Table 4.4 Output Features

Feature	*Description*
Scenario manager	Create user-defined scenarios to simulate
Run manager	Make all runs (scenarios and replications) and save results for future analyses
Warmup capability	For steady-state analysis
Independent replications	Using a different set of random numbers
Optimization	Genetic algorithms, tabu search, etc.
Standardized reports	Summary reports including averages, counts, minimum and maximum, etc.
Customized reports	Tailored presentations for managers
Statistical analysis	Confidence intervals, designed experiments, etc.
Business graphics	Bar charts, pie charts, time lines, etc.
Costing module	Activity-based costing included
File export	Input to spreadsheet or database for custom processing and analysis
Database maintenance	Store output in an organized manner

Table 4.5 Vendor Support and Product Documentation

Feature	*Description*
Training	Regularly scheduled classes of high quality
Documentation	Quality, completeness, online
Help system	General or context-sensitive
Tutorials	For learning the package or specific features
Support	Telephone, e-mail, web
Upgrades, maintenance	Regularity of new versions and maintenance releases that address customer needs
Track record	Stability, history, customer relations

Implementation and capability are what is important. As a second example, most packages offer a runtime license, but these vary considerably in price and features.

6. Simulation users ask whether the simulation model can link to and use code or routines written in external languages such as C, C++, or Java. This is a good feature, especially when the external routines already exist and are suitable for the purpose at hand. However, the more important question is whether the simulation package and language are sufficiently powerful to avoid having to write logic in any external language.
7. There may be a significant trade-off between the graphical model-building environments and ones based on a simulation language. Graphical model-building removes the learning curve due to language syntax, but it does not remove the need for procedural logic in most real-world models and the debugging to get it right. Beware of "no programming required" unless either the package is a near-perfect fit to your problem domain or programming (customized procedural logic) is possible with the supplied blocks, nodes, or process-flow diagram—in which case "no programming required" refers to syntax only and not the development of procedural logic.

4.3 AN EXAMPLE SIMULATION

Example 4.1: The Checkout Counter: A Typical Single-Server Queue

The system, a grocery checkout counter, is modeled as a single-server queue. The simulation will run until 1000 customers have been served. In addition, assume that the interarrival times of customers are exponentially distributed with a mean of 4.5 minutes, and that the service times are (approximately) normally distributed with a mean of 3.2 minutes and a standard deviation of 0.6 minute. (The approximation is that service times are always positive.) When the cashier is busy, a queue forms with no customers turned away. This example was simulated manually in Examples 3.3 and 3.4 by using the event-scheduling point of view. The model contains two events: the arrival and departure events. Figures 3.5 and 3.6 provide the event logic.

The following three sections illustrate the simulation of this single-server queue in Java, GPSS/H, and SSF. Although this example is much simpler than models that arise in the study of complex systems, its simulation contains the essential components of all discrete-event simulations.

4.4 SIMULATION IN JAVA

Java is a widely used programming language that has been used extensively in simulation. It does not, however, provide any facilities directly aimed at aiding the simulation analyst, who therefore must program all details of the event-scheduling/time-advance algorithm, the statistics-gathering capability, the generation of samples from specified probability distributions, and the report generator. However, the runtime library does provide a random-number generator. Unlike with FORTRAN or C, the object-orientedness of Java does support modular construction of large models. For the most part, the special-purpose simulation languages hide the details of event scheduling, whereas in Java all the details must be explicitly programmed. However, to a certain extent, simulation libraries such as SSF (Cowie 1999) alleviate the development burden by providing access to standardized simulation functionality and by hiding low-level scheduling minutiae.

There are many online resources for learning Java; we assume a prior working knowledge of the language. Any discrete-event simulation model written in Java contains the components discussed in Section 4.3: system state, entities and attributes, sets, events, activities and delays, and the components listed shortly. To facilitate development and debugging, it is best to organize the Java model in a modular fashion by using methods. The following components are common to almost all models written in Java:

Clock A variable defining simulated time
Initialization method A method to define the system state at time 0
Min-time event method A method that identifies the imminent event, that is, the element of the future event list (`FutureEventList`) that has the smallest time-stamp
Event methods For each event type, a method to update system state (and cumulative statistics) when that event occurs
Random-variate generators Methods to generate samples from desired probability distributions
Main program To maintain overall control of the event-scheduling algorithm
Report generator A method that computes summary statistics from cumulative statistics and prints a report at the end of the simulation

The overall structure of a Java simulation program is shown in Figure 4.1. This flowchart is an expansion of the event-scheduling/time-advance algorithm outlined in Figure 3.2. (The steps mentioned in Figure 4.1 refer to the five steps in Figure 3.2.)

