODEL: ral38f17
   METHOD: simulation
QUEUE: io1q
   TYPE: FCFS
   CLASS LIST: io1
      WORK DEMANDS: .22
QUEUE: io2q
   TYPE: FCFS
   CLASS LIST: io2
      WORK DEMANDS: .019
INCLUDE: rrqueue
```

```
INVOCATION: cpuq
   TYPE: rrqueue
   MEAN_SERVE: .05
   QUANTUM: .001
   OVERHEAD: .001
   CHN: interactiv
CHAIN: interactiv
   TYPE: closed
   POPULATION: 15
   : cpuq.output -> io1 ;.10
   : cpuq.output -> io2 ;.90
   : io1 -> cpuq.input ;1.0
   : io2 -> cpuq.input ;1.0
CONFIDENCE INTERVAL METHOD: REGENERATIVE
REGENERATIVE STATE DEFINITION-
CHAIN: interactiv
   NODE LIST: cpuq.cls
   REGEN POP: 15
   INIT POP: 15
CONFIDENCE LEVEL:  90
SEQUENTIAL STOPPING RULE: YES
   QUEUES TO BE CHECKED: cpuq.q
      MEASURES: qt
      ALLOWED WIDTHS:  10
SAMPLING PERIOD GUIDELINES-
   EVENTS: 50000
LIMIT - CP SECONDS:  30
TRACE: NO
END
```

10.7. Cyclic Server - SUBQ

10.7.1. Description

Submodel to represent a cyclic server queueing discipline. There will be NUMQS served by the cyclic server.

10.7.2. Chain Parameters

- CHN - chain parameter to be assigned to chain at higher level.

10.7.3. Global Variables

- NUMJOBS - number of active jobs in the model.

10.7.4. Job Variables

- JV(0) identifies the queue number that is being served.

10.7.6. Submodel Listing and Diagram

```
SUBMODEL: subq
   CHAIN PARAMETERS: chn
QUEUE: q1
TYPE: FCFS
CLASS LIST: c1
   WORK DEMANDS: .5
QUEUE: pq1
   TYPE:  PASSIVE
   TOKENS: 0
   DSPL: PRTY
```

```
   ALLOCATE NODE LIST: al11
      NUMBER OF TOKENS TO ALLOCATE: 1
      PRIORITIES: 1
   ALLOCATE NODE LIST: al12
      NUMBER OF TOKENS TO ALLOCATE: 1
      PRIORITIES: 2
   RELEASE NODE LIST: rel
   DESTROY NODE LIST: del
   CREATE NODE LIST: crl
      NUMBER OF TOKENS TO CREATE: 1
SET NODE: decnj
   ASSIGNMENT LIST: numjobs=numjobs-1
SET NODE: incnj
   ASSIGNMENT LIST: numjobs=numjobs+1
CHAIN: chn
   TYPE: EXTERNAL
   INPUT: crl
   OUTPUT: del
   : crl -> al12 ;1.0
   : al12 -> del ;1.0
CHAIN: chnopen
   TYPE: OPEN
   SOURCE LIST: sl
      ARRIVAL TIMES: 1
   : sl -> incnj ;1.0
   : incnj -> al11 ;1.0
   : al11 -> cl ;1.0
   : cl -> rel ;1.0
   : rel -> decnj ;1.0
   : decnj -> SINK ;1.0
END OF SUBMODEL subq
```

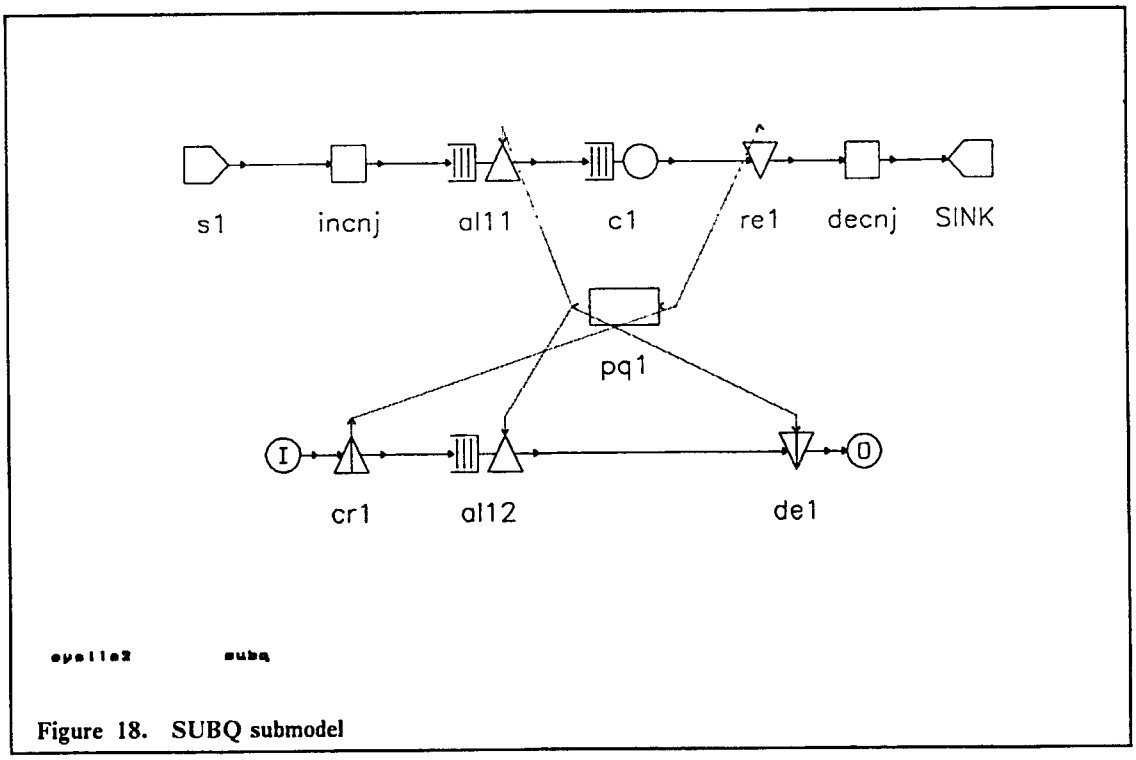

Figure 18. SUBQ submodel

10.7.7. Model Which Invokes SUBQ

```
MODEL: cyclic2
   METHOD: SIMULATION
   NUMERIC PARAMETER: numqs
   GLOBAL VARIABLE: numjobs
      NUMJOBS: 0
   NODE ARRAY: d1(numqs)
SET NODE: setinc
   ASSIGNMENT LIST: jv(0)=(jv(0)+1) mod numqs
WAIT NODE: w1
   PREDICATE LIST: until(numjobs>0)
DUMMY NODE: d1(*)
