Simulation Modeling and Analysis

THIRD EDITION

Averill M. Law

President
Averill M. Law & Associates, Inc.
Tucson, Arizona, USA
www.averill-law.com

W. David Kelton

Professor
Department of Quantitative Analysis and Operations Management
College of Business Administration
University of Cincinnati
Cincinnati, Ohio, USA
www.econqa.cba.uc.edu/~keltond

Boston Burr Ridge, IL Dubuque, IA Madison, WI
New York San Francisco St. Louis
Bangkok Bogotá Caracas Lisbon London Madrid Mexico City

CHAPTER 1

Basic Simulation Modeling

Recommended sections for a first reading: 1.1 through 1.4 (except 1.4.8), 1.7, 1.9

1.1 THE NATURE OF SIMULATION

This is a book about techniques for using computers to imitate, or *simulate*, the operations of various kinds of real-world facilities or processes. The facility or process of interest is usually called a *system*, and in order to study it scientifically we often have to make a set of assumptions about how it works. These assumptions, which usually take the form of mathematical or logical relationships, constitute a *model* that is used to try to gain some understanding of how the corresponding system behaves.

If the relationships that compose the model are simple enough, it may be possible to use mathematical methods (such as algebra, calculus, or probability theory) to obtain *exact* information on questions of interest; this is called an *analytic* solution. However, most real-world systems are too complex to allow realistic models to be evaluated analytically, and these models must be studied by means of simulation. In a *simulation* we use a computer to evaluate a model *numerically*, and data are gathered in order to *estimate* the desired true characteristics of the model.

As an example of the use of simulation, consider a manufacturing company that is contemplating building a large extension onto one of its plants but is not sure if the potential gain in productivity would justify the construction cost. It certainly would not be cost-effective to build the extension and then remove it later if it does not work out. However, a careful simulation study could shed some light on the question by simulating the operation of the plant as it currently exists and as it *would* be *if* the plant were expanded.

Application areas for simulation are numerous and diverse. Below is a list of some particular kinds of problems for which simulation has been found to be a useful and powerful tool:

- Designing and analyzing manufacturing systems
- Evaluating military weapons systems or their logistics requirements
- Determining hardware requirements or protocols for communications networks
- Determining hardware and software requirements for a computer system
- Designing and operating transportation systems such as airports, freeways, ports, and subways
- Evaluating designs for service organizations such as call centers, fast-food restaurants, hospitals, and post offices
- Reengineering of business processes
- Determining ordering policies for an inventory system
- Analyzing financial or economic systems

Simulation is one of the most widely used operations-research and management-science techniques, if not *the* most widely used. One indication of this is the Winter Simulation Conference, which attracts 600 to 700 people every year. In addition, there are several simulation vendor users' conferences with more than 100 participants per year.

There are also several surveys related to the use of operations-research techniques. For example, Lane, Mansour, and Harpell (1993) reported from a longitudinal study, spanning 1973 through 1988, that simulation was consistently ranked as one of the three most important "operations-research techniques." The other two were "math programming" (a catch-all term that includes many individual techniques such as linear programming, nonlinear programming, etc.) and "statistics" (which is not an operations-research technique per se). Gupta (1997) analyzed 1294 papers from the journal *Interfaces* (one of the leading journals dealing with applications of operations research) from 1970 through 1992, and found that simulation was second only to "math programming" among 13 techniques considered.

There have been, however, several impediments to even wider acceptance and usefulness of simulation. First, models used to study large-scale systems tend to be very complex, and writing computer programs to execute them can be an arduous task indeed. This task has been made much easier in recent years by the development of excellent software products that automatically provide many of the features needed to "program" a simulation model. A second problem with simulation of complex systems is that a large amount of computer time is sometimes required. However, this difficulty is becoming much less severe as computers become faster and cheaper. Finally, there appears to be an unfortunate impression that simulation is just an exercise in computer programming, albeit a complicated one. Consequently, many simulation "studies" have been composed of heuristic model building, coding, and a single run of the program to obtain "the answer." We fear that this attitude, which neglects the important issue of how a properly coded model should be used to make inferences about the system of interest, has doubtless led to erroneous conclusions being drawn from many simulation studies. These questions of simulation *methodology*, which are largely independent of the software and hardware used, form an integral part of the latter chapters of this book.

In the remainder of this chapter (as well as in Chap. 2) we discuss systems and models in considerably greater detail and then show how to write computer programs in general-purpose languages to simulate systems of varying degrees of complexity. All of the computer code shown in this chapter can be downloaded from http://www.mhhe.com/lawkelton.

1.2 SYSTEMS, MODELS, AND SIMULATION

A *system* is defined to be a collection of entities, e.g., people or machines, that act and interact together toward the accomplishment of some logical end. [This definition was proposed by Schmidt and Taylor (1970).] In practice, what is meant by "the system" depends on the objectives of a particular study. The collection of entities that comprise a system for one study might be only a subset of the overall system for another. For example, if one wants to study a bank to determine the number of tellers needed to provide adequate service for customers who want just to cash a check or make a savings deposit, the system can be defined to be that portion of the bank consisting of the tellers and the customers waiting in line or being served. If, on the other hand, the loan officer and the safety deposit boxes are to be included, the definition of the system must be expanded in an obvious way. [See also Fishman (1978, p. 3).] We define the *state* of a system to be that collection of variables necessary to describe a system at a particular time, relative to the objectives of a study. In a study of a bank, examples of possible state variables are the number of busy tellers, the number of customers in the bank, and the time of arrival of each customer in the bank.

