Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

Implementation parallelized queueing network simulations using FORTRAN and data abstraction

Keith W. Miller
David M. Nicol
Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185

ABSTRACT

Researchers experimenting with simulation on novel computers with parallel architectures must contend with numerous disadvantages: limited availability of programming tools, unique synchronization problems when implementing the simulation, and unexpected interactions between elements of the simulation and characteristics of the underlying computer. Since FORTRAN is likely to be available on the computer and familiar to the researcher, it may be the most logical candidate for coding the simulation. However, FORTRAN does not enforce data abstraction, a useful tool in specifying, designing, and implementing a simulation effort. FAD is a FORTRAN preprocessor which allows the encapsulation of user-defined abstract data types. Data abstraction and FAD is illustrated in the simulation of a queueing network. The separation of concerns between a simulation expert ("the user") and an expert in exploiting a parallel architecture ("the implementor") has benefits for both.

1. INTRODUCTION

Despite many alternatives, FORTRAN remains a much-used language in simulation. Its advantages include wide availability, well developed compilers, and large subroutine libraries for many applications. The specialized nature of constructing simulations that execute on certain parallel architectures may well be restricted to a widely available general purpose language like FORTRAN, because only such languages are supported on many of these machines.

FORTRAN is an old language, and lacks many facilities that would be useful in large simulation programming projects. Because of FORTRAN's drawbacks, some researchers have counseled using other general purpose programming languages (e.g., L'Ecuyer and Giroux (1987) suggest Modula-2), preprocessors (GPSS, SIMSCRIPT, SLAM, and so on), and large collections of FORTRAN libraries. Each of these suggestions has advantages and disadvantages (for a lively presentation of the issues, see Brately, Fox and Schrage (1983)), but for many parallel machines, such arguments are moot - FORTRAN may be the only or one of a few high level languages available. In addition, FORTRAN has some advantages in programming the minute details essential to implementing complex algorithms on parallel architectures. Unfortunately, it is precisely these details that make the resulting code difficult to write, read, and debug, especially when the details of parallel implementation are interleaved with details of the simulation application.

1.1. FORTRAN and Abstract Data Types for Simulation

We propose to combine the utility and familiarity of FORTRAN programming with the information hiding features of more modern programming languages to facilitate high level programming of simulations that are to be executed on parallel architectures. Our approach is to implement high level data objects, called abstract data types, and then make these implementations available in a controlled FORTRAN programming environment that enforces information hiding. The goal is to spare simulation modelers from the intricacies of parallel architectures by constructing a software interface that hides these details of implementation. When this interface is properly specified and implemented, a simulation modeler can concentrate on simulation problems instead of parallel algorithms. On the other side of the information-hiding barrier, the expert in parallel programming can use the interface as a precise guide to the services required by simulation users.

There are a plethora of simulation languages available, many of which are FORTRAN-based. For a recent survey, see Kreutzer (1986). Before discussing yet another strategy for simulation programming, it seems reasonable to defend its creation. Our emphasis on using FORTRAN and abstract data types is intended to give simulation programmers detailed control without artificial restrictions. When using pre-packaged simulation languages or collections of subroutines, the programmer may be forced to use inappropriate tools because the task at hand was not anticipated by the authors of the language or subroutines. This problem is especially acute when experimenting with novel computer architectures, where familiar "tricks of the trade" in simulation programming no longer apply. In this paper, we describe a technique for building new tools, as the need arises, without confusing the tools with the goals of the simulation itself. Many general purpose languages could be used with this programming strategy, but in this paper we focus on FORTRAN because of its wide availability and the many experienced FORTRAN programmers involved in simulation.

An encapsulated abstract data type (ADT) is a collection of data declarations and routines (subroutines and functions in FORTRAN) that manipulate instances of the abstract data type. The user of an ADT can declare instances of the encapsulated type and can access these instances using the ADT routines. However, the user cannot access the instances directly - the implementation details, including the data structure and algorithms used, are hidden from the user. The ADT concept is closely related to work by Parnas (1979) on information hiding. An Ada package, a SIMULA class, and a CLU cluster are all language facilities that have been used for data type encapsulation (Ghezzi and Jazayeri 1987).