The simulation begins by setting the simulation `Clock` to zero, initializing cumulative statistics to zero, generating any initial events (there will always beat least one) and placing them on the `FutureEventList`,

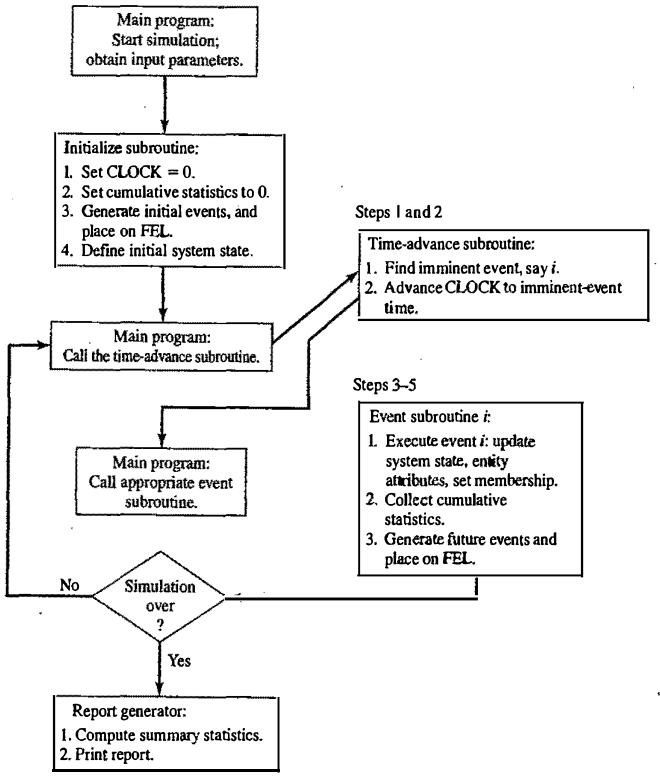

Figure 4.1 Overall structure of an event-scheduling simulation program.

and defining the system state at time 0. The simulation program then cycles, repeatedly passing the current least-time event to the appropriate event methods until the simulation is over. At each step, after finding the imminent event but before calling the event method, the simulation `Clock` is advanced to the time of the imminent event. (Recall that, during the simulated time between the occurrence of two successive events, the system state and entity attributes do not change in value. Indeed, this is the definition of discrete-event simulation: The system state changes only when an event occurs.) Next, the appropriate event method is called to execute the imminent event, update cumulative statistics, and generate future events (to be placed on the `FutureEventList`). Executing the imminent event means that the system state, entity attributes, and set membership are changed to reflect the fact that the event has occurred. Notice that all actions in an event method take place at one instant of simulated time. The value of the variable `Clock` does not change in an event method. If the simulation is not over, control passes again to the time-advance method, then to the appropriate event method, and so on. When the simulation is over, control passes to the report generator, which computes the desired summary statistics from the collected cumulative statistics and prints a report.

The efficiency of a simulation model in terms of computer runtime is determined to a large extent by the techniques used to manipulate the `FutureEventList` and other sets. As discussed earlier in Section 4.3,

removal of the imminent event and addition of a new event are the two main operations performed on the `FutureEventList`. Java includes general, efficient data structures for searching and priority lists; it is usual to build a customized interface to these to suit the application. In the example to follow, we use customized interfaces to implement the event list and the list of waiting customers. The underlying priority-queue organization is efficient, in the sense of having access costs that grow only in the logarithm of the number of elements in the list.

Example 4.2: Single-Server Queue Simulation in Java

The grocery checkout counter, defined in detail in Example 4.1, is now simulated by using Java. A version of this example was simulated manually in Examples 3.3 and 3.4, where the system state, entities and attributes, sets, events, activities, and delays were analyzed and defined.