We categorize systems to be of two types, discrete and continuous. A *discrete* system is one for which the state variables change instantaneously at separated points in time. A bank is an example of a discrete system, since state variables—e.g., the number of customers in the bank—change only when a customer arrives or when a customer finishes being served and departs. A *continuous* system is one for which the state variables change continuously with respect to time. An airplane moving through the air is an example of a continuous system, since state variables such as position and velocity can change continuously with respect to time. Few systems in practice are wholly discrete or wholly continuous; but since one type of change predominates for most systems, it will usually be possible to classify a system as being either discrete or continuous.

At some point in the lives of most systems, there is a need to study them to try to gain some insight into the relationships among various components, or to predict performance under some new conditions being considered. Figure 1.1 maps out different ways in which a system might be studied.

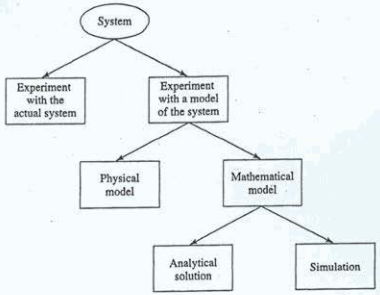


FIGURE 1.1
Ways to study a system.

- *Experiment with the Actual System vs. Experiment with a Model of the System.* If it is possible (and cost-effective) to alter the system physically and then let it operate under the new conditions, it is probably desirable to do so, for in this case there is no question about whether what we study is valid. However, it is rarely feasible to do this, because such an experiment would often be too costly or too disruptive to the system. For example, a bank may be contemplating reducing the number of tellers to decrease costs, but actually trying this could lead to long customer delays and alienation. More graphically, the "system" might not even exist, but we nevertheless want to study it in its various proposed alternative configurations to see how it should be built in the first place; examples of this situation might be a proposed communications network, or a strategic nuclear weapons system. For these reasons, it is usually necessary to build a *model* as a representation of the system and study it as a surrogate for the actual system. When using a model, there is always the question of whether it accurately reflects the system for the purposes of the decisions to be made; this question of model *validity* is taken up in detail in Chap. 5.
- *Physical Model vs. Mathematical Model.* To most people, the word "model" evokes images of clay cars in wind tunnels, cockpits disconnected from their airplanes to be used in pilot training, or miniature supertankers scurrying about in a swimming pool. These are examples of *physical* models (also called *iconic* models), and are not typical of the kinds of models that are usually of interest in operations research and systems analysis. Occasionally, however, it has been found useful to build physical models to study engineering or management systems; examples include tabletop scale models of material-handling systems, and in at least one case a full-scale physical model of a fast-food restaurant inside a warehouse, complete with full-scale, real (and presumably hungry) humans [see Swart and Donno (1981)]. But the vast majority of models built for such purposes are *mathematical*, representing a system in terms of logical and quantitative relationships that are then manipulated and changed to see how the model reacts, and thus how the system *would* react—*if* the mathematical model is a valid one. Perhaps the simplest example of a mathematical model is the familiar relation $d = rt$, where r is the rate of travel, t is the time spent traveling, and d is the distance traveled. This might provide a valid model in one instance (e.g., a space probe to another planet after it has attained its flight velocity) but a very poor model for other purposes (e.g., rush-hour commuting on congested urban freeways).
- *Analytical Solution vs. Simulation.* Once we have built a mathematical model, it must then be examined to see how it can be used to answer the questions of interest about the system it is supposed to represent. If the model is simple enough, it may be possible to work with its relationships and quantities to get an exact, *analytical* solution. In the $d = rt$ example, if we know the distance to be traveled and the velocity, then we can work with the model to get $t = d/r$ as the time that will be required. This is a very simple, closed-form solution obtainable with just paper and pencil, but some analytical solutions can become extraordinarily complex, requiring vast computing resources; inverting a large nonsparse matrix is a well-known example of a situation in which there is an analytical formula known in principle, but obtaining it numerically in a given instance is far from trivial. If an analytical solution to a mathematical model is available and is computationally efficient, it is usually desirable to study the model in this way rather than via a simulation. However, many systems are highly complex, so that valid mathematical models of them are themselves complex, precluding any possibility of an analytical solution. In this case, the model must be studied by means of *simulation*, i.e., numerically exercising the model for the inputs in question to see how they affect the output measures of performance.

While there may be a small element of truth to pejorative old saws such as "method of last resort" sometimes used to describe simulation, the fact is that we are very quickly led to simulation in most situations, due to the sheer complexity of the systems of interest and of the models necessary to represent them in a valid way.

