*Proceedings of the 1986 Winter Simulation Conference*
J. Wilson, J. Henriksen, S. Roberts (eds.)

# DISCRETE EVENT SIMULATION IN PASCAL WITH SIMTOOLS

Andrew F. Seila
Department of Management Sciences
and Information Technology
College of Business Administration
University of Georgia
Athens, GA 30602 U.S.A.

## ABSTRACT

SIMTOOLS is an integrated collection of procedures and functions which allow one to write discrete event simulation programs in Pascal using the event view. Facilities are provided for generating random variates, creating and deleting entities, managing resources, list processing, scheduling, cancelling, sequencing and executing events, tracing events and collecting data. This paper describes the structure and capabilities of SIMTOOLS, and presents an example simulation using SIMTOOLS.

## 1. INTRODUCTION

SIMTOOLS is an integrated collection of procedures and functions which allow a Pascal programmer to write discrete event simulation programs using the event view, or event scheduling approach. This paper presents a brief review of the facilities in SIMTOOLS. For a more detailed description, refer to the SIMTOOLS User's Manual (Seila, 1986).

Facilities are provided in SIMTOOLS for:

- Generating random variates.
- Declaring entities and attributes.
- Creating and deleting entities.
- Inserting entities into and removing them from lists (queues).
- Declaring and managing resources.
- Declaring event notices.
- Scheduling and cancelling events.
- Sequencing and executing events.
- Tracing events.
- Collecting data.

Version 1.0 was developed with the following objectives:

1. Data structures are simple.
2. Procedures and functions are simple and descriptive, and have few parameters.
3. The simulation program is self-documenting as far as possible.
4. Standard Pascal is used.
5. The internal mechanics of list-processing, data collection, tracing and other operations are transparent to the user.

## 2. PASCAL KNOWLEDGE REQUIRED

The SIMTOOLS user, as any Pascal programmer, must be familiar with the simple types real, integer, char, boolean, scalar and subrange. The structured types ARRAY and RECORD, including record variants, are also used extensively in declaring entities and event notices. Knowledge and use of pointers and dynamic records is also required. The user does not need extensive experience with FILE types, other than standard text files, since they are not used by SIMTOOLS.

A simulation program using SIMTOOLS, just like any other complex program should be developed and written using topdown design. This means that the program should be extensively structured using procedures and functions. The user must be very familiar with procedure and function declaration and execution, and parameter passing conventions, including passing of procedures. The effective use of SIMTOOLS will be greatly enhanced by the user's ability to structure the simulation program using procedures and functions.

The SIMTOOLS user should, in short, be an experienced Pascal programmer. A good text is Programming in Pascal, Second Edition, by Peter Grogono (Addison-Wesley, Reading, Mass., 1983). Chapters 1 through 6 and 8 through 10, omitting sections 10.5 and 10.6, are recommended along with the experience of writing several nontrivial Pascal programs. A good one-semester course in introductory programming with Pascal is sufficient background.

## 3. RANDOM VARIATE GENERATION

SIMTOOLS contains routines for generating random variates from a number of different distributions. These random variate generators support multiple streams of random number seeds and can be logically divided into three groups:

1. Routines for managing the streams of seeds;
2. The random number generator;
3. Routines for generating random variates from various distributions.

A procedure, getseeds, will read in a specified number of random number seeds and

initialize the seed streams with them. Function seed will return the current seed in a specified stream.

The random number generator, which is in function rand01, is a prime modulus 32-bit multiplicative congruential generator using modulus $2^{32} - 1$. This generator has been tested exhaustively by Fishman and Moore (1984), using both geometrical and statistical tests, and has been shown to have quite good properties if the appropriate multiplier is used. Fishman and Moore list ten of the best multipliers. Of these ten, the multiplier 742938285, was implemented because it performed as well as or better than the other multipliers.

Currently, SIMTOOLS has functions for generating random variates from the following distributions:

| Continuous: | Discrete: |
|---|---|
| Uniform | Empirical |
| Exponential | Bernoulli |
| Erlang | Binomial |
| Normal | Poisson |
| Lognormal | |
| Triangular | |

In addition, a function is available which returns the boolean values true and false with a specified probability, p, so that one of two options can be selected with probability p.