FORTRAN does not allow a programmer to encapsulate new data types. Algorithms can be hidden by distributing object code without source code, but in order to use a subroutine or function a programmer must know in precise detail how each piece of data is represented. This distribution of representational information defeats one of the major goals of ADTs - the localization of implementation information. This localization is particularly useful when implementation details are likely to change. In some simulation applications, such change is an essential part of a project. In this paper we will concentrate on one such application, parallelized simulations of queueing networks.

1.2. An Outline of the Paper

In the following section we introduce a FORTRAN preprocessor, called FAD (FORTRAN Abstract Data). FAD enables a programmer to create new data types by storing data declaration information and routines in a database. Once an ADT is defined in FAD, FORTRAN programmers can declare instances of this type in a manner similar to traditional FORTRAN declarations. They can also manipulate these instances via calls to the FAD routines. However, the preprocessor prohibits any direct manipulation of instances of FAD ADTs. Using the FAD preprocessor, a FAD implementor encapsulates these new data types for FORTRAN programmers. (For more details about FAD, see Miller, Morell, and Stevens (1989)). The section describes how a FAD user might describe, at a high level of abstraction, the behavior of a queueing network using several ADTs and ADT operations.

In the third section we discuss the implementation of the ADTs that specify a queueing network. The implementation process could be repeated for any environment that includes a FORTRAN compiler. Serial machines, shared-memory or distributed-memory multiprocessors could all support the same high level ADTs. When the ADTs are properly implemented, the FAD user's software could run on all these machines without changing the code that manipulates the ADTs. Any differences in the underlying machines and the algorithms that take advantage of these differences would be encapsulated in the FAD implementations.

The final section of this paper discusses the implications of FAD for other simulation applications, and suggests future research directions. Three appendices give examples of ADT routines.

2. DESCRIBING A QUEUEING NETWORK

We propose to separate the task of programming a parallelized simulation of a network of queues into two distinct phases:

(1) Specifying the behavior of the network without regard to the data structures used to represent the network or the architecture on which the network will be simulated.

(2) Coding the implementation details necessary to realize the behaviors defined in step 1.

The specification of the network behavior will be assigned to a "FAD user" who exploits ADTs made available by the "FAD implementor." The FAD implementor must determine how to deliver the required ADT behavior using the FORTRAN virtual machine and the underlying hardware available in a particular environment. Furthermore, the implementor has total control over such decisions as data structures, scheduling algorithms, and the like. However, the FAD implementor need not be concerned about any details of the particular network being simulated.

The rest of this section describes a set of ADTs that can be used to specify a network of queues. For the sake of brevity, we will restrict our attention to a limited subset of possible queueing networks. Although this subset has its limitations, its specification and implementation provides a realistic demonstration of the utility of the proposed techniques.

The ADTs and associated routines are described here in English. However, an advantage in using ADTs is that more formal specifications can be produced. For more information concerning the formal specification of ADTs, see Parnas (1977) and Guttag and Horning (1978).

2.1. A Model for Queueing Network Simulations

We propose the following model for the high level definition of a queueing network simulation:

(1) A network is made up of sources, queues, and directional links.

(2) Each source is connected to at most one queue. A queue may be linked to other queues or with a link that loops back to the queue itself. (A queue that has no links to other queues is a special case called a "sink.")

(3) Sources generate jobs according to a distribution. These jobs move through the network along the directional links. The creation time of a job is noted.

(4) When a job arrives at a queue, its arrival time is noted, it is assigned a service time according to a distribution, its destination after servicing is determined according to another distribution, and then the job is enqueued.

(5) The next job to be serviced is determined by the queueing discipline. After servicing, the job is sent to its next destination (if one exists) and the job is removed from the queue. The exit time is recorded. The exit time from this queue is equal to the arrival time in the destination queue.

There are several elaborations that would be useful for particular simulations. For example, this model assumes no communication time for moving a job from a source or queue to a queue. No blocking queues are allowed. We will also ignore the possibility of priority jobs, load dependent service times, and preemption in this discussion. However, our simple model will illustrate our techniques of FAD implementations, and elaborations can be added without invalidating the techniques.