Given, then, that we have a mathematical model to be studied by means of simulation (henceforth referred to as a *simulation model*), we must then look for particular tools to do this. It is useful for this purpose to classify simulation models along three different dimensions:

- *Static vs. Dynamic Simulation Models.* A *static* simulation model is a representation of a system at a particular time, or one that may be used to represent a system in which time simply plays no role; examples of static simulations are Monte Carlo models, discussed in Sec. 1.8.3. On the other hand, a *dynamic* simulation model represents a system as it evolves over time, such as a conveyor system in a factory.

- *Deterministic vs. Stochastic Simulation Models.* If a simulation model does not contain any probabilistic (i.e., random) components, it is called *deterministic*; a complicated (and analytically intractable) system of differential equations describing a chemical reaction might be such a model. In deterministic models, the output is "determined" once the set of input quantities and relationships in the model have been specified, even though it might take a lot of computer time to evaluate what it is. Many systems, however, must be modeled as having at least some random input components, and these give rise to *stochastic* simulation models. (For an example of the danger of ignoring randomness in modeling a system, see Sec. 4.7.) Most queueing and inventory systems are modeled stochastically. Stochastic simulation models produce output that is itself random, and must therefore be treated as only an estimate of the true characteristics of the model; this is one of the main disadvantages of simulation (see Sec. 1.9) and is dealt with in Chaps. 9 through 12 of this book.
- *Continuous vs. Discrete Simulation Models.* Loosely speaking, we define *discrete* and *continuous* simulation models analogously to the way discrete and continuous systems were defined above. More precise definitions of discrete (event) simulation and continuous simulation are given in Secs. 1.3 and 1.8, respectively. It should be mentioned that a discrete model is not always used to model a discrete system, and vice versa. The decision whether to use a discrete or a continuous model for a particular system depends on the specific objectives of the study. For example, a model of traffic flow on a freeway would be discrete if the characteristics and movement of individual cars are important. Alternatively, if the cars can be treated "in the aggregate," the flow of traffic can be described by differential equations in a continuous model. More discussion on this issue can be found in Sec. 5.2, and in particular in Example 5.2.

The simulation models we consider in the remainder of this book, except for those in Sec. 1.8, will be discrete, dynamic, and stochastic and will henceforth be called *discrete-event simulation models*. (Since deterministic models are a special case of stochastic models, the restriction to stochastic models involves no loss of generality.)

1.3 DISCRETE-EVENT SIMULATION

Discrete-event simulation concerns the modeling of a system as it evolves over time by a representation in which the state variables change instantaneously at separate points in time. (In more mathematical terms, we might say that the system can change at only a *countable* number of points in time.) These points in time are the ones at which an event occurs, where an *event* is defined as an instantaneous occurrence that may change the state of the system. Although discrete-event simulation could conceptually be done by hand calculations, the amount of data that must be stored and manipulated for most real-world systems dictates that discrete-event simulations be done on a digital computer. (In Sec. 1.4.2 we carry out a small hand simulation, merely to illustrate the logic involved.)

EXAMPLE 1.1. Consider a service facility with a single server—e.g., a one-operator barbershop or an information desk at an airport—for which we would like to estimate the (expected) average delay in queue (line) of arriving customers, where the delay in queue of a customer is the length of the time interval from the instant of his arrival at the facility to the instant he begins being served. For the objective of estimating the average delay of a customer, the state variables for a discrete-event simulation model of the facility would be the status of the server, i.e., either idle or busy, the number of customers waiting in queue to be served (if any), and the time of arrival of each person waiting in queue. The status of the server is needed to determine, upon a customer's arrival, whether the customer can be served immediately or must join the end of the queue. When the server completes serving a customer, the number of customers in the queue is used to determine whether the server will become idle or begin serving the first customer in the queue. The time of arrival of a customer is needed to compute his delay in queue, which is the time he begins being served (which will be known) minus his time of arrival. There are two types of events for this system: the arrival of a customer and the completion of service for a customer, which results in the customer's departure. An arrival is an event since it causes the (state variable) server status to change from idle to busy or the (state variable) number of customers in the queue to increase by 1. Correspondingly, a departure is an event because it causes the server status to change from busy to idle or the number of customers in the queue to decrease by 1. We show in detail how to build a discrete-event simulation model of this single-server queueing system in Sec. 1.4.

In the above example both types of events actually changed the state of the system, but in some discrete-event simulation models events are used for purposes that do not actually effect such a change. For example, an event might be used to schedule the end of a simulation run at a particular time (see Sec. 1.4.7) or to schedule a decision about a system's operation at a particular time (see Sec. 1.5) and might not actually result in a change in the state of the system. This is why we originally said that an event *may* change the state of a system.

1.3.1 Time-Advance Mechanisms

Because of the dynamic nature of discrete-event simulation models, we must keep track of the current value of simulated time as the simulation proceeds, and we also need a mechanism to advance simulated time from one value to another. We call the variable in a simulation model that gives the current value of simulated time the *simulation clock*. The unit of time for the simulation clock is never stated explicitly when a model is written in a general-purpose language such as FORTRAN or C, and it is assumed to be in the same units as the input parameters. Also, there is generally no relationship between simulated time and the time needed to run a simulation on the computer.