## 4. ENTITIES AND LIST PROCESSING

The RECORD type is used to implement entities. In an entity record, each field represents an attribute of the entity. Any system will generally have several types of entities and multiple instances of entities of each type. For example, in a simple queueing simulation, there are two types of entities, customers and servers, and at any point in time, there may be multiple customers and multiple servers operating.

Each entity has two groups of attributes: system-defined and user-defined. System-defined attributes are required by the SIMTOOLS package to identify entities, allow them to belong to a list, and to permit other operations performed by the package. The user is never required to (and indeed should not) reference these attributes directly. User-defined attributes may be declared for each type of entity by inserting additional fields in the variant part of the entity RECORD declaration. The variant label serves to specify which type of entity the attributes apply to, and the field identifiers name these attributes.

Entities are normally classified into two categories: permanent and temporary. This distinction is implemented in SIMTOOLS in the way the user declares an entity or group of entities. Since permanent entities exist for the duration of the simulation, they are declared as an ARRAY (or other appropriate static data structure) of entities. Temporary entities, on the other hand, are declared as dynamic records that can be created and deleted as needed. One of the strengths of using Pascal is that the richness of data structures allows the programmer to organize collections of entities in any way desired.

### 4.1 Creating and Deleting Entities

Procedure create creates a new instance of a specified type of entity, and procedure delete deletes an existing temporary entity. SIMTOOLS automatically counts the number of entities of each type that have been created and deleted. Functions entycreated, entydeleted and entycount return the cumulative number of entities of a given type that have been created, deleted, or are currently in the system, respectively. Procedure nameentity assigns a 10-character name to a specified type of entity. This name, along with the sequence number assigned to each entity created, is used to identify the entity uniquely in any messages that are printed.

### 4.2 List Processing

List processing involves creating lists, placing entities into lists and removing entities from them. Other simulation packages and languages use the terms "files," "queues," "sets," and "waiting lines" to refer to lists. All lists are implemented by SIMTOOLS as two-way linked lists, and all list processing operations are performed through procedure and function calls.

Before entities can be placed into a list, it must be declared and created. A procedure, setuplist, creates and initializes the list. After this call, entities may be placed into the list. SIMTOOLS has five procedures that may be used to place an entity in a list. Two of these, filefirst and filelast, are used to place entities first or last in the list. Two more, filebefore and fileafter, place an entity either before or after another entity in the list. The last procedure, filepriority, is used to place entities in a list ordered by increasing value of an attribute. This procedure can be used to maintain lists in priority order, and by defining the priority attribute carefully, it can be used to order lists by priority within multiple categories.

Functions tell the status of a list and give access to the entities in it. Functions full and empty return boolean values specifying whether the list is full or empty. Function listcount returns the number of entities currently in the list, and functions first and last return pointers to the first and last entities in the list.

Procedure remove removes an entity from the list to which it belongs. The procedures which place an entity into a list first check to see if it already belongs to a list. If so, they remove the entity from that list before placing it in the new list. Thus,

there is no need to first remove an entity from a list before placing it in another list. Note that SIMTOOLS does not allow entities to belong to more than one list at a time.

Another procedure, take, removes an entity from a list and assigns it to another entity pointer. Calling this procedure is equivalent to first assigning the entity pointer to the new variable (the pointer may be the value returned by functions first or last), then calling procedure remove.

Finally, SIMTOOLS also has a procedure, search, for searching a list to find an entity that satisfies a user-defined criterion. The user-defined criterion is specified by passing to this procedure a boolean function which evaluates whether an entity satisfies the criterion and returns true if it does, false otherwise.

## 5. RESOURCES

A resource is a particular type of entity that other entities "use" during the course of a simulation. Since resources interact with other entities in a very specific way, they were given a different treatment. In general, a simulation may have several distinct types of resources. For example, a manufacturing plant may have presses, cranes, rivets and oil. Each of these is a particular type of resource. There may then be multiple instances of each type of resource. The manufacturing plant may have 3 presses, 2 cranes, 10 bins of rivets and 1 tank of oil. Then, each instance of the resource will have a quantity of the resource that is available to be used. The presses and cranes cannot be divided, so the quantity of these resources at each instance is 1. The quantity of rivets may be several thousand at each location, and the quantity of oil may be several thousand gallons.