Class `Event` represents an event. It stores a code for the event type (`arrival` or `departure`), and the event time-stamp. It has associated methods (functions) for creating an event and accessing its data. It also has an associated method **compareTo**, which compares the event with another (passed as an argument) and reports whether the first event should be considered to be smaller, equal, or larger than the argument event. The methods for this model and the flow of control are shown in Figure 4.2, which is an adaptation of Figure 4.1 for this particular problem. Table 4.6 lists the variables used for system state, entity attributes and sets, activity durations, and cumulative and summary statistics; the functions used to generate samples from the exponential and normal distributions; and all the other methods needed.

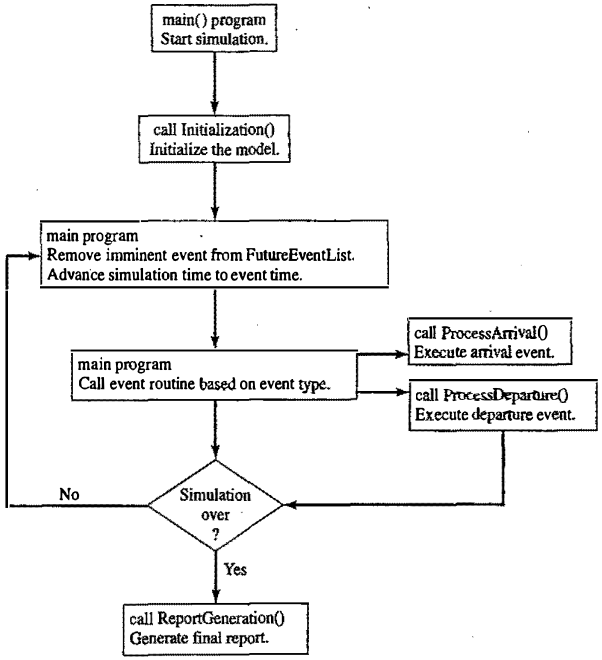

Figure 4.2 Overall structure of Java simulation of a single-server queue.

Table 4.6 Definitions of Variables, Functions, and Subroutines in the Java Model of the Single-Server Queue

Variables	*Description*
System state	
`QueueLength`	Number of customers enqueued (but not in service) at current simulated time
`NumberInService`	Number being served at current simulated time
Entity attributes and sets	
`Customers`	FCFS Queue of customers in system
Future event list	
`FutureEventList`	Priority-ordered list of pending events
Activity durations	
`MeanInterArrivalTime`	The interarrival time between the previous customer's arrival and the next arrival
`MeanServiceTime`	The service time of the most recent customer to begin service
Input parameters	
`MeanInterarrivalTime`	Mean interarrival time (4.5 minutes)
`MeanServiceTime`	Mean service time (3.2 minutes)
`SIGMA`	Standard deviation of service time (0.6 minute)
`TotalCustomers`	The stopping criterion— number of customers to be served (1000)
Simulation variables	
`Clock`	The current value of simulated time
Statistical Accumulators	
`LastEventTime`	Time of occurrence of the last event
`TotalBusy`	Total busy time of server (so far)
`MaxQueueLength`	Maximum length of waiting line (so far)
`SumResponseTime`	Sum of customer response times for all customers who have departed (so far)
`NumberOfDepartures`	Number of departures (so far)
`LongService`	Number of customers who spent 4 or more minutes at the checkout counter (so far)
Summary statistics	
`RHO = BusyTime/Clock`	Proportion of time server is busy (here the value of `Clock` is the final value of simulated time)
`AVGR`	Average response time (equal to `SumResponseTime/TotalCustomers`)
`PC4`	Proportion of customers who spent 4 or more minutes at the checkout counter

Functions	*Description*
`exponential(mu)`	Function to generate samples from an exponential distribution with mean `mu`
`normal(xmu,SIGMA)`	Function to generate samples from a normal distribution with mean `xmu` and standard deviation `SIGMA`

(*continued overleaf*)

Table 4.6 (*continued*)

Methods	*Description*
`Initialization`	Initialization method
`ProcessArrival`	Event method that executes the arrival event
`ProcessDeparture`	Event method that executes the departure event
`ReportGeneration`	Report generator