Historically, two principal approaches have been suggested for advancing the simulation clock: *next-event time advance* and *fixed-increment time advance*. Since the first approach is used by all major simulation software and by most people coding their model in a general-purpose language, and since the second is a special case of the first, we shall use the next-event time-advance approach for all discrete-event simulation models discussed in this book. A brief discussion of fixed-increment time advance is given in App. 1A (at the end of this chapter).

With the next-event time-advance approach, the simulation clock is initialized to zero and the times of occurrence of future events are determined. The simulation clock is then advanced to the time of occurrence of the *most imminent* (first) of these future events, at which point the state of the system is updated to account for the fact that an event has occurred, and our knowledge of the times of occurrence of future events is also updated. Then the simulation clock is advanced to the time of the (new) most imminent event, the state of the system is updated, and future event times are determined, etc. This process of advancing the simulation clock from one event time to another is continued until eventually some prespecified stopping condition is satisfied. Since all state changes occur only at event times for a discrete-event simulation model, periods of inactivity are skipped over by jumping the clock from event time to event time. (Fixed-increment time advance does not skip over these inactive periods, which can eat up a lot of computer time; see App. 1A.) It should be noted that the successive jumps of the simulation clock are generally variable (or unequal) in size.

EXAMPLE 1.2. We now illustrate in detail the next-event time-advance approach for the single-server queueing system of Example 1.1. We need the following notation:

t_i = time of arrival of the ith customer ($t_0 = 0$)
$A_i = t_i - t_{i-1}$ = interarrival time between $(i-1)$st and ith arrivals of customers
S_i = time that server actually spends serving ith customer (exclusive of customer's delay in queue)
D_i = delay in queue of ith customer
$c_i = t_i + D_i + S_i$ = time that ith customer completes service and departs
e_i = time of occurrence of ith event of any type (ith value the simulation clock takes on, excluding the value $e_0 = 0$)

Each of these defined quantities will generally be a random variable. Assume that the probability distributions of the interarrival times $A_1, A_2, \ldots$ and the service times $S_1, S_2, \ldots$ are known and have cumulative distribution functions (see Sec. 4.2) denoted by F_A and F_S, respectively. (In general, F_A and F_S would be determined by collecting data from the system of interest and then specifying distributions consistent with these data using the techniques of Chap. 6.) At time $e_0 = 0$ the status of the server is idle, and the time t_1 of the first arrival is determined by generating A_1 from F_A (techniques for generating random observations from a specified distribution are discussed in Chap. 8) and adding it to 0. The simulation clock is then advanced from e_0 to the time of the next (first) event, $e_1 = t_1$. (See Fig. 1.2, where the curved arrows represent advancing the simulation clock.) Since the customer arriving at time t_1 finds the server idle, she immediately enters service and has a delay in queue of $D_1 = 0$ and the status of the server is changed from idle to busy. The time, c_1, when the arriving customer will complete service is computed by generating S_1 from F_S and adding it to t_1. Finally, the time of the second arrival, t_2, is computed as $t_2 = t_1 + A_2$, where A_2 is generated from F_A. If $t_2 < c_1$, as depicted in Fig. 1.2, the simulation clock is advanced from e_1 to the time of the next event, $e_2 = t_2$. (If c_1 were less than t_2, the clock would be advanced from e_1 to c_1.) Since the customer arriving at time t_2 finds the server already busy, the number of customers in the queue is increased from 0 to 1 and the time of arrival of this customer is recorded; however, his service time S_2 is not generated at this time. Also, the time of the third arrival, t_3, is computed as $t_3 = t_2 + A_3$. If $c_1 < t_3$, as depicted in the figure, the simulation clock is advanced from e_2 to the time of the next event, $e_3 = c_1$, where the customer

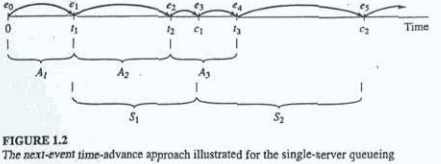


FIGURE 1.2
The next-event time-advance approach illustrated for the single-server queueing system.

completing service departs, the customer in the queue (i.e., the one who arrived at time t_2) begins service and his delay in queue and service-completion time are computed as $D_2 = c_1 - t_2$ and $c_2 = c_1 + S_2$ (S_2 is now generated from F_S), and the number of customers in the queue is decreased from 1 to 0. If $t_3 < c_1$, the simulation clock is advanced from e_3 to the time of the next event, $e_4 = t_3$, etc. The simulation might eventually be terminated when, say, the number of customers whose delays have been observed reaches some specified value.