Resource records have attributes (fields) to store the resource's type, name and number values. Each resource can be uniquely identified by these three values. In addition, the name, which is a 10-character string, is used in trace output to identify actions which increase and decrease the level of the resource, or check the resource's level. Resources are generally implemented as permanent entities.

Three procedures are provided in SIMTOOLS to manage resources. Two of them, increase and decrease, increase and decrease the level of the resource. The third, checkresource, checks the level to see if a requested amount is available and assigns the value true to a boolean parameter if it is.

## 6. EVENTS AND THE EVENT LIST

The event view, or event scheduling approach, is implemented in SIMTOOLS in the usual way with an event list containing event notices arranged in the order of the time of occurrence. SIMTOOLS routines are available to schedule and cancel event occurrences, and perform other utility functions associated with managing the event list. A timing routine is provided to control the execution of events.

### 6.1 Event Notices

Event notices are declared as records. These records have fields in the fixed part that specify the event type, time of occurrence, and other items of information that are used in scheduling events. These attributes are always assigned and used by the routines that schedule and cancel events and perform other actions involving event notices. A variant part of the event notice record stores the values of event parameters to be passed to event routines. Some types of events may not have parameters, and in this case, the variant part of the event notice will remain a null variant.

### 6.2 The Event List and Event Scheduling

The event list in SIMTOOLS is a two-way linked list containing event notices ordered by increasing values of the event time. When the simulation program begins execution, the event list, just like other lists, does not exist. Procedure setupsimulation creates the event list, initializes it to be empty and in addition performs other chores such as setting the starting time to 0.0, initializing the number of entities created, deleted and in the system to 0, and assigning blank names to all entity and event types. Thus, procedure setupsimulation should be called first in the initialization part of the simulation program, before any names are assigned, or events are scheduled. After setupsimulation is executed, function eventlist will return a pointer to the event list and function now will return the current system time. Another function, delay, returns a value which is a specified length of time after the current system time and can be used to schedule events after a specific delay.

Once the event list is established, events can be scheduled by inserting their event notices in the event list. If the event notice does not exist, procedure generate will create the event notice and schedule the event. The event will be placed in the event list according to the time of the event, and the user can control whether the event is scheduled before or after other events with the same event time. If the event notice exists, procedure schedule can be used to schedule the event. Thus, event notices can be re-used. If an event is scheduled, procedure reschedule can be used to remove its event notice from the event list, give it a new time, and schedule the event at the new time. Procedure cancel will remove an event notice from the event list, thus cancelling the event occurrence.

A procedure, findevent, is provided to search the event list for the first event notice for a particular type of event. The

procedure can start searching the event list at the beginning or at any specified event notice, and thus can be used to locate any or all scheduled events of a particular type.

### 6.3 The Timing Routine

The timing routine, which is procedure runsimulation in SIMTOOLS, is the "heart" of a discrete event simulation. It is this routine, along with the event routines that move the simulation through time and causes the correct sequence of changes in the system. Procedure runsimulation will let the simulation run until either a specified length of time has elapsed, or until there are no more event notices in the event list. The simulation may also be stopped by calling procedure stopsimulation. After stopsimulation is called, any attempt to schedule events by calling schedule, generate or reschedule will have no effect. The next statement executed after the simulation stops is the one following the runsimulation statement.

The system must be initialized to its starting state before procedure runsimulation is called. Initialization involves calling setupsimulation to create the event list, creating all lists and other dynamic data structures in the system, initializing system attributes, and scheduling at least one event. Once the system is initialized, runsimulation performs the following actions:

1. If the duration of the segment is positive, schedule an event to end the segment.

Repeat the following:

2. Remove the first event notice on the event list. This becomes the event notice of the current event.
3. Update the system clock to the time on this event notice. Procedure now will return this time.
4. Execute the event routine corresponding to the event notice.
5. If the event notice was not rescheduled, then dispose of it.

Until the end of segment event notice is removed or the event list is empty.