If we disregard implementation issues, we can outline a straightforward process for specifying a particular queueing network simulation:

(1) Specify the number of sources and the distribution each will use to generate jobs.

(2) Specify the number of queues, the links between each source and a queue, and the links between the queues.

(3) For each queue, specify a distribution for determining service times and a distribution for determining destinations.

(4) For each queue, specify the queueing discipline.

The word "specify" above refers to definitions of objects and behaviors. Notice that these four steps say nothing about *how* the simulation will be implemented: data structures, number of processors, and communications protocols are ignored. These details are all essential to the implementation of the simulation, but are not essential to the definition of the the simulation itself. Therefore, we assign to the FAD user the specification of these four steps, and we assign the implementation of high level queueing data structures and operations to the FAD implementor.

2.2. Specifying the Network Using ADTs

We define four ADTs that define the architecture of the network: RANDOM DISTRIBUTIONS (RNDDIST), SOURCES, QUEUES, and NETWORK. Figure 1 describes each of the ADTs.

The ADTs in Figure 1 can be used to specify the first three aspects of a network listed above. However, the queueing protocol will require another ADT for jobs in the system, and several operations for examining and manipulating the jobs in a queue. The ADT for jobs and the operations are part of the software interface supplied by the FAD implementor. Figure 2 includes the JOB ADT and examples of the operations that involve jobs in a queue. Appendix A includes several other operations.

The operations in Figure 2 emphasize the high level nature of the FAD users access to the network. Because the FAD user does not know how a job is to be represented, explicit routines are provided for accessing the information in a job (e.g., SETARR, GETARR). The ordering of a queue is not known to the user, so routines are provided that identify jobs with various properties (e.g., FIRST, LONG). Many of these access routines turn out to be trivial to implement when a representation is chosen, so in order to improve the efficiency of the eventual code, FAD allows an implementor to specify in-line FORTRAN code substitutions for FAD routine calls. (For further explanation of FAD in-line substitution, see Miller, Morell, and Stevens (1989).)

Implementation details of how simulation time is managed can involve complex issues of synchronization, lookahead, rollbacks, and the like (see Nicol (1988)). The FAD user does not contend with these details. Instead, the FAD user requests that a certain event occurs at a certain time (e.g., STARTS). Each FAD implementation will determine how that request is realized. Notice that this separation of concerns has conceptual appeal and practical advantages: the FAD user can concentrate on the clear, concise specification of the desired protocol, uncluttered by myriad implementation and coding details. On the other hand, the FAD implementor can concentrate on generally useful code that embodies sophisticated techniques that would be more difficult to describe when tied to a particular application.

ADT: RANDOM DISTRIBUTION (RNDDIST)
DECLARATION: RNDDIST R
INITIALIZATION: SUBROUTINE RNDINI(R,DST,RND,SEED)
This subroutine initializes the random distribution R. RND is a random number generator function and DST is a distribution function. SEED is used to initialize the random number function. The FAD user can either write DST and RND or can use functions available in the FAD database. In either case, an initialized RND variable is used by the implementation as a stream of random values.

ADT: SOURCES
DECLARATION: SOURCES SRCS(n)
where n is the number of sources to be defined.
INITIALIZATIONS:
SUBROUTINE SRSINI(SRCS, WHEN)
An array of sources, SRCS, is initialized with a distribution function, WHEN. WHEN controls how often each source generates jobs in the network.
SUBROUTINE SRINI(SRC, WHEN)
Identical to SRSINI, except that only one source, SRC, is initialized.

ADT:QUEUES
DECLARATION: QUEUES QS(N)
N is the number of queues in the network.
INITIALIZATIONS:
SUBROUTINE QSINI(QS, SRVDST, OUTDST)
The array of queues, QS, is initialized with two distribution functions, SRVDST and OUTDST. SRVDST ("service distribution") controls the service time. Because a queue may have several different queues to which it can send a completed job, OUTDST ("output distribution") determines the destination of an outgoing job.
SUBROUTINE QINI(Q, SRVDST, OUTDST)
Identical action to QSINI, except that only one queue, Q, is initialized.