INCLUDE: subq
INVOCATION: inv1(numqs)
   TYPE: subq
   CHN: chn
CHAIN: chn
   TYPE: CLOSED
   POPULATION: 1
   : w1 -> d1(jv(0)+1) ;1.0
   : d1(*) -> inv1(*).INPUT ;1.0
   : inv1(*).OUTPUT -> setinc ;1.0
   : setinc -> w1 ;1.0
CONFIDENCE INTERVAL METHOD: REPLICATIONS
INITIAL STATE DEFINITION-
CHAIN: chn
   NODE LIST: w1
   INIT POP: 1
REPLIC LIMITS-
   EVENTS: 1000
LIMIT - CP SECONDS: 5
TRACE: NO
END
```

10.8. Unlimited Receiving Area - UNLIMSB2

10.8.1. Description

Submodel of an unlimited receiving area which produces a job on request.

10.8.2. Chain Parameters

- CH - chain parameter to be assigned to chain at higher level.

10.8.3. Submodel Listing and Diagram

```
SUBMODEL: unlimsb2
   /* Unlimited receiving area which produces a job on request. */
   CHAIN PARAMETERS: ch
   /* ch - chain parameter assigned to chain at higher level. */
SPLIT NODE: crjob
DUMMY NODE: src
DUMMY NODE: dout
CHAIN: ch
   TYPE: EXTERNAL
   INPUT: crjob
   OUTPUT: dout
   : crjob -> src ;SPLIT
   : crjob -> dout ;SPLIT
END OF SUBMODEL unlimsb2
```

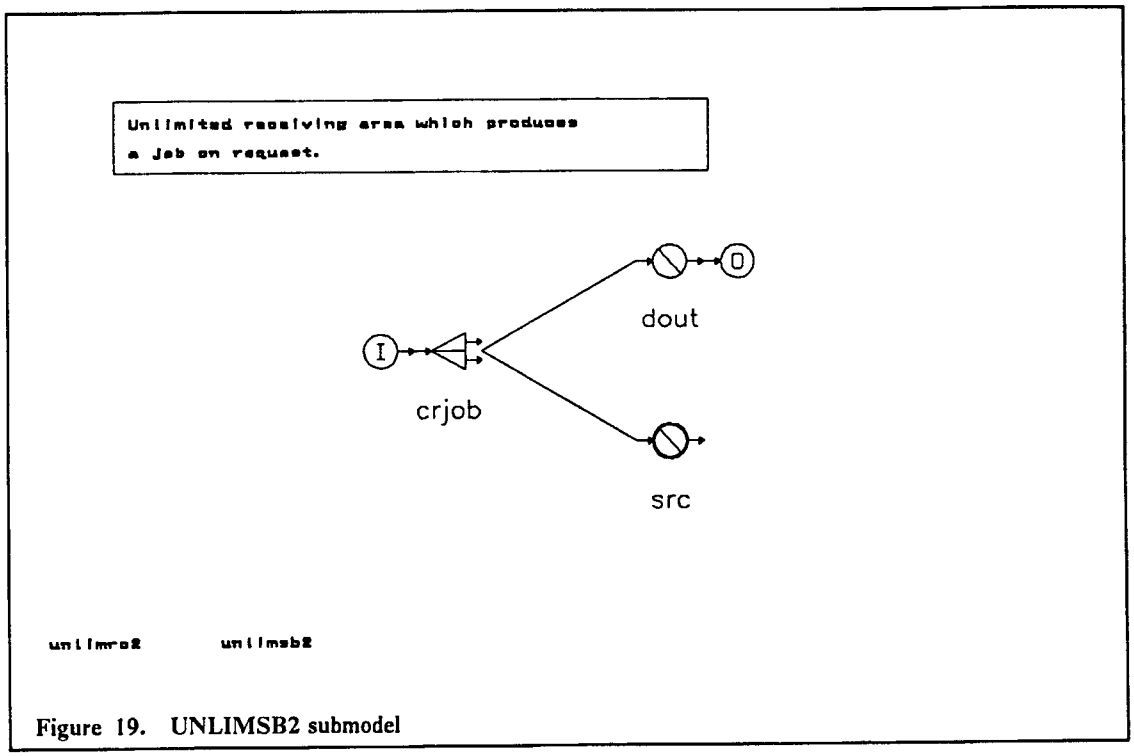

Figure 19. UNLIMSB2 submodel

10.8.4. Model with example

```
MODEL: unlimrc2
   METHOD: SIMULATION
QUEUE: q1
TYPE: FCFS
CLASS LIST: c1
   WORK DEMANDS: .2
QUEUE: waitnxtq
TYPE: FCFS
CLASS LIST: waitnxt
   WORK DEMANDS: 1.0
INCLUDE: unlimsb2
INVOCATION: unlim
   TYPE: unlimsb2
   CH: ch
CHAIN: ch
   TYPE: OPEN
   : waitnxt -> unlim.INPUT ;1.0   /* request next arrival */
   : unlim.OUTPUT -> waitnxt ;1.0
   : unlim.src    -> c1 ;1.0   /* next arrival */
   : c1 -> SINK ;1.0
CONFIDENCE INTERVAL METHOD: NONE
INITIAL STATE DEFINITION-
CHAIN: ch
   NODE LIST: waitnxt
   INIT POP: 1
RUN LIMITS-
   EVENTS: 1000
LIMIT - CP SECONDS:  10
TRACE: NO
END
```

11. The RESearch Queueing Package Modeling Environment (RESQME)

In this section, we describe the RESearch Queueing Package Modeling Environment (RESQME). This is a graphical workstation environment that allows the modeler to create, edit, run and analyze performance models. It provides access to all the underlying functionality of RESQ. The user interaction in specifying and analyzing the model takes place on the workstation, whereas the computation-intensive evaluation of the model can take place on the workstation for small models and pilot runs and on the host for large models at the option of the user. Detailed information on hardware and software requirements, installation procedure and user interaction is provided in "The RESearch Queueing Package Modeling Environment OS/2 Guide". Here we provide a general overview of RESQME.