When the timing routine finishes execution, the system state (all entities, attributes, list memberships, lists, resources, etc.) is not destroyed or altered. This allows the simulation to be run in segments.

Function currentevent returns a pointer to the event notice of the current event. This is useful to access the event parameters on the current event notice, or to re-use the current event notice to schedule another event occurrence. Function eventlist returns a pointer to the event list. Although this is normally not used, it is useful when calling procedure findevent to start searching at the start of the event list, and possibly in other cases when the event notices in the event list are to be accessed. Function nextevent returns a pointer to the first event notice on the event list, and function eventsremaining is a boolean function which tells whether the event list is empty or not.

Sometimes, it is useful to "disable" events, so that they will not occur, even though they are scheduled. For example, if an automatic teller machine is being modelled and there is a period of time when the machine is inoperative, we would want transactions during this time period not to be processed. This could easily be done by "disabling" the event that starts transaction processing at the start of the inoperative period, and "enabling" it at the end of this period. Two procedures are provided to allow events of a specific type to be "disabled" and "enabled." Procedure enable "enables" events and procedure disable "disables" them. At the start of the simulation, all events are "enabled". Function occurflag can be used to determine if a particular type of event is "enabled" or "disabled."

### 6.4 Event Routines and Event Parameters

Associated with each type of event is an event routine, which is a set of actions that take place when that particular type of event occurs. In order to implement the event view, two things must be provided: a mechanism for carrying out these actions, and a mechanism for executing the correct event routine when an event notice is removed from the event list. The first of these is accomplished by writing a procedure for each event routine. This is called an "Event procedure." The second is accomplished by providing a procedure, doevent, which selects and executes the appropriate event routine and passes any required parameters from the event notice to the event procedure. There are many ways to pass this information. SIMTOOLS stores the parameters on the event notice, then when the event routine is executed, passes the values of the parameters directly to the event procedure.

## 7. EVENT TRACING

Facilities are built into SIMTOOLS for producing a readable trace describing system execution. This is an important part of any simulation language because it allows the analyst to follow system operation and verify its correctness as it moves through time. This section discusses facilities to turn this trace on and off, what output is produced, and procedures for generating additional trace output to provide more information.

### 7.1 Routines for Controlling the Event Trace

SIMTOOLS includes three routines for managing the event trace. Procedure runtrace turns on the automatic trace for a specified number of events, or indefinitely. A call to procedure stoptrace turns the trace off and ends trace output. Function trace returns a boolean value telling whether the trace is currently on or not. When the event trace is activated, one line is produced at the start of each event that gives the current system time, the name of the event, if a name has been assigned, and a message indicating that the event is starting. Within each event, a message (usually one line) is produced when any of the following occurs:

- An entity is created or deleted.
- A list is created and initialized.
- An entity is placed in or removed from a list.
- A list is searched.
- A resource is initialized.
- A resource is increased, decreased or checked.
- An event is generated, scheduled, rescheduled or cancelled.
- The event list is searched to find an event notice.
- The timing routine is executed.
- The event trace is turned on or off.

Therefore, the standard trace output will report most actions that relate to entities, resources or events. An example of a standard trace output, taken from a fuel depot model in (Seila, 1986), is given in Appendix A.

### 7.2 Additional Trace Output

The standard trace output produced by SIMTOOLS is frequently sufficient to assure that the model is implemented correctly. However, additional information is sometimes needed about user-defined attributes, list memberships, conditional actions and other items of information that SIMTOOLS does not provide directly. When this is the case, the user must provide additional output statements. Function trace provides a way to produce trace output only when the trace is turned on. If this is done, one does not need to remove the additional trace output statements from the program in order to turn all trace output off and proceed to make production runs of the simulation.

Sometimes it is useful to print the contents of a list in order to see which entities are in the list. SIMTOOLS has a procedure, showlist, which does this and also allows the user-defined attributes of any entity in the list to be printed when the entities are listed. Another procedure, showeventlist, prints information about all event notices in the event list and can include the values of the event parameters.

## 8. DATA COLLECTION

SIMTOOLS provides basic facilities for collecting data and computing some sample statistics. Since there are a large number of ways to analyze simulation data, depending upon the objectives of the analysis and the nature of the data generated, a comprehensive set of data analysis procedures would be difficult to provide in a reasonable size package.