ADT: NETWORK
DECLARATION: QNET N(MAXSRC, MAXQS, MAXJOB)
MAXSRC gives the maximum number of sources that can be included in the network N. MAXQS gives the maximum number of queues that can be included in the network N. MAXJOB gives the maximum number of jobs that can be present in the network N at any given time.
INITIALIZATION:
SUBROUTINE NETINI(N, SRCS, QS)
The network N is initialized with the sources in the array SRCS and the queues in QS. Initially, there are no links in the network, only nodes.
OPERATIONS:
SUBROUTINE S2QLNK(N, S, Q)
An output link from a source S to a queue Q is placed into the network N.
SUBROUTINE Q2QLNK(N, Q1, Q2)
An output link from queue Q1 to queue Q2 is placed into the network N.
SUBROUTINE GOSIM(N, TLAST)
This subroutine initiates the simulation defined by the network N. TLAST gives the time limit after which the simulation halts. We assume that the simulation begins at time 0.0, and that TLAST is a real.

Figure 1. Four ADTs for Specifying a Network of Queues.

ADT: JOB
DECLARATION: JOB J
INITIALIZATION:
a job is always initialized by the FAD implementation.
OPERATIONS:
SUBROUTINE SETARR(J)
Set the arrival time of the job J into its current queue.
REAL FUNCTION GETARR(J)
Returns the arrival time of the job J into its current queue.
SUBROUTINE FIRST(J, Q)
Identifies the job that has been in the queue Q the longest, and copies it into the ADT job variable J (first come, first served).
SUBROUTINE LONG(J, Q)
Identifies the job in Q that has the longest service time, and copies it into J.
SUBROUTINE STARTS(J, Q, T)
Requests the simulation to start servicing a job J in queue Q at time T.
SUBROUTINE ENDS(J, Q, T)
Requests the simulation to stop servicing a job J in queue Q at time T.

Figure 2. An ADT for the jobs in a queue.

When specifying the queueing discipline, the FAD user describes three event handlers and a subroutine for selecting the next job to be serviced from a queue. Figure 3 lists the four high level routines.

SUBROUTINE JOBARR(J, Q)
Describes the event where a job J arrives at a queue Q.
SUBROUTINE ENTSRV(J,Q)
Describes the event where a job J enters service in a queue Q.
SUBROUTINE JOBXIT(J, Q)
Describes the event where a job J leaves a queue Q after being serviced.
SUBROUTINE GETNXT(Q, J)
Given a queue Q, GETNXT identifies which job in Q should be serviced next. A copy of that job is placed into J.

Figure 3. Routines for specifying a queueing discipline.

The FAD database includes defaults for these four routines. The default routines define the first-come-first-served queueing discipline. When defining other queueing disciplines, some of the defaults may remain unchanged. For example, the queueing disciplines of longest-service-time first, shortest-service-time-first, and longest-in- network-first can all be implemented by changing the GETNXT subroutine but using the defaults for JOBARR, ENTSRV, and JOBXIT. Appendix B shows code for the four default subroutines. This code is written at the same level of abstraction required of a FAD user. This code includes calls to exception handling routines that are described in the next subsection.

2.3. Exception Handlers

There are several error conditions that may occur when a simulation is executing. For each anticipated error condition, the FAD implementor includes a default subroutine that the implementation calls when the error occurs. If a FAD user wants a different response to a particular error, s/he writes a subroutine with the same name as the default error handler. The user subroutine takes precedence over the FAD routine with the same name. Figure 4 includes examples of error handlers; more are shown in Appendix C.

SUBROUTINE NETOVR(N, S, J)
Called when source S produces job J in net N, but N is already at its capacity.
SUBROUTINE BADQ(Q)
Called when a FAD user refers to an uninitialized queue, Q.
SUBROUTINE GETMT(Q)
Called when GETNXT is called with an empty queue, Q.

Figure 4. Examples of error handling routines.