RESQME presents a menu interface with direct manipulation of icons representing the RESQ elements to iteratively specify, run and analyze the model. You can create and edit queueing models by placing RESQ icons on a modeling display and specifying their attributes in pop-up windows. You can simulate the models by specifying the run parameters and selecting the evaluate menu items. You can view the performance measures graphically by pointing to the desired node(s) and specifying the form for the desired output graph. You can view the results both in performance charts and through the animation of jobs and tokens moving along the network diagram.

RESQME runs on a workstation connected to the host system or optionally stand-alone on a workstation. In this cooperative processing environment, all user interaction (specification and control) takes place on the workstation and the syntax checking and computation of the model results (evaluation) can take place on the host system or on the workstation, as desired by the user.

The modeling process is divided into three tasks represented by three submenus (task menus) in RESQME and corresponding to the experimental process described in Section 1. RESQME provides the capability for the user to move freely back and forth among these tasks to iteratively construct and analyze the model by using the menu items for Create/Edit, Evaluate and Output Analysis. The control of the three tasks is handled by the Main Menu.

The RESQME screen consists of the Main Menu and the appropriate task menus on the bottom left and a Screen Management Menu on the right border, with most of the screen used to display the model diagram. It provides a viewport into the modeling canvas containing the model diagram and to each separate canvas containing submodels. Each modeling canvas can hold a model diagram that is approximately fifty times the height and fifty times the width of the displayed portion. The Screen Management Menu allows the modeler to traverse this modeling canvas by panning, zooming, and locating model nodes, as well as to view submodel networks at each hierarchical level of the model. This provides the modeler with the ability to create realistic models in terms of both size of model and number of submodels. Each submodel is a logically separate entity and is graphically defined on its own modeling canvas. Submodels can be archived in a library and selected to be part of multiple models as well as shared among different users.