SIMTOOLS does not have facilities for automatic report generation. This omission reflects the opinion that reports produced without consideration of the nature of the model and the simulation environment can be misleading and indeed lead to erroneous conclusions concerning the system. For example, if the simulation run takes place during a transient period, such as when the amount of work in the system is steadily building up, such figures as average number of entities in a list are meaningless. If the system is operating in a stable condition, averages computed in the usual way can be appropriate, but measures of variation, such as the ordinary sample variance, are almost always highly biased, due to the correlation among observations.

Data collection facilities in SIMTOOLS consist of a general data structure to store accumulated observations for both discrete and continuous data, and procedures to initialize the accumulators and accumulate sample statistics.

### 8.1 User-controlled Data Collection

Procedures resetsums and tally initialize the data accumulators and accumulate sums and sums-of-squares for discrete statistics, respectively. Once the sums and sums-of-squares have been accumulated, functions average and variance can be referenced to compute the sample mean and sample variance of the data.

Certain variables such as the the level of a resource or the number of entities in a list must be accumulated continuously over time in order to compute time averages. In a discrete event simulation, continuous observations form a step function, since the system changes only at discrete points in time, and remains constant between these points. In this case, accumulating the area under the observation function becomes a matter of adding up the areas of rectangles. Procedure resetarea initializes the area accumulators. Once resetarea has been called, procedure accumarea may be called just before each change to the variable to accumulate the area. After the area has been accumulated, function timeavg will return the time average of the observed variable.

### 8.2 Automatic Data Collection

SIMTOOLS automatically collects summary statistics on the following measurements:

- the number of entities in each list;
- the number of entities of each type in the system;

- the level of each resource.

Thus, computing the time averages of these quantities simply involves a call to function timeaverage at the appropriate point in the program. If data collection for these quantities is to begin at some point other than the start of the simulation (which is usually the case), procedure resetarea can be called to reinitialize the data accumulators.

## 9. SIMTOOLS ADVANTAGES

Appendix B contains a simulation of a multiserver queueing system, written in Pascal using SIMTOOLS. This is a simple model which has been used in many textbooks. The version which is in the Appendix was written to be directly comparable to the example that was presented in the pamphlet "A Quick Look at SIMSCRIPT II.5," which was distributed by CACI.

A careful study of the example in the Appendix B shows that the simulation program written in Pascal using SIMTOOLS is very readable and self-documenting. A programmer can write programs using SIMTOOLS that need very little documentation within the program code (in comments). This can also be said of languages such as SIMSCRIPT II.5, but can certainly not be said of FORTRAN.

Most simulation programs are quite complex. The simple example in the appendix would most likely never be used in pratice. Realistic simulations involve tens-of-thousands of lines of code. Thus, any approach to developing the simulation program should allow the programming team to organize the task of designing the program. Although a top-down approach can and should be employed with any language, some facilitate using this approach more than others. Pascal, in particular, greatly facilitates the use of top-down design, and helps organize the process of program development. This is another advantage in using SIMTOOLS.

One philosophy that has been employed in software development is that one should start with a basic library of routines to do basic activities in the program. Then, these can be used to develop a library routines to do more complex tasks. This library can then be used to develop a library of routines to do even more complex tasks, and so on, until the tasks one wishes to perform can be done. Kernighan and Plauger (1981) do an excellent job of presenting this approach to software development. This is the approach embodied in simulation software development using SIMTOOLS. The current package is a first-level library. This can then be used to develop a second-level library of routines that are oriented toward a particular class of systems. For example, one can use SIMTOOLS to develop a library of routines for manufacturing simulation. This library would have facilities for operating conveyors, AGV's, and other equipment to move parts and finished goods, as well as more specific facilities for inventory management, parts routing, assembly operations and other processes associated with manufacturing. Once this library is in place, it can be used to develop a package of routines for manufacturing in a particular industry, such as automobiles, or textiles.