3. IMPLEMENTING QUEUEING NETWORK ADTS

The FAD user describes behavior in terms of high level ADTs. The FAD implementor determines the data structures and codes the algorithms that realize that behavior. This section includes examples of implementation decisions that would be significantly different on different types of machines. It is exactly this type of decision that is hidden from the FAD user by ADT encapsulation.

A fundamental decision is how sources, queues, and links would be represented in specific data structures. On a single processor machine one reasonable representation would be an adjacency matrix for sources and queues and a single dimensional array for all the jobs currently in the system. However, this data structure would be inappropriate for a computer architecture that emphasizes message passing between autonomous multiple processors. In that case, each processor could manage one or more sources and queues, each of which could have an entirely local data structure. Links are defined by path destinations stored locally by a queue.

Similar differences arise when considering the implementation of the simulation manager, and its view of simulation time. Time management in a serial environment is a well understood problem. Parallel architectures with queues distributed among processors will require dramatically different approaches, particularly when an implementation strives for exploiting the inherent parallelism in simulation applications (see Nicol (1988)). A processor participating in a parallel simulation must sometimes block its progress, even if it has pending events. The decisions governing blocking and unblocking rely on a synchronization protocol, which typically is complex. The model builder can be isolated from these synchronization decisions using ADTs. For example, the ADT routine LONG identifies the job currently enqueued with the largest service requirements. In a serial environment LONG can be executed immediately. In a parallel environment the processor may not yet have all the information needed to correctly identify that job---its arrival may not yet have been reported to the processor by another. Consequently, the parallel implementation may have to block and eventually unblock the processor. The ADT implementor is responsible for the correct implementation of the LONG function, the ADT user is unaware of the added complications.

Serial and parallel implementations may vary in other ways. In Nicol (1988) we point out the advantages of pre-sampling job service times and branching destinations before the jobs actually arrive. In a serial implementation, the ADT routine GETARR would create the requested service time; a parallel implementation would have already created that time---GETARR simply reports it, and then creates other job characteristics for later arrivals. Again, the ADT implementor applys the techniques needed for parallelization, the ADT user need not be aware of these details.

Serial and parallel implementations may also vary in when job arrival events are posted. A serial implementation will not create an arrival event for a job at one queue before that job completes service at the routing queue. Yet, as shown in Nicol(1988), a parallel simulation can benefit from *pre-scheduling* job arrivals. A parallel implementation may consequently create a job arrival event within the ENDS ADT routine, before the job actually leaves service. A serial implementation of ENDS would not create the job arrival event for the destination queue.

Using the FAD system does not reduce the difficulty of implementing ADT routines, but it does isolate the implementation effort from the modeling decisions of the FAD user. This gives the FAD implementor the freedom of experimenting with different data structures and algorithms without requiring new coding effort from the FAD users. Even when the FAD user and the FAD implementor are the same programmer, the FAD preprocessor enforces this separation of concerns automatically, providing a measure of control and discipline to the implementation process. This discipline should result in making much of the implementation code reusable in different projects and by different researchers.

The FAD implementor can take advantage of the many subroutine packages already available for simulation. When a routine needed for the FAD user can be implemented using an existing routine from another library, the library can be renamed and placed in the FAD library or a new FAD routine can call the library routine. Thus, FAD can provide a well defined, limited interface to existing FORTRAN code without sacrificing the data type encapsulation benefits alluded to previously.

4. CONCLUSIONS

Simulations are in themselves complex entities, and specifying a simulation in any language requires deep understanding of the behavior being modeled as well as detailed knowledge of the language used for specification. When the description of a simulation in a programming language must include the intimate details of a parallel processing computer, the task is daunting. The resulting descriptions (be they in English, a design language, or a programming language) are difficult to write, difficult to read, and difficult to maintain.

There are substantial advantages in using FORTRAN to simulate queueing networks on parallel architectures; most machines have a FORTRAN compiler, FORTRAN subroutine libraries for simulation can reduce an implementation effort considerably, and many simulation researchers know FORTRAN. The FAD preprocessor allows FORTRAN programmers to exploit these advantages but also to use data abstraction techniques in the design and implementation of simulations. The separation of concerns embodied in FAD can be useful both to simulation experts and experts in parallel architecture: the information-hiding wall enforced by encapsulated ADTs protects the FAD user from machine-specific details and protects the FAD implementor from details specific to any particular simulation.