The entry point of the program and the location of the control logic is through class `Sim`, shown in Figure 4.3. Variables of classes `EventList` and `Queue` are declared. As these classes are all useful for programs other than `Sim`, their declarations are given in other files, per Java rules. A variable of the Java built-in class `Random` is also declared; instances of this class provided random-number streams. The main method controls the overall flow of the event-scheduling/time-advance algorithm.

```
class Sim {

// Class Sim variables
public static double Clock, MeanInterArrivalTime, MeanServiceTime,
       SIGMA, LastEventTime, TotalBusy, MaxQueueLength, SumResponseTime;
public static long NumberOfCustomers, QueueLength, NumberInService,
       TotalCustomers, NumberOfDepartures, LongService;

public final static int arrival = 1;
public final static int departure = 2;

public static EventList FutureEventList;
public static Queue Customers;
public static Random stream;

public static void main(String argv[]) {

  MeanInterArrivalTime = 4.5; MeanServiceTime = 3.2;
  SIGMA                = 0.6; TotalCustomers  = 1000;
  long seed            = Long.parseLong(argv[0]);
  stream = new Random(seed);            // initialize rng stream
  FutureEventList = new EventList();
  Customers = new Queue();

  Initialization();

  // Loop until first "TotalCustomers" have departed
  while(NumberOfDepartures < TotalCustomers ) {
    Event evt = (Event) FutureEventList.getMin();     // get imminent event
    FutureEventList.dequeue();                        // be rid of it
    Clock = evt.get_time();                           // advance in time
    if( evt.get_type() == arrival ) ProcessArrival(evt);
    else ProcessDeparture(evt);
    }
  ReportGeneration();
}
```

Figure 4.3 Java main program for the single-server queue simulation.

The main program method first gives values to variables describing model parameters; it creates instances of the random-number generator, event list, and customer queue; and then it calls method `Initialization` to initialize other variables, such as the statistics-gathering variables. Control then enters a loop which is exited only after `TotalCustomers` customers have received service. Inside the loop, a copy of the imminent event is obtained by calling the `getMin` method of the priority queue, and then that event is removed from the event list by a call to dequeue. The global simulation time `Clock` is set to the time-stamp contained in the imminent event, and then either `ProcessArrival` or `ProcessDeparture` is called, depending on the type of the event. When the simulation is finally over, a call is made to method `ReportGeneration` to create and print out the final report.

A listing for the `Sim` class method `Initialization` is given in Figure 4.4. The simulation clock, system state, and other variables are initialized. Note that the first arrival event is created by generating a local `Event` variable whose constructor accepts the event's type and time. The event time-stamp is generated randomly by a call to `Sim` class method `exponential` and is passed to the random-number stream to use with the mean of the exponential distribution from which to sample. The event is inserted into the future event list by calling method `enqueue`. This logic assumes that the system is empty at simulated time `Clock=0`, so that no departure can be scheduled. It is straightforward to modify the code to accommodate alternative starting conditions by adding events to `FutureEventList` and `Customers` as needed.

Figure 4.5 gives a listing of `Sim` class method `ProcessArrival`, which is called to process each arrival event. The basic logic of the arrival event for a single-server queue was given in Figure 3.5 (where LQ corresponds to `QueueLength` and LS corresponds to `NumberInService`). First, the new arrival is added to the queue `Customers` of customers in the system. Next, if the server is idle (`NumberInService==0`) then the new customer is to go immediately into service, so `Sim` class method `ScheduleDeparture` is called to do that scheduling. An arrival to an idle queue does not update the cumulative statistics, except possibly the maximum queue length. An arrival to a busy queue does *not* cause the scheduling of a departure, but does increase the total busy time by the amount of simulation time between the current event and the one immediately preceding it (because, if the server is busy now, it had to have had at least one customer in service by the end of processing the previous event). In either case, a new arrival is responsible for scheduling the next arrival, one random interarrival time into the future. An arrival event is created with simulation time equal to the current `Clock` value plus an exponential increment, that event is inserted into the future event list, the variable `LastEventTime` recording the time of the last event processed is set to the current time, and control is returned to the `main` method of class `Sim`.

```
public static void Initialization()    {
  Clock = 0.0;
  QueueLength = 0;
  NumberInService = 0;
  LastEventTime = 0.0;
  TotalBusy = 0 ;
  MaxQueueLength = 0;
  SumResponseTime = 0;
  NumberOfDepartures = 0;
  LongService = 0;

  // create first arrival event
  Event evt =
   new Event(arrival, exponential( stream, MeanInterArrivalTime));
  FutureEventList.enqueue( evt );
}
```

Figure 4.4 Java initialization method for the single-server queue simulation.


```
public static void ProcessArrival(Event evt)  {
  Customers.enqueue(evt);
  QueueLength++;
  // if the server is idle, fetch the event, do statistics
  // and put into service
  if( NumberInService == 0) ScheduleDeparture();
  else TotalBusy += (Clock - LastEventTime);     // server is busy

  // adjust max queue length statistics
  if (MaxQueueLength < QueueLength) MaxQueueLength = QueueLength;

  // schedule the next arrival
  Event next_arrival =
   new Event(arrival, Clock+exponential(stream,MeanInterArrivalTime));
  FutureEventList.enqueue( next_arrival );
  LastEventTime = Clock;
}
```

Figure 4.5 Java arrival event method for the single-server queue simulation.