1.3.2 Components and Organization of a Discrete-Event Simulation Model

Although simulation has been applied to a great diversity of real-world systems, discrete-event simulation models all share a number of common components and there is a logical organization for these components that promotes the programming, debugging, and future changing of a simulation model's computer program. In particular, the following components will be found in most discrete-event simulation models using the next-event time-advance approach programmed in a general-purpose language:

System state: The collection of state variables necessary to describe the system at a particular time
Simulation clock: A variable giving the current value of simulated time
Event list: A list containing the next time when each type of event will occur
Statistical counters: Variables used for storing statistical information about system performance
Initialization routine: A subprogram to initialize the simulation model at time 0
Timing routine: A subprogram that determines the next event from the event list and then advances the simulation clock to the time when that event is to occur
Event routine: A subprogram that updates the system state when a particular type of event occurs (there is one event routine for each event type)
Library routines: A set of subprograms used to generate random observations from probability distributions that were determined as part of the simulation model

Report generator: A subprogram that computes estimates (from the statistical counters) of the desired measures of performance and produces a report when the simulation ends

Main program: A subprogram that invokes the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately. The main program may also check for termination and invoke the report generator when the simulation is over.

The logical relationships (flow of control) among these components are shown in Fig. 1.3. The simulation begins at time 0 with the main program invoking the initialization routine, where the simulation clock is set to zero, the system state and the statistical counters are initialized, and the event list is initialized. After control has been returned to the main program, it invokes the timing routine to determine

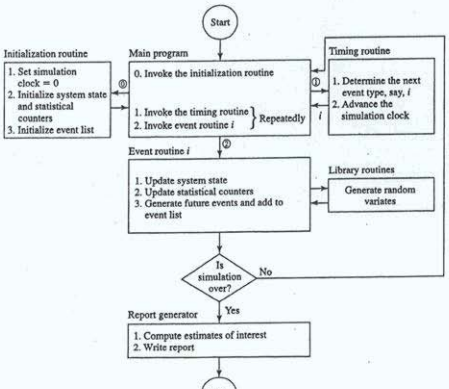


FIGURE 1.3
Flow of control for the next-event time-advance approach.

which type of event is most imminent. If an event of type i is the next to occur, the simulation clock is advanced to the time that event type i will occur and control is returned to the main program. Then the main program invokes event routine i, where typically three types of activities occur: (1) The system state is updated to account for the fact that an event of type i has occurred; (2) information about system performance is gathered by updating the statistical counters; and (3) the times of occurrence of future events are generated, and this information is added to the event list. Often it is necessary to generate random observations from probability distributions in order to determine these future event times; we will refer to such a generated observation as a *random variate*. After all processing has been completed, either in event routine i or in the main program, a check is typically made to determine (relative to some stopping condition) if the simulation should now be terminated. If it is time to terminate the simulation, the report generator is invoked from the main program to compute estimates (from the statistical counters) of the desired measures of performance and to produce a report. If it is not time for termination, control is passed back to the main program and the main program–timing routine–main program–event routine–termination check cycle is repeated until the stopping condition is eventually satisfied.

Before concluding this section, a few additional words about the system state may be in order. As mentioned in Sec. 1.2, a system is a well-defined collection of *entities*. Entities are characterized by data values called *attributes*, and these attributes are part of the system state for a discrete-event simulation model. Furthermore, entities with some common property are often grouped together in *lists* (or *files* or *sets*). For each entity there is a *record* in the list consisting of the entity's attributes, and the order in which the records are placed in the list depends on some specified rule. (See Chap. 2 for a discussion of efficient approaches for storing lists of records.) For the single-server queueing facility of Examples 1.1 and 1.2, the entities are the server and the customers in the facility. The server has the attribute "server status" (busy or idle), and the customers waiting in queue have the attribute "time of arrival." (The number of customers in the queue might also be considered an attribute of the server.) Furthermore, as we shall see in Sec. 1.4, these customers in queue will be grouped together in a list.

The organization and action of a discrete-event simulation program using the next-event time-advance mechanism as depicted above are fairly typical when coding such simulations in a general-purpose programming language such as FORTRAN or C; it is called the *event-scheduling approach* to simulation modeling, since the times of future events are explicitly coded into the model and are scheduled to occur in the simulated future. It should be mentioned here that there is an alternative approach to simulation modeling, called the *process approach*, that instead views the simulation in terms of the individual entities involved, and the code written describes the "experience" of a "typical" entity as it "flows" through the system; coding simulations modeled from the process point of view usually requires the use of special-purpose simulation software, as discussed in Chap. 3. Even when taking the process approach, however, the simulation is actually executed behind the scenes in the event-scheduling logic as described above.

1.4 SIMULATION OF A SINGLE-SERVER QUEUEING SYSTEM

This section shows in detail how to simulate a single-server queueing system such as a one-operator barbershop. Although this system seems very simple compared with those usually of real interest, how it is simulated is actually quite representative of the operation of simulations of great complexity.

In Sec. 1.4.1 we describe the system of interest and state our objectives more precisely. We explain intuitively how to simulate this system in Sec. 1.4.2 by showing a "snapshot" of the simulated system just after each event occurs. Section 1.4.3 describes the language-independent organization and logic of the FORTRAN and C codes given in Secs. 1.4.4 and 1.4.5. The simulation's results are discussed in Sec. 1.4.6, and Sec. 1.4.7 alters the stopping rule to another common way to end simulations. Finally, Sec. 1.4.8 briefly describes a technique for identifying and simplifying the event and variable structure of a simulation.

1.4.1 Problem Statement

Consider a single-server queueing system (see Fig. 1.4) for which the interarrival times $A_1, A_2, \ldots$ are *independent and identically distributed* (IID) random variables.