All graphical objects on the viewing surface, whether they are nodes in the model diagram or output charts, can be directly manipulated by the user to move, copy and delete the graphics and to edit and view their attributes. The textual attributes associated with an icon can be viewed and edited in a pop-up window.

All user interaction adheres to the following procedure. The interaction flow begins with the Main Menu. Selecting one of the three tasks from that menu causes the appropriate sub-menu for that task to also appear on the screen. All menu selections are accomplished by moving the graphic cursor to the desired item and pressing the selection button on the mouse. Selecting an item in a menu or a RESQ icon can require attribute specifications to be filled out in the attribute pop-up window. Examples of attribute specifications are the type, class list and service times for a queue icon, the probabilities for routing in a chain, the parameter values for a run, or the content and view specifications for an output chart.

The modeler can move freely between task and menu selections in RESQME and build the model in any order. We have organized the menus based on task and ordered the selections within task

based on the typical requirements when building, running and analyzing a model. The usual sequence to follow is outlined below, grouped by task, with an asterisk used to denote optional items.

1. Select an existing model or start a new model
2. Create/Edit
   a. Header
   b. Select icons, specify attributes, route
   c. Distributions *
   d. Simulation
   e. Tracing *
   f. Save
   g. Setup
3. Evaluate
   a. Set Parameters
   b. Execute *
   c. Exe Foreground
   d. Access Output
   e. Download Trace *
4. Output Analysis
   a. Specify Content
   b. Specify View
   c. Plot || Table
   d. Animation *
      1) Start Animation *

In addition to providing the functionality of RESQ in a visual interface for both specification and analysis, RESQME provides help and tutorial components. Pointing to an icon or menu item, the user can get context-sensitive help. Selecting playback of the tutorial section, the user can view a tutorial that explains specific aspects of RESQ. The tutorial shows the menu and icon selections, mouse movements and textual input as it demonstrates the selected request. Annotations can be used to further explain the action. Models developed by the tutorial component can then be used by the modeler in the same way as those he or she developed. Additionally, modelers can use the record mechanism to create their own tutorials.

RESQME allows the modeler to also extend the basic RESQ objects with a draw and link facility. Submodels can be created and stored and associated with user-drawn icons. The resulting icon is an encapsulation of the submodel and can be used in the same manner as the elementary icons in building models. Pre-tested code can thus be reused and shared. Modeling experts can develop submodels for use by others. Objects more representative of the application domain, such as submodels of a Round Robin queue or a cyclic server, can be used to build the final model. The modeler can link these user-created objects together at a higher level without having to develop them from scratch with elementary RESQ icons as shown in Figure 20 on page 63 below.

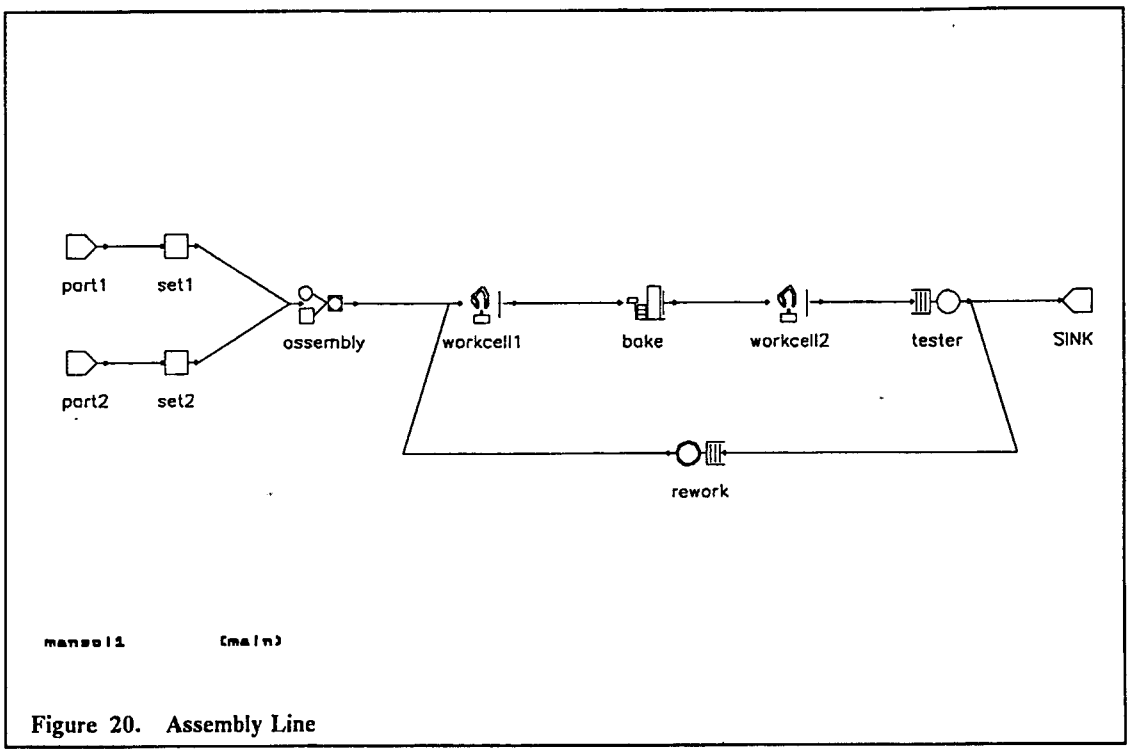

Figure 20. Assembly Line

Bibliography