`Sim` class method `ProcessDeparture`, which executes the departure event, is listed in Figure 4.6, as is method `ScheduleDeparture`. A flowchart for the logic of the departure event was given in Figure 3.6. After removing the event from the queue of all customers, the number in service is examined. If there are customers waiting, then the departure of the next one to enter service is scheduled. Then, cumulative statistics recording the sum of all response times, sum of busy time, number of customers who used more than 4 minutes of service time, and number of departures are updated. (Note that the maximum queue length cannot change in value when a departure occurs.) Notice that customers are removed from `Customers` in

```
public static void ScheduleDeparture()  {
  double ServiceTime;
  // get the job at the head of the queue
  while (( ServiceTime = normal(stream,MeanServiceTime, SIGMA)) < 0 );
  Event depart = new Event(departure,Clock+ServiceTime);
  FutureEventList.enqueue( depart );
  NumberInService = 1;
  QueueLength--;
  }

public static void ProcessDeparture(Event e) {
  // get the customer description
  Event finished = (Event) Customers.dequeue();
  // if there are customers in the queue then schedule
  // the departure of the next one
   if( QueueLength > 0 ) ScheduleDeparture();
   else NumberInService = 0;
   // measure the response time and add to the sum
   double response = (Clock - finished.get_time());
   SumResponseTime += response;
   if( response > 4.0 ) LongService++; // record long service
   TotalBusy += (Clock - LastEventTime );
   NumberOfDepartures++;
   LastEventTime = Clock;
  }
```

Figure 4.6 Java departure event method for the single-server queue simulation.

FIFO order; hence, the response time `response` of the departing customer can be computed by subtracting the arrival time of the job leaving service (obtained from the copy of the arrival event removed from the `Customers` queue) from the current simulation time. After the incrementing of the total number of departures and the saving of the time of this event, control is returned to the main program.

Figure 4.6 also gives the logic of method `ScheduleDeparture`, called by both `ProcessArrival` and `ProcessDeparture` to put the next customer into service. The `Sim` class method `normal`, which generates normally distributed service times, is called until it produces a nonnegative sample. A new event with type `departure` is created, with event time equal to the current simulation time plus the service time just sampled. That event is pushed onto `FutureEventList`, the number in service is set to one, and the number waiting (`QueueLength`) is decremented to reflect the fact that the customer entering service is waiting no longer.

The report generator, `Sim` class method `ReportGeneration`, is listed in Figure 4.7. The summary statistics, RHO, AVGR, and PC4, are computed by the formulas in Table 4.6; then the input parameters are printed, followed by the summary statistics. It is a good idea to print the input parameters at the end of the simulation, in order to verify that their values are correct and that these values have not been inadvertently changed.

Figure 4.8 provides a listing of `Sim` class methods exponential and `normal`, used to generate random variates. Both of these functions call method `nextDouble`, which is defined for the built-in Java `Random` class generates a random number uniformly distributed on the (0,1) interval. We use `Random` here for simplicity of explanation; superior random-number generators can be built by hand, as described in Chapter 7.

```
public static void ReportGeneration() {
double RHO  = TotalBusy/Clock;
double AVGR = SumResponseTime/TotalCustomers;
double PC4  = ((double)LongService)/TotalCustomers;


System.out.print( "SINGLE SERVER QUEUE SIMULATION ");
System.out.println( "- GROCERY STORE CHECKOUT COUNTER ");
System.out.println( "\tMEAN INTERARRIVAL TIME                "
   + MeanInterArrivalTime );
System.out.println( "\tMEAN SERVICE TIME                     "
   + MeanServiceTime );
System.out.println( "\tSTANDARD DEVIATION OF SERVICE TIMES   "
   + SIGMA );
System.out.println( "\tNUMBER OF CUSTOMERS SERVED            "
   + TotalCustomers );
System.out.println();
System.out.println( "\tSERVER UTILIZATION                    "
   + RHO );
System.out.println( "\tMAXIMUM LINE LENGTH                   "
   + MaxQueueLength );
System.out.println( "\tAVERAGE RESPONSE TIME                 "
   + AVGR + "  MINUTES" );
System.out.println( "\tPROPORTION WHO SPEND FOUR ");
System.out.println( "\t MINUTES OR MORE IN SYSTEM            "
   + PC4 );
System.out.println( "\tSIMULATION RUNLENGTH                  "
   + Clock + " MINUTES" );
System.out.println( "\tNUMBER OF DEPARTURES                  "
   + TotalCustomers );
 }
```