One application of SIMTOOLS is SIMTOOLS/P (Seila, 1985). This is a package of routines which uses SIMTOOLS to implement the process view of simulation. In this implementation, processes can be defined, initiated, activated and terminated, and they can use resources and facilities for periods of time, or can wait indefinitely until a particular resource or facility is available. The package allows events to continue to be scheduled as well as processes to operate. Systems modeled using the process view can be represented more naturally and succinctly than using the event view.

## REFERENCES

Fishman, G. S. and Moore, L. W. (1980) "An Exhaustive Analysis of Multiplicative Congruential Generators with Modulus $2^{31} - 1$," Technical Report, Curriculum in Operations Research and Systems Analysis, University of North Carolina, Chapel Hill, NC.

Kernighan, B. and Plauger, K. (1981), Software Tools in Pascal, Addison-Wesley, Reading, Massachusetts.

Seila, A. F. (1985), "Implementing the Process View in Pascal," Technical Report, College of Business Administration, University of Georgia, Athens, Georgia.

Seila, A. F. (1986), SIMTOOLS User's Manual, Technical Report, College of Business Administration, University of Georgia, Athens, Georgia.

## AUTHOR'S BIOGRAPHY

Andrew F. Seila is an Associate Professor of Management Sciences in the College of Business Administration at the University of Georgia. He received the B.S. degree in physics in 1970 and the Ph.D. degree in operations research in 1976, both from The University of North Carolina at Chapel Hill. Before joining the faculty at the University of Georgia, he was at Bell Laboratories in Holmdel, New Jersey. His current research interests include all aspects of discrete event simulation, including model development and validation, and simulation output analysis, especially analysis of multivariate output data. He is a member of ORSA, TIMS and ASA.

Andrew F. Seila
Department of Management Sciences
and Information Technology
College of Business Administration
University of Georgia
Athens, GA 30602
(404) 542-8067

**APPENDIX A: SAMPLE TRACE OUTPUT FROM SIMTOOLS**

```
SIMTOOLS  VERSION 1.0                                              02-23-86
(C) COPYRIGHT 1985   Andrew F. Seila                               22:09:52

  Time      Event                Action(s)
================================================================================
                      Run trace for   5 events.
                      Set up list Truck Que     1
                      Setup resource
                         Fuel        1 with level   10.000
                      Setup resource
                         Fuel        2 with level    5.000
                      Generate new Arrival    event at time      0.737
                      reset area accumulators.
                      reset area accumulators.
                      Reset sum accumulators.
                      Run simulation indefinitely.
    0.737 Arrival     -------------- Start event ------------
                      Schedule Arrival    at time       0.792
                      Create Truck         1
                      Check Fuel        1
                           12.05 requested;
                           10.00 available.
                      Insert Truck         1 last in Truck Que     1
                      Generate new Refilling  event at time       1.176
    0.792 Arrival     -------------- Start event ------------
                      Schedule Arrival    at time       0.846
                      Create Truck         2
                      Check Fuel        1
                            5.99  requested;     10.00 available.
                      Decrease Fuel          1 by    5.994
                         New level is    4.006
                      Generate new Pump endng event at time       1.091
    0.846 Arrival     -------------- Start event ------------
                      Schedule Arrival    at time       1.801
                      Create Truck         3
                      Check Fuel        1
                            7.20 requested;      4.01 available.
                      Insert Truck         3 last in Truck Que     1
                      Find next Refilling  event.
                         Found one at time       1.176
    1.091 Pump endng  -------------- Start event ------------
                      Delete Truck         2
                      Search in Truck Que    1
                       none found.
    1.176 Refilling   -------------- Start event ------------
                      Increase Fuel          1 by   97.195
                         New level is  101.201
                      Search in Truck Que    1
                        Found Truck          1
                      Remove Truck         1 from Truck Que    1
                      Decrease Fuel          1 by   12.052
                         New level is   89.149
                      Generate new Pump endng event at time       1.779
                      Search in Truck Que    1
                        Found Truck          3
                      Remove Truck         3 from Truck Que    1
                      Decrease Fuel          1 by    7.195
                         New level is   81.954
                      Generate new Pump endng event at time       2.076
================================================================================

                      END OF SIMULATION SEGMENT
                         Event Counts:
                       ID      Name      Count
                       --   ----------   -----
                        0                    0
                        1    Arrival        40
                        2    Pump endng     40
                        3    Refilling       5
                        4                    0
                        5                    0
```