Our limited experience with using ADT descriptions and FAD implementations for queueing network simulations has been encouraging. Future research will include expanding the operations available in the FAD database and experimenting with FAD on new architectures.

APPENDIX A: QUEUEING ROUTINES

SUBROUTINE SETARR(J)
  Set the arrival time of the job J into its current queue.
REAL FUNCTION GETARR(J)
  Returns the arrival time of the job J into its current queue.
SUBROUTINE SETSRV(J,Q)
  Set the service time of the job J using the service distribution of the queue Q.
REAL FUNCTION GETSRV(J)
  Returns the service time of a job J.
REAL FUNCTION GSTIME(Q)
  Return the next random service time from a queue Q.
REAL FUNCTION GDEST(Q)
  Returns the next random destination for a job leaving a queue Q.
INTEGER FUNCTION JOBCNT(Q)
  Returns the number of jobs currently in a queue, Q.
SUBROUTINE ENQUE(J, Q)
  Place the job J into the queue Q.
SUBROUTINE DEQUE(J, Q)
  Remove the job J from the queue Q.
SUBROUTINE FIRST(J, Q)
  Identifies the job that has been in the queue Q the longest, and copies it into the ADT job variable J (first come, first served).
SUBROUTINE LAST(J, Q)
  Identifies the job that has been in Q the shortest, and copies it into J.
SUBROUTINE OLDEST(J, Q)
  Identifies the job in Q that has the earliest creation date, and copies it into J.
SUBROUTINE LONG(J, Q)
  Identifies the job in Q that has the longest service time, and copies it into J.
SUBROUTINE SHORT(J, Q)
  Identifies the job in Q that has the shortest service time, and copies it into J.
SUBROUTINE STARTS(J, Q, T)
  Requests the simulation to start servicing a job J in queue Q at time T.
SUBROUTINE ENDS(J, Q, T)
  Requests the simulation to stop servicing a job J in queue Q at time T.
REAL FUNCTION NOW()
  Returns the simulated time when the function is executed.
REAL FUNCTION TLIMIT()
  Returns the simulation time when the simulation will end. The FAD user sets this time when s/he calls GOSIM.
LOGICAL FUNCTION ISJOB(J)
  Returns true if J is a valid job; returns false otherwise.
LOGICAL FUNCTION ISQUE(Q)
  Returns true if Q is a valid job; returns false otherwise.

APPENDIX B: DEFAULT SIMULATION CODE

```
      SUBROUTINE JOBARR(J, Q)
C     Simulation action on the arrival of J at Q.
      JOB    J
      QUEUE       Q
      IF (.NOT.ISJOB(J)) GOTO 9000
      IF (.NOT.ISQUE(Q)) GOTO 9001
      CALL SETARR(J, NOW())
      CALL SETSRV(J, GSTIME(Q))
      CALL SETDST(J, GDEST (Q))
      IF (JOBCNT(Q) .EQ. 0) GOTO 900
      CALL ENQ(J, Q)
      GOTO 1000
 900  STARTS(J, Q, NOW())
1000  RETURN
9000  CALL BADJOB(J)
      RETURN
9001  CALL BADQUE(Q)
      RETURN
      END

      SUBROUTINE ENTSRV(J, Q)
C     Simulation action when J goes into service at Q.
      JOB    J
      QUEUE       Q
      IF (.NOT.ISJOB(J)) GOTO 9000
      IF (.NOT.ISQUE(Q)) GOTO 9001
      CALL ENDS(J, Q, NOW() + GETSRV(Q))
      RETURN
9000  CALL BADJOB(J)
      RETURN
9001  CALL BADQUE(Q)
      RETURN
      END

      SUBROUTINE JOBXIT(J, Q)
C     Simulation action when J leaves Q.
      JOB    J
      QUEUE       Q
      IF (.NOT.ISJOB(J)) GOTO 9000
      IF (.NOT.ISQUE(Q)) GOTO 9001
      CALL SEND(J)
      CALL DEQUE(J, Q)
      IF (JOBCNT(Q) .EQ. 0) GOTO 1000
      CALL GETNXT(J, Q)
      CALL STARTS(J, Q, NOW())
9000  CALL BADJOB(J)
      RETURN
9001  CALL BADQUE(Q)
      RETURN
1000  RETURN
      END

      SUBROUTINE GETNXT(J, Q)
C     Determining the next job to service in Q.
      JOB    J
      QUEUE       Q
      REAL  SOFAR
      IF (.NOT.ISQUE(Q)) GOTO 9001
      IF (JOBCNT(Q) .EQ. 0) GOTO 9002
      CALL FIRST(J, Q)
      RETURN
9001  CALL BADQUE(Q)
      RETURN
9002  CALL GETMT(Q)
      RETURN
      END
```