Figure 4.7 Java report generator for the single-server queue simulation.


```
public static double exponential(Random rng, double mean) {
 return -mean*Math.log( rng.nextDouble() );
}

public static double SaveNormal;
public static int NumNormals = 0;
public static final double PI = 3.1415927 ;

public static double normal(Random rng, double mean, double sigma) {
        double ReturnNormal;
        // should we generate two normals?
        if(NumNormals == 0 ) {
          double r1 = rng.nextDouble();
          double r2 = rng.nextDouble();
          ReturnNormal = Math.sqrt(-2*Math.log(r1))*Math.cos(2*PI*r2);
          SaveNormal   = Math.sqrt(-2*Math.log(r1))*Math.sin(2*PI*r2);
          NumNormals = 1;
        } else {
          NumNormals = 0;
          ReturnNormal = SaveNormal;
        }
        return ReturnNormal*sigma + mean ;
}
```

Figure 4.8 Random-variate generators for the single-server queue simulation.

The techniques for generating exponentially and normally distributed random variates, discussed in Chapter 8, are based on first generating a $U(0,1)$ random number. For further explanation, the reader is referred to Chapters 7 and 8.

The output from the grocery-checkout-counter simulation is shown in Figure 4.9. It should be emphasized that the output statistics are estimates that contain random error. The values shown are influenced by the particular random numbers that happened to have been used, by the initial conditions at time 0, and by the run length (in this case, 1000 departures). Methods for estimating the standard error of such estimates are discussed in Chapter 11.

In some simulations, it is desired to stop the simulation after a fixed length of time, say TE = 12 hours = 720 minutes. In this case, an additional event type, `stop` event, is defined and is scheduled to occur by scheduling a stop event as part of simulation initialization. When the stopping event does occur, the cumulative

```
SINGLE SERVER QUEUE SIMULATION - GROCERY STORE CHECKOUT COUNTER
        MEAN INTERARRIVAL TIME                  4.5
        MEAN SERVICE TIME                       3.2
        STANDARD DEVIATION OF SERVICE TIMES     0.6
        NUMBER OF CUSTOMERS SERVED              1000

        SERVER UTILIZATION                      0.671
        MAXIMUM LINE LENGTH                     9.0
        AVERAGE RESPONSE TIME                   6.375 MINUTES
        PROPORTION WHO SPEND FOUR
         MINUTES OR MORE IN SYSTEM              0.604
        SIMULATION RUNLENGTH                    4728.936 MINUTES
        NUMBER OF DEPARTURES                    1000
```

Figure 4.9 Output from the Java single-server queue simulation.

statistics will be updated and the report generator called. The main program and method `Initialization` will require minor changes. Exercise 1 asks the reader to make these changes. Exercise 2 considers balking of customers.

4.5 SIMULATION IN GPSS

GPSS is a highly structured, special-purpose simulation programming language based on the process-interaction approach and oriented toward queueing systems. A block diagram provides a convenient way to describe the system being simulated. There are over 40 standard blocks in GPSS. Entities called transactions may be viewed as flowing through the block diagram. Blocks represent events, delays, and other actions that affect transaction flow. Thus, GPSS can be used to model any situation where transactions (entities, customers, units of traffic) are flowing through a system (e.g., a network of queues, with the queues preceding scarce resources). The block diagram is converted to block statements, control statements are added, and the result is a GPSS model.

The first version of GPSS was released by IBM in 1961. It was the first process-interaction simulation language and became popular; it has been implemented anew and improved by many parties since 1961, with GPSS/H being the most widely used version in use today. Example 4.3 is based on GPSS/H.