1. J. Banks and J.S. Carson, II, *Discrete-Event System Simulation*, Prentice-Hall, Englewood Cliffs, N.J., 1984.
2. P. Bratley, B.L. Fox and L.E. Schrage, *A Guide to Simulation*, Springer-Verlag, New York, 1983.
3. G.J. Burkland, R.F. Gordon, E.A. MacNair, K.J. Gordon and J.F. Kurose, "A Dynamic Tutorial Facility For a Graphical Modeling Package," IBM Research Report RC-13670, Yorktown Heights, New York, April 1988.
4. K.J. Gordon, J.F. Kurose, R.F. Gordon and E.A. MacNair, "An Extensible Visual Environment for Construction and Analysis of Hierarchically-Structured Models of Resource Contention Systems," *Management Science, Vol.37*, No.6, June 1991.
5. R.F. Gordon, E.A. MacNair, P.D. Welch, K.J. Gordon and J.F. Kurose, "Examples of Using the RESearch Queueing Package Modeling Environment (RESQME)," *Proceedings of the Winter Simulation Conference*, Washington, D.C., December 1986, pp. 494-503.
6. R.F. Gordon, E.A. MacNair, K.J. Gordon and J.F. Kurose, "A Visual Programming Approach to Manufacturing Modeling," *Proceedings of the Winter Simulation Conference*, Atlanta, December 1987, pp. 465-471.
7. R.F. Gordon, E.A. MacNair, K.J. Gordon and J.F. Kurose, "Higher Level Modeling in RESQME," *Proceedings of the European Simulation Multiconference 1988*, Nice, France, June 1988, pp. 52-57.
8. R.F. Gordon, P.G. Loewner, E.A. MacNair, K.J. Gordon and J.F. Kurose, "The RESearch Queueing Package Modeling Environment (RESQME) OS/2 Guide for Version 4.1," IBM Research Report, Yorktown Heights, New York, April 1991.
9. J.F. Kurose, K.J. Gordon, R.F. Gordon, E.A. MacNair and P.D. Welch, "A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queueing Package (RESQ)," *ACM/IEEE Fall Joint Computer Conference*, Dallas, November 1986, pp. 719-728.
10. H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*, Addison-Wesley, 1978.
11. S.S. Lavenberg, ed., *Computer Performance Modeling Handbook*, Academic Press, New York, 1983.
12. A.M. Law and W.D. Kelton, *Simulation Modeling and Analysis*, Second Edition, McGraw-Hill, Inc., 1991.
13. E.A. MacNair and C.H. Sauer, *Elements of Practical Performance Modeling*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
14. C.H. Sauer and K.M. Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
15. C.H. Sauer and E.A. MacNair, *Simulation of Computer Communication Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1983.
16. C.H. Sauer, A.M. Blum, P.G. Loewner, E.A. MacNair and J.F. Kurose, "The Research Queueing Package Version 2: CMS Reference Manual," IBM Research Report RA-139, Yorktown Heights, New York, November 1986.