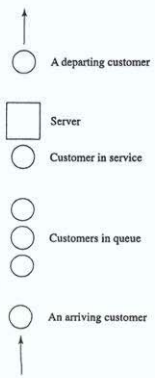


FIGURE 1.4
A single-server queueing sys...m.

("Identically distributed" means that the interarrival times have the same probability distribution.) A customer who arrives and finds the server idle enters service immediately, and the service times $S_1, S_2, \ldots$ of the successive customers are IID random variables that are independent of the interarrival times. A customer who arrives and finds the server busy joins the end of a single queue. Upon completing service for a customer, the server chooses a customer from the queue (if any) in a first-in, first-out (FIFO) manner. (For a discussion of other queue disciplines and queueing systems in general, see App. 1B.)

The simulation will begin in the "empty-and-idle" state; i.e., no customers are present and the server is idle. At time 0, we will begin waiting for the arrival of the first customer, which will occur after the first interarrival time, A_1, rather than at time 0 (which would be a possibly valid, but different, modeling assumption). We wish to simulate this system until a fixed number (n) of customers have completed their delays in queue; i.e., the simulation will stop when the nth customer enters service. Note that the *time* the simulation ends is thus a random variable, depending on the observed values for the interarrival and service-time random variables.

To measure the performance of this system, we will look at estimates of three quantities. First, we will estimate the expected average delay in queue of the n customers completing their delays during the simulation; we denote this quantity by $d(n)$. The word "expected" in the definition of $d(n)$ means this: On a given run of the simulation (or, for that matter, on a given run of the actual system the simulation model represents), the actual average delay observed of the n customers depends on the interarrival and service-time random variable observations that happen to have been obtained. On another run of the simulation (or on a different day for the real system) there would probably be arrivals at different times, and the service times required would also be different; this would give rise to a different value for the average of the n delays. Thus, the average delay on a given run of the simulation is properly regarded as a random variable itself. What we want to estimate, $d(n)$, is the *expected value* of this random variable. One interpretation of this is that $d(n)$ is the average of a large (actually, infinite) number of n-customer average delays. From a single run of the simulation resulting in customer delays $D_1, D_2, \ldots, D_n$, an obvious estimator of $d(n)$ is

$$\hat{d}(n) = \frac{\sum_{i=1}^{n} D_i}{n}$$

which is just the average of the n D_i's that were observed in the simulation [so that $\hat{d}(n)$ could also be denoted by $\bar{D}(n)$]. [Throughout this book, a hat ($\hat{}$) above a symbol denotes an estimator.] It is important to note that by "delay" we do not exclude the possibility that a customer could have a delay of zero in the case of an arrival finding the system empty and idle (with this model, we know for sure that $D_1 = 0$); delays with a value of 0 *are* counted in the average, since if many delays were zero this would represent a system providing very good service, and our output measure should reflect this. One reason for taking the average of the D_i's, as opposed to just lookin' them individually, is that they will not have the same distribution (e.g., $D_1 = 0$, but D_2 could be positive), and the average gives us a single composite

measure of all the customers' delays; in this sense, this is not the usual "average" taken in basic statistics, as the individual terms are not independent random observations from the same distribution. Note also that by itself, $\bar{d}(n)$ is an estimator based on a sample of size 1, since we are making only one complete simulation run. From elementary statistics, we know that a sample of size 1 is not worth much; we return to this issue in Chaps. 9 through 12.

While an estimate of $\bar{d}(n)$ gives information about system performance from the customers' point of view, the management of such a system may want different information; indeed, since most real simulations are quite complex and may be time-consuming to run, we usually collect many output measures of performance, describing different aspects of system behavior. One such measure for our simple model here is the expected average number of customers in the queue (but not being served), denoted by $q(n)$, where the n is necessary in the notation to indicate that this average is taken over the time period needed to observe the n delays defining our stopping rule. This is a different kind of "average" than the average delay in queue, because it is taken over (continuous) time, rather than over customers (being discrete). Thus, we need to define what is meant by this *time*-average number of customers in queue. To do this, let $Q(t)$ denote the number of customers in queue at time t, for any real number $t \geq 0$, and let $T(n)$ be the time required to observe our n delays in queue. Then for any time t between 0 and $T(n)$, $Q(t)$ is a nonnegative integer. Further, if we let p_i be the expected *proportion* (which will be between 0 and 1) of the time that $Q(t)$ is equal to i, then a reasonable definition of $q(n)$ would be

$$q(n) = \sum_{i=0}^{\infty} ip_i$$

Thus, $q(n)$ is a weighted average of the possible values i for the queue length $Q(t)$, with the weights being the expected proportion of time the queue spends at each of its possible lengths. To estimate $q(n)$ from a simulation, we simply replace the p_i's with estimates of them, and get

$$\hat{q}(n) = \sum_{i=0}^{\infty} i\hat{p}_i \tag{1.1}$$

where $\hat{p}_i$ is the *observed* (rather than expected) proportion of the time *during the simulation* that there were i customers in the queue. Computationally, however, it is easier to rewrite $\hat{q}(n)$ using some geometric considerations. If we let T_i be the *total* time during the simulation that the queue is of length i, then $T(n) = T_0 + T_1 + T_2 + \cdots$ and $\hat{p}_i = T_i/T(n)$, so that we can rewrite Eq. (1.1) above as