**APPENDIX B: AN EXAMPLE SIMULATION PROGRAM IN PASCAL USING SIMTOOLS**

```
{$title:'Bank simulation from A Quick Look at SIMSCRIPT II.5'}

PROGRAM banksim( input, output, simfile);

{   The following are compiler directives to include the type
    declarations file and two files containing the declarations
    for the SIMTOOLS procedures and functions:   }
{   In the declarations file, the following declarations are made:
      A single type of entity, "bankcustomer," is declared to
      represent a bank customer.  This entity has a single attribute,
      "atime", which contains the customer's arrival time.
      A single type of resource is declared to represent the tellers
      in the bank. This resource has no user-defined attributes.
      Three types of events are declared: "newcustomer," to denote the
      arrival of a new customer; "finishbanking," to denote the end
      of service at a teller; and "closebank," to denote the close of
      the day and end of the customer arrivals.  "Newcustomer" and
      "closebank" have no parameters.  A single event parameter,
      "customer," which is a pointer to an entity, is declared for
      the "finishbanking" event.
{$subtitle:'Simulation declarations.',$include:'Bankdecl.dcl'}
{$list-  Turn off the listing. }
{$subtitle:'Event View declarations',$include:'evview.dcl'}
{$subtitle:'RV Generation declarations',$include:'rvgen.dcl'}

{$subtitle:'Simulation program.' }

{  Variable declarations for this simulation }

  VAR  queue              : listhead;  { The queue for customers }

       server             : resource;  { The tellers }

       lambda,                         { Arrival rate }
       mu,                             { Service rate }
       daylength          : real;      { Length of day }

       lq,                             { Average queue length }
       l,                              { Average no. in system }
       wq,                             { Mean wait in queue }
       w      : real;                  { Mean wait in system }

       quewait,                        { Data record for wait in queue }
       syswait          : statistics; { Data record for wait in system }

       syscustnum    : entitypointer; { Pointer for number in system statistics

       firstarrival,                   { Event notice for first arrival event }
       bankclosing   : eventpointer;  { Event notice for bank closing event }


PROCEDURE initializesystem;
{ Purpose: To initialize the system prior to starting the simulation. }
BEGIN
  { Assign names to each type of entity and event: }
       nameentity( bankcustomer, 'Customer  ' );
       nameevent( newcustomer, 'Arrival   ');
       nameevent( finishbanking, 'Endservice');
       nameevent( closebank, 'Close Bank');
  { Create and initialize the queue: }
       setuplist( queue, 'Queue     ', 1 );
  { Initialize the server: }
       setupresource( server,'Server    ',1,0.0 );
  { Initialize the user-controlled data collectors: }
       resetsums( quewait );  resetsums( syswait );
  { Input two random number seeds: }
       getseeds( input, 2 )
END;  {  initializesystem  }
```

```
{  Forward Declarations for Event Procedures }

PROCEDURE arrival;  FORWARD;

PROCEDURE endofservice( customer : entitypointer ); FORWARD;

PROCEDURE closing;   FORWARD;

PROCEDURE doevent( event : eventpointer ) [PUBLIC];
{ Purpose: To select the appropriate event routine and
           execute it, passing the event parameters. }
BEGIN
  WITH event^ DO CASE id OF
       endsegmt : ;  { Required by Event View Package. }
    { If event type is new arrival, execute procedure "arrive": }
       newcustomer : arrival;
    { If event type is end of banking, execute procedure "endofservice": }
       finishbanking : endofservice( customer );
    { If event type is bank closing, execute procedure "closing": }
       closebank : closing
    END   { CASE id OF ... }
END;   {  doevent  }

{ Event procedure for the new customer event: }
PROCEDURE arrival;
  VAR person      : entitypointer;
      leavebank   : eventpointer;
      qtime       : real;
BEGIN
  create( person, bankcustomer );     { Create a new customer }
  person^.atime := now;               { Record the arrival time: }
  IF server.unitsavail = 0            { Can service begin? }