Index

-> 12

A

Abs 17
Accuracy 32
Acquire elements 11
ACTIVE 5, 10, 20
Active queues 2, 10, 13, 19, 20
Adding operators 18
ALLCTE 38
Allocate 11, 21
Allocate nodes 11, 21, 25
Analyze 61
AND allocate nodes 22
Animation 37, 39
Animation menu items 40
Animation options 39
Animation variables 39
Animations 4, 14, 61
Annotations 62
AQTRAC 38
Arithmetic expressions 9, 17, 19
Arrays 7, 16
Arrival rate 17
ARRIVAL TIMES: 12
Arrivals 13
ARRIVE 37, 38
Arrow 12
Assignment list 11
Assignment statements 11
Attribute information 4
Attribute pop-up window 61
Attribute windows 4
Attributes 61
Automated run length control 32
Averages 42

B

Batch process 13
Batch service 44
Batch sizes 44
Batches 13, 22
BATCHPRO 44
Blocking 53
Boolean constants 18
Boolean expressions 18, 19
Boolean factors 18
Boolean terms 18
Bottleneck 7
Branches 12
Branching probabilities 16
Breakdowns 20
Breakpoints 39
Buffers 21
Building blocks 16
Bulk arrivals 22, 46
BULKARR 46

C

CARRSUB 49
Ceil 17, 23
Ceiling function 23
Chain arrays 16
Chain definitions 12
Chain names 28
Chain parameters 28
Chain populations 6, 13, 16
Chain types 12, 28
Chain variables 16, 17, 43
Chains 6, 12, 13, 16
Charts 4, 61
Checkpoint files 40
Children 24
Classes 5, 9, 10, 20
  Multiple 5
CLOSED 12
Closed chains 6, 12
Codes 42
Coefficient of variation 19
Colors 39
Commands 9
COMPLT 38
Computations 61
Conditional arrivals 49
Conditions 25
Confidence interval widths 33, 34
Confidence intervals 2, 32, 34
Confidence levels 32, 33
Constant distributions 19
Construct 61
Contention for resources 1
Context-sensitive help 3, 62
Continuous distributions 23, 30
Cooperative processing 61
Copies 22
Copy 61
Correlations 35
CP SECONDS: 33
Cpu limit 33

Cpu seconds 13
Create 61
Create nodes 21, 30
Create/Edit 61
Current status 19
Cursor 61
Cv 19, 43
Cv's 16
Cv(0) 17
Cycles 34
Cyclic Server 57

D

Debug 39
Debugging 37
Decision making 32
Decision points 25
Default distributions 19
Delays 10
Delete 61
Departures 39, 42
Dependencies 35
Dependent jobs 24
Destroy nodes 21, 30
Direct manipulations 61
Discard 33
Discrete 19
Discrete distributions 11, 19
Distribution 42
Distribution identifiers 16
Distribution parameters 16
Distributions 5, 10, 11, 12, 16, 19, 23
  Constant 19
  Default 19
  Discrete 11, 19
  Erlang 19
  Exponential 5, 10, 12, 19
  Hyperexponential 19
  Standard 19
  Uniform 16, 19, 23
Dotted lines 21
Draw 62
DSPL: 10
Dummy nodes 25
Dynamic tutorial facility 3

E

Edit 61
Elementary icons 62
Elements 11, 61
Environments 61
Equilibrium 34, 35
Equilibrium behavior 32, 33
Equilibrium characteristics 32
Erlang distributions 19
Errors 32
Estimates 7, 32
Evaluate 61
Evaluate menu items 61
Evaluations 7, 14, 61
Event counts 39
Event handling trace 37, 38
Event list trace 37, 38
Event times 39
Events 13, 37
Execution time 16
Executions 4, 7
Exp 17
Experimental process 3, 61
Exponential 19
Exponential distributions 5, 10, 12, 19
Exponential stages 34
Expressions 17
EXPRT 37, 38
Extended queueing networks 2
EXTERNAL 28

F

F 18
Factors 17
Failures 51
FAIL1 51
False 18
Families 24
FCFS 10, 11
Final queue lengths 43
FINBUFW 52
Finite capacity buffers 53
Finite capacity queues 53
Finite number of elements 11
First available 25
First-Come-First-Served 10
FISSION 24, 25
Fission nodes 20, 22, 24
Fixed population 12
Fixed rate servers 10
Fixed service rates 20
Floating point 18
Floor 17

Flow of jobs 12
Formatted results 4, 7
Forms 61
FROM nodes 12
Fromm, Hans 44
Fusion nodes 22, 24

G

Global consistency checks 7
Global variables 13, 17, 22, 23, 25, 28, 43, 53
Grammar 17
Graphical level 4
Graphical objects 61
GV 43

H

Header 6, 13, 16
Help 3, 62
Hierarchical levels 61
Hierarchies of jobs 24
Hold elements 11
Hyperexponential distributions 19

I

Icons 5, 6, 11, 21, 22, 24, 61, 62
Identifiers 5, 7, 17
If 18
Inaccuracies 32
INCLUDE: 44
Increments 13
Independent and identically distributed 35
Independent jobs 22
Independent replications 32
Indexes 9