$$\hat{q}(n) = \frac{\sum_{i=0}^{\infty} iT_i}{T(n)} \tag{1.2}$$

Figure 1.5 illustrates a possible time path, or *realization*, of $Q(t)$ for this system in the case of $n = 6$; ignore the shading for now. Arrivals occur at times 0.4, 1.6, 2.1, 3.8, 4.0, 5.6, 5.8, and 7.2 Departures (service completions) occur at times 2.4, 3.1, 3.3, 4.9, and 8.6, and the simulation ends at time $T(6) = 8.6$. Remember in looking

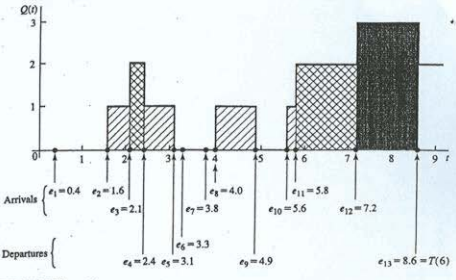


FIGURE 1.5
$Q(t)$, arrival times, and departure times for a realization of a single-server queueing system.

at Fig. 1.5 that $Q(t)$ does not count the customer in service (if any), so between times 0.4 and 1.6 there is one customer in the system being served, even though the queue is empty [$Q(t) = 0$]; the same is true between times 3.1 and 3.3, between times 3.8 and 4.0, and between times 4.9 and 5.6. Between times 3.3 and 3.8, however, the system is empty of customers and the server is idle, as is obviously the case between times 0 and 0.4. To compute $\hat{q}(n)$, we must first compute the T_i's, which can be read off Fig. 1.5 as the (sometimes separated) intervals over which $Q(t)$ is equal to 0, 1, 2, and so on:

$$T_0 = (1.6 - 0.0) + (4.0 - 3.1) + (5.6 - 4.9) = 3.2$$
$$T_1 = (2.1 - 1.6) + (3.1 - 2.4) + (4.9 - 4.0) + (5.8 - 5.6) = 2.3$$
$$T_2 = (2.4 - 2.1) + (7.2 - 5.8) = 1.7$$
$$T_3 = (8.6 - 7.2) = 1.4$$

($T_i = 0$ for $i \geq 4$, since the queue never grew to those lengths in this realization.) The numerator in Eq. (1.2) is thus

$$\sum_{i=0}^{\infty} iT_i = (0 \times 3.2) + (1 \times 2.3) + (2 \times 1.7) + (3 \times 1.4) = 9.9 \tag{1.3}$$

and so our estimate of the time-average number in queue from this particular simulation run is $\hat{q}(6) = 9.9/8.6 = 1.15$. Now, note that each of the nonzero terms on the right-hand side of Eq. (1.3) corresponds to one of the shaded areas in Fig. 1.5: 1×2.3 is the diagonally shaded area (in four pieces), 2×1.7 is the cross-hatched area (in two pieces), and 3×1.4 is the screened area (in a single piece). In other

words, the summation in the numerator of Eq. (1.2) is just the *area under the $Q(t)$ curve between the beginning and the end of the simulation*. Remembering that "area under a curve" is an integral, we can thus write

$$\sum_{i=0}^{\infty} iT_i = \int_0^{T(n)} Q(t)\,dt$$

and the estimator of $q(n)$ can then be expressed as

$$\hat{q}(n) = \frac{\int_0^{T(n)} Q(t)\,dt}{T(n)} \tag{1.4}$$

While Eqs. (1.4) and (1.2) are equivalent expressions for $\hat{q}(n)$, Eq. (1.4) is preferable since the integral in this equation can be accumulated as simple areas of rectangles as the simulation progresses through time. It is less convenient to carry out the computations to get the summation in Eq. (1.2) explicitly. Moreover, the appearance of Eq. (1.4) suggests a continuous average of $Q(t)$, since in a rough sense, an integral can be regarded as a continuous summation.

The third and final output measure of performance for this system is a measure of how busy the server is. The expected *utilization* of the server is the expected proportion of time during the simulation [from time 0 to time $T(n)$] that the server is busy (i.e., not idle), and is thus a number between 0 and 1; denote it by $u(n)$. From a single simulation, then, our estimate of $u(n)$ is $\hat{u}(n)$ = the *observed* proportion of time during the simulation that the server is busy. Now $\hat{u}(n)$ could be computed directly from the simulation by noting the times at which the server changes status (idle to busy or vice versa) and then doing the appropriate subtractions and division. However, it is easier to look at this quantity as a continuous-time average, similar to the average queue length, by defining the "busy function"

$$B(t) = \begin{cases} 1 & \text{if the server is busy at time } t \\ 0 & \text{if the server is idle at time } t \end{cases}$$

and so $\hat{u}(n)$ could be expressed as the proportion of time that $B(t)$ is equal to 1. Figure 1.6 plots $B(t)$ for the same simulation realization as used in Fig. 1.5 for $Q(t)$. In this case, we get