    THEN filelast( person, queue )    { If not, put customer in queue }
    ELSE BEGIN                        { If yes, do the following: }
      qtime := 0.0;                   { Time in queue is 0. }
      tally( qtime, quewait );        { Record time in queue statistics }
      server.unitsavail := server.unitsavail - 1; { Take a server }
      generate( leavebank, finishbanking,  { Generate an end-of-service }
           delay( rvexpon( 1.0/mu, 1 ))); { event. }
      leavebank^.customer := person  { Put customer pointer on event notice }
      END;  { ELSE }
  { Reuse the event notice to schedule the next arrival after an
    exponentially distributed delay time. }
  reschedule( currentevent, delay( rvexpon(1.0/lambda, 2 )), after )
END;  {  arrival  }

{ Event procedure for the end of banking event: }
PROCEDURE endofservice(  customer: entitypointer  );
  VAR  nextcustomer : entitypointer;
       leavebank    : eventpointer;
       eltime,
       qtime        : real;
BEGIN
  eltime := now - customer^.atime;    { Compute time in system for customer }
  tally( eltime, syswait );           { Record time in system statistics }
  delete( customer );                 { Delete the customer }
  IF empty( queue )                   { Check for another customer in queue }
    THEN                              { Empty queue -- return the teller }
      server.unitsavail := server.unitsavail + 1
    ELSE BEGIN                        { Queue not empty -- start service }
      take( first( queue ), nextcustomer );  { Take the first customer }
      qtime := now - nextcustomer^.atime;    { Compute time in queue }
      tally( qtime, quewait );        { Record time in queue statistics }
      generate( leavebank, finishbanking,    { Generate end-of-service event }
                delay( rvexpon(1.0/mu, 1 )));
      leavebank^.customer := nextcustomer { Put customer pointer on event notic
      END  { ELSE }
END;  {  endofservice  }

{ Event procedure for bank closing event: }
PROCEDURE closing;
  VAR  nextarrival : eventpointer;
BEGIN
  { Locate the next arrival event in the event list }
```

```
      findevent( nextarrival, newcustomer, eventlist );
  { Cancel the event and dispose of the event notice }
  cancel( nextarrival );  dispose( nextarrival )
END;  {   closing   }


{ Main Program: }
BEGIN
  rewrite( simfile );  page( simfile );  { Use file "simfile" for output }
  runtrace( -100 );                      { Turn on the event trace indefinitely }
  initializesystem;                      { Initialize the system }
                                         { Write the report headings. }
  writeln( simfile,'Simulation of a single-queue multiple server system.');
  writeln( simfile );  writeln( simfile );  writeln( simfile );
  writeln( simfile,'  No of    Arrival   Service Hours  ',
     'Avg Queue  Avg No.   Avg Time   Avg Time');
  writeln( simfile,' Servers    rate      rate   /Day   ',
     '  Length  In System  In Queue   In system');
  read( server.unitsavail );   { Read the number of servers in the system. }
  REPEAT   { Enter a loop for each set of parameters to be evaluated. }
    { Read the parameters. }
        read( lambda, mu, daylength );
    { Schedule first arrival. }
        generate( firstarrival, newcustomer, now );
    {Schedule first closing. }
        generate( bankclosing, closebank, delay(daylength) );

    { Turn over control to the timing routine. }
        runsimulation( indefinitely );

    { At this point, the simulation has finished running. }
    { Compute and print the averages. }
        custcount := listcount( queue );
        accumarea( custcount, queue^.data );
        lq := timeavg( queue^.data );
        syscustnum := entystatistics(bankcustomer);
        l  := timeavg( syscustnum^.data );
        wq := average( quewait );
        w  := average( syswait );
        writeln( simfile, server.unitsavail:7,lambda:11:3,mu:10:3,
              daylength:6:1,lq:11:3,l:10:3,wq:11:3,w:10:3);
    { Reset the data accumulators for the next run. }
        resetsums( quewait );  resetsums( syswait );
        resetarea( syscustnum^.data );  resetarea( queue^.data );
    { Read the next number of servers. }
        read( server.unitsavail )
    { Continue the simulations until all data has been
      read on file "input". }
  UNTIL  eof( input );
END.  {  program  }
```