Infinite population 12
Infinite server 6, 7, 10
Initial populations 13
Initial portion discarded 13
INITIAL PORTION DISCARDED: 33
Initial state definitions 13, 35
Initial states 33, 34, 35
Initial system state 33
Initial transient phase 33
Initialization states 33
Input nodes 28
INPUT: 27
Interactive syntax checker 4
Interarrival time distributions 12
Interrupt 39
Intervals 32
Introduction 1
Invocations 26, 28
INVOCATION: 28
Invokes 26
IS 6, 10
I.i.d. 35

J

Job attributes 9
Job copies 20, 22, 37, 38
Job flows 12, 22, 37
Job movement trace 37
Job movements 39
Job numbers 37, 38
Job routings 12
Job traces 39
Job types 10, 11, 12
Job variables 9, 11, 12, 16, 17, 22, 24, 38, 43
Jobs 1, 5, 9, 13, 61
Jobs on request 59
JV 43
Jv's 9, 16

K

Keywords 23

L

Last-Come-First-Served 20
Layer 39
LCFS 20
Length 39
Levels of accuracy 34
Libraries 29
Library 61
Linear times 39
Little's Rule 8
Ln 17
LNG 43
Locating 61
Log function 17
Logical and 18
Logical negation 18
Logical or 18
Long run 42
Longest-Remaining-Time-First 20
Lower limit 19
LRTF 20

M

Macro definitions 26
Main menu 61
Matrices 16, 17
Max 17
Max cv's 16
Max jv's 16
Maximum functions 22
Maximum number 16
Maximum queue length 42
Maximum queueing time 42
Maximum values 30
Mean 19
Mean values 19
Memory 3, 11, 14
Memory partitions 9
Menu interfaces 61
Merge 22
Min 17
Model diagrams 37, 40, 61
Model header 6, 9, 13, 16
Model structure 26
Modeling canvas 61
Modeling elements 9, 16
Modulo function 18
Mouse 61
Move 39, 61
Moving pictures 37
Multi-server queues 10
Multiple assignment statements 22
Multiple classes 20
Multiple conditions 22
Multiple destinations 25
Multiple servers 20
Multiplying operators 18
Multiprocessor 10
MXQL 42
MXQT 42

N

Names 5
Natural log 17
ND 42
Network diagrams 61
New users 9
No waiting 10
Node arrays 16
Node departure counts 33
Nodes 1, 5, 6, 9, 11, 13, 61
Non-preemptive priorities 20
NONE 13
Not 18
Number of buffers 25, 53
Number of chain variables 16
Number of chains 16
Number of customers 20
Number of cv's 16
Number of departures 13, 39, 42
Number of events 13
Number of job variables 16
Number of jv's 16
Number of replications 33
Number of servers 10
Number of tokens 11, 21
Numeric functions 17, 22
Numeric identifiers 9, 11
Numeric parameters 9, 12, 16, 23
Numeric values 18
Numerical precision 18
Numerical solution 1, 5, 7, 32

O

Offspring 24
OPEN 12
Open chain population 42
Open chain response time 42
Open chains 6, 12, 13, 17
Operators 18
OR allocate nodes 22
Or operator 22
Output analysis 14, 61
Output nodes 28
OUTPUT: 27

P

Packets 25
Pan 39
Panning 61
Parallel activities 24
Parameter estimation 32
Parameter values 14, 26, 28
Parameters 4, 7, 9, 14
Parenthesized expressions 17
Parents 24
PASSIVE 5, 11
Passive queues 11, 19, 21
Passive resources 2, 5, 11, 21
Paths 5, 6, 12
Pending events 38
Performance measures 1, 8, 30, 32, 42, 61
Plots 4, 14
PO 42
Point estimates 32
Pointer files 40
Polish notation 38

Pool 30
Pool of tokens 11, 21
Pop-up windows 4, 61
Populations 6, 12, 13, 42
PQTRAC 38
Pre-tested code 29
Precision 18
Predicate notation 18
Predicates 12, 17, 18, 22
Preemptive priorities 20
Primitive elements 17, 18
PRINT 17
PRINT function 17
Priority 20
Priority levels 10, 20, 21
Probabilities 1, 6, 7, 12, 16, 19, 22, 30, 32
Probability distributions 13, 16, 17, 19, 22
Processor Sharing 5, 20
PRTY 20
PRTYPR 20, 51
PS 5, 20
Pull systems 53
Pure delays 10
Pushdown stacks 20

Q

QL 20, 42
Qld 30, 42
QT 42
Qtd 30, 42
Qualifications 28
Qualified 29
Queue definitions 21
Queue departure counts 33
Queue dependent service rates 20
Queue length 1, 42
Queue length distribution 42
Queue length distributions 30
Queue lengths 7, 20, 39
Queue trace 37, 38
Queueing disciplines 5, 9, 10, 20
Queueing networks 1
Queueing time 42
Queueing time distribution 42
Queueing time distributions 30
Queueing times 3, 7
Queues 5

R

Random number streams 13, 32
Random numbers 13
Random values 19