APPENDIX C: ERROR HANDLERS

SUBROUTINE QOVER(Q, J)
Called when queue Q is sent job J, but Q is already at its capacity.
SUBROUTINE NETOVR(N, S, J)
Called when source S produces job J in net N, but N is already at its capacity.
SUBROUTINE BADQ(Q)
Called when a FAD user refers to an uninitialized queue, Q.
SUBROUTINE BADJOB(J)
Called when a FAD user refers to a non-existent job, J.
SUBROUTINE BADSRC(S)
Called when a FAD user refers to a non-existent source, S.
SUBROUTINE SYSOUT(N)
Called when a network initialization for N exceeds system memory limits.
SUBROUTINE GETMT(Q)
Called when GETNXT is called with an empty queue, Q.

REFERENCES

Brately, P., Fox, B.L., and Schrage, L.E. (1983). *A Guide to Simulation.* Springer-Verlag, New York.

Ghezzi, C. and Jazayeri, M. (1987). *Programming Language Concepts,* Second Edition. John Wiley and Sons, New York.

Guttag, J. and Horning, J. (1978) The algebraic specification of abstract data types. *Acta Informatica 10,* 27-52.

Kreutzer, W. (1986). *System Simulation - Programming Styles and Languages.* Addison Wesley.

L'Ecuyer, P. and Giroux, N. (1987). A process-oriented simulation package based on Modula-2. In: *Proceedings of the 1987 Winter Simulation Conference* (L.Thesen, H.Grant, and W.David Kelton, eds.) 165-173.

Miller, K.W., Morell, L.J., and Stevens, F. (1989). Enhancing the reuse of FORTRAN software by enforcing data abstraction. *I.E.E.E. Software,* to appear in January.

Nicol, D.M. (1988). High performance parallelized discrete-event simulation of stochastic queueing networks. In: *Proceedings of the 1988 Winter Simulation Conference,* San Diego, CA, December 1988.

Parnas, D.L. (1977). The use of precise specifications in the development of software. In: *Proceeding of the IFIP Congress 1977.* North Holland Publishing Company, 861-867.

Parnas, D.L. (1979). Designing software for ease of extension and contraction. *I.E.E.E. Transactions on Software Engineering SE-5,* 128-138.

AUTHORS' BIOGRAPHIES

KEITH W. MILLER is an assistant professor in the computer science department of the College of William and Mary. He received his Ph.D. from the University of Iowa in 1983, his M.S. in mathematics from the College of William and Mary in 1976, and his B.S. in education from Concordia Teachers College in 1973. His research interests include abstract data types, formal specifications, and computer vision. He is a member of ACM and IEEE.

Keith Miller
The Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185
(804) 253-4748

DAVID M. NICOL received the B.A. in mathematics from Carleton College, Northfield, MN. in 1979, and the M.S. and Ph.D. degrees in computer science from the University of Virginia in 1984 and 1985. He was a programmer/analyst with Control Data Corp. from 1979 to 1982, and a staff scientist at the Institute for Computer Applications in Science and Engineering from 1985 to 1987. He is presently an assistant professor in the department of computer science at the College of William and Mary. He has published numerous articles in the area of parallel processing, and is a member of ACM, IEEE Computer Society, and ORSA/TIMS.

David M. Nicol
The Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185
(804) 253-4748