GPSS/H is a product of Wolverine Software Corporation, Annandale, VA (Banks, Carson, and Sy, 1995; Henriksen, 1999). It is a flexible, yet powerful tool for simulation. Unlike the original IBM implementation, GPSS/H includes built-in file and screen I/O, use of an arithmetic expression as a block operand, an interactive debugger, faster execution, expanded control statements, ordinary variables and arrays, a floating-point clock, built-in math functions, and built-in random-variate generators.

The animator for GPSS/H is Proof Animation™, another product of Wolverine Software Corporation (Henriksen, 1999). Proof Animation provides a 2-D animation, usually based on a scale drawing. It can run in postprocessed mode (after the simulation has finished running) or concurrently. In postprocessed mode, the animation is driven by two files: the layout file for the static background, and a trace file that contains commands to make objects move and produce other dynamic events. It can work with any simulation package that can write the ASCII trace file. Alternately, it can run concurrently with the simulation by sending the trace file commands as messages, or it can be controlled directly by using its DLL (dynamic link library) version.

Example 4.3: Single-Server Queue Simulation in GPSS/H

Figure 4.10 exhibits the block diagram and Figure 4.11 the GPSS program for the grocery-store checkout-counter model described in Example 4.2. Note that the program (Figure 4.11) is a translation of the block diagram together with additional definition and control statements.

In Figure 4.10, the GENERATE block represents the arrival event, with the interarrival times specified by RVEXPO(1,*&IAT*). RVEXPO stands for "random variable, exponentially distributed," the 1 indicates the random-number stream to use, and *&IAT* indicates that the mean time for the exponential distribution comes from a so-called ampervariable *&IAT*. Ampervariable names begin with the "&" character; Wolverine added ampervariables to GPSS because the original IBM implementation had limited support for ordinary global variables, with no user freedom for naming them. (In the discussion that follows, all nonreserved words are shown in italics.)

The next block is a QUEUE with a queue named *SYSTIME*. It should be noted that the QUEUE block is not needed for queues or waiting lines to form in GPSS. The true purpose of the QUEUE block is to work in conjunction with the DEPART block to collect data on queues or any other subsystem. In Example 4.3,

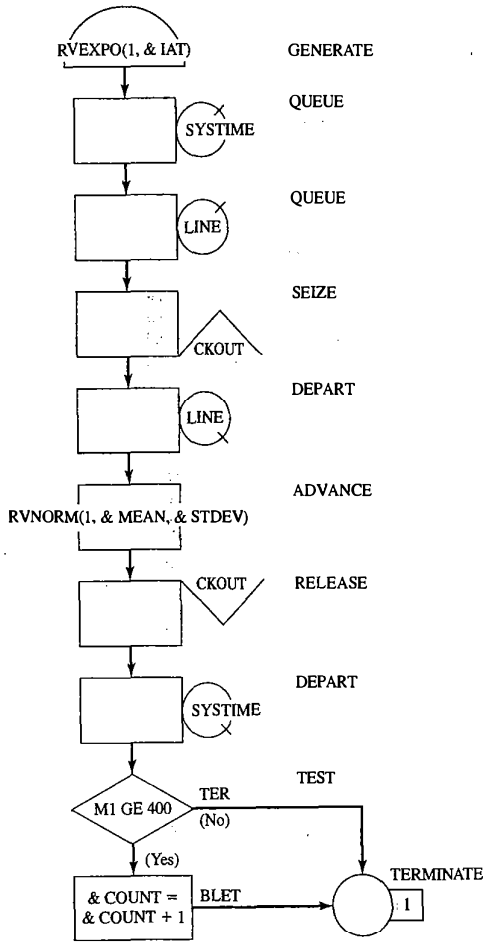

Figure 4.10 GPSS block diagram for the single-server queue simulation.

we want to measure the system response time—that is, the time a transaction spends in the system. Placing a QUEUE block at the point that transactions enter the system and placing the counterpart of the QUEUE block, the DEPART block, at the point that the transactions complete their processing causes the response times to be collected automatically. The purpose of the DEPART block is to signal the end of data collection for an individual transaction. The QUEUE and DEPART block combination is not necessary for queues to be modeled, but rather is used for statistical data collection.

The next QUEUE block (with name *LINE*) begins data collection for the waiting line before the cashier. The customers may or may not have to wait for the cashier. Upon arrival to an idle checkout counter, or after