Random variables 13
Range of values 19
Rates 10
RATES: 10
Readability 9
Reassembled 25
Reassembly 24
Rebatch 25
Record mechanism 62
Regeneration cycles 34
Regeneration states 32, 34
Regenerative characteristics 32
Regenerative method 32, 34
Related jobs 20, 22
Relational expressions 18
Relatives 20, 24
RELEAS 38
Release 11
Release nodes 11, 21
REPLIC LIMITS- 33
Replications 33
Requests 11
Resource elements 11
Resources 1
Response time 42
Response times 3, 7
RESQME 3, 61
Restart 39
Results 14, 32, 61
Reuse 29
RJ 20
Round Robin 55
Round Robin scheduling 20
Routing 5, 6, 10, 22, 25
Routing decisions 1, 9, 11, 12, 25
Routing definitions 18
Routing predicates 17, 19, 25
Routing transitions 6, 12
RRQUEUE 55
RTM 42
Run length control 13, 34
Run length templates 13
Run limits section 13
Run parameters 61
Runs 14

S

SA 19
Sampling periods 34
Scalar values 19
Scalars 16, 17
Scenarios 14
Scheduling algorithms 5, 9
Screen management menus 40, 61
SDQL 42
SDQT 42
Seeds 13, 33
Selection buttons 61
Selective animation 39
Sequential stopping rules 34, 35
Server definitions 10
Servers 5, 9, 10, 19, 20
Servers available 19
Service 9
Service centers 2, 5, 9, 20
Service completions 13
Service rates 5, 10, 20
  Fixed 5
  Queue dependent 20
Service time distribution 5, 10
Service times 1, 5, 7, 10, 43
Set nodes 11, 13, 22
Shortest-Remaining-Time-First 20
Signed factors 17
Simulated times 13, 33
Simulation 2, 9
Simulation dependent 17
Simulation estimates 32
Simulation times 39
Simultaneous resource possession 2, 11
Single run 34, 35
Single server queues 5
SINK 24
Sinks 6, 12
Sizes 39
Snapshot trace 37, 39
Snapshots 40
Solution methods 9
SOURCE LIST: 12
Sources 6, 12
Spectral method 32, 35
Speeds 39
SPLIT 23, 24
Split nodes 22
SRTF 20
ST 43
Stand-alone 61
Standard 19
Standard deviation of queue length 42
Standard deviation of queueing time 42
Standard deviations 19
Standard distributions 19
State-dependence 49
States 33
Statistical output analysis 2
Statistical variability 32
Statistically identical 34
Status functions 17, 19
Steady state 13
Stochastic 13
Stopping conditions 13
Stopping rules 2
Submenus 61
Submodel headers 27
Submodel inputs 29
Submodel libraries 29
Submodel names 27, 28
Submodel outputs 29
Submodels 26, 44, 61
Subnetworks 26
SUBQ 57
Subscripts 9
Subsets of chains 39
Subsets of nodes 39
Subsets of queues 39
Symbolic names 9, 13, 16
Synchronized processes 24
Synonyms 28
Syntactic errors 4
Syntax checking 61
System resources 9

T

T 18
TA 19
Tasks 61
Templates 4, 13, 14, 26
Terminate 13
Terms 18
Textual attributes 61
Textual versions 12
TH 20
Think time 1, 9
Throughput 7, 42
Time 9
Time slice 20
Time units 5
TO nodes 12
Token flow 37
Token movements 39
Token paths 21
Tokens 11, 19, 21, 22, 24, 30, 42, 61
Tokens available 19, 38

Tokens held 20, 37, 38
Tokens in pool 38
Tokens in pool distribution 42
Tokens in use 42
Tokens in use distribution 42
Tokens in use distributions 30
Tokens requested 38
Total queue 20
Total tokens distributions 30
Total tokens in pool 42
TP 42
TQ 20
Trace 37
Trace files 39
Trace menu items 40
Transfer nodes 22
Transient behavior 33
Transient characteristics 32
Transitions 6, 12
True 18
True values 32
TT 42
Ttd 30, 42
TU 42
Tud 30, 42
Tutorials 62
Type 5, 6, 12
  Chains 6, 12
  General 5
  Queues 5
  Specialized 5
Type of chains 12
Types of jobs 10, 11, 12, 20
TYPE: 28

U

Unbatch 25
Uniform 19
Uniform distributions 16, 19, 23
Unlimited receiving areas 59
UNLIMSB2 59
Unsigned numbers 17
Until conditions 25
Upper limit 19
USER function 17
User-drawn icons 62
Users 1, 6, 9
UT 42
Utilizations 7, 14, 42

V

Values 5, 7, 11, 12, 13, 16, 22, 24, 30, 43
Variables 11, 12, 43
Vectors 9, 16, 17, 53
Viewports 61
Views 61
Visual interfaces 62

W

Wait 11
Wait nodes 18, 25, 53
Waiting 10
Waiting areas 21
Waiting lines 5, 10, 20
Waiting times 7
Warning conditions 34
Width criteria 34
Work demands 5, 10, 20
Workstations 61

Y

YES 35

Z

Zero 7, 9
Zoom 39
Zooming 61