$$\hat{u}(n) = \frac{(3.3 - 0.4) + (8.6 - 3.8)}{8.6} = \frac{7.7}{8.6} = 0.90 \tag{1.5}$$

indicating that the server was busy about 90 percent of the time during this simulation. Again, however, the numerator in Eq. (1.5) can be viewed as the area under the $B(t)$ function over the course of the simulation, since the height of $B(t)$ is always either 0 or 1. Thus,

$$\hat{u}(n) = \frac{\int_0^{T(n)} B(t)\,dt}{T(n)} \tag{1.6}$$

and we see again that $\hat{u}(n)$ is the continuous average of the $B(t)$ function, corresponding to our notion of utilization. As was the case for $q(n)$, the reason for writing

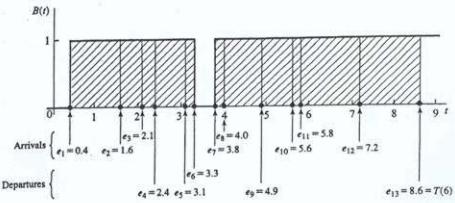


FIGURE 1.6
$B(t)$, arrival times, and departure times for a realization of a single-server queueing system (same realization as in Fig. 1.5).

$\hat{u}(n)$ in the integral form of Eq. (1.6) is that computationally, as the simulation progresses, the integral of $B(t)$ can easily be accumulated by adding up areas of rectangles. For many simulations involving "servers" of some sort, utilization statistics are quite informative in identifying bottlenecks (utilizations near 100 percent, coupled with heavy congestion measures for the queue leading in) or excess capacity (low utilizations); this is particularly true if the "servers" are expensive items such as robots in a manufacturing system or large mainframe computers in a data-processing operation.

To recap, the three measures of performance are the average delay in queue $\hat{d}(n)$, the time-average number of customers in queue $\hat{q}(n)$, and the proportion of time the server is busy $\hat{u}(n)$. The average delay in queue is an example of a *discrete-time statistic*, since it is defined relative to the collection of random variables $\{D_i\}$ that have a discrete "time" index, $i = 1, 2, \ldots$. The time-average number in queue and the proportion of time the server is busy are examples of *continuous-time statistics*, since they are defined on the collection of random variables $\{Q(t)\}$ and $\{B(t)\}$, respectively, each of which is indexed on the continuous time parameter $t \in [0, \infty)$. (The symbol $\in$ means "contained in." Thus, in this case, t can be any nonnegative real number.) Both discrete-time and continuous-time statistics are common in simulation, and they furthermore can be other than averages. For example, we might be interested in the *maximum* of all the delays in queue observed (a discrete-time statistic), or the *proportion* of time during the simulation that the queue contained at least five customers (a continuous-time statistic).

The events for this system are the arrival of a customer and the departure of a customer (after a service completion); the state variables necessary to estimate $\hat{d}(n)$, $q(n)$, and $u(n)$ are the status of the server (0 for idle and 1 for busy), the number of customers in the queue, the time of arrival of each customer currently in the queue (represented as a list), and the time of the last (i.e., most recent) event. The time of the last event, defined to be e_{i-1} if $e_{i-1} \le t < e_i$ (where t is the current time in the

simulation), is needed to compute the width of the rectangles for the area accumulations in the estimates of $q(n)$ and $u(n)$.

1.4.2 Intuitive Explanation

We begin our explanation of how to simulate a single-server queueing system by showing how its simulation model would be represented inside the computer at time $e_0 = 0$ and the times $e_1, e_2, \ldots, e_{13}$ at which the 13 successive events occur that are needed to observe the desired number, $n = 6$, of delays in queue. For expository convenience, we assume that the interarrival and service times of customers are

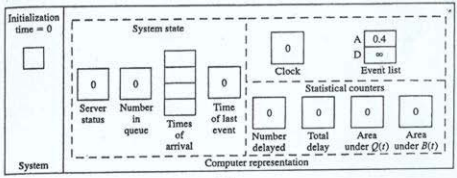
$$A_1 = 0.4, A_2 = 1.2, A_3 = 0.5, A_4 = 1.7, A_5 = 0.2,$$
$$A_6 = 1.6, A_7 = 0.2, A_8 = 1.4, A_9 = 1.9, \ldots$$

$$S_1 = 2.0, S_2 = 0.7, S_3 = 0.2, S_4 = 1.1, S_5 = 3.7, S_6 = 0.6, \ldots$$

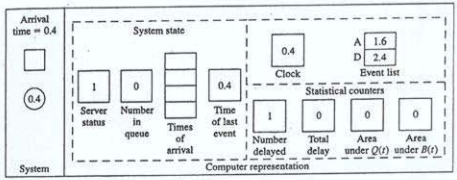
Thus, between time 0 and the time of the first arrival there is 0.4 time unit, between the arrivals of the first and second customers there are 1.2 time units, etc., and the service time required for the first customer is 2.0 time units, etc. Note that it is not necessary to declare what the time units are (minutes, hours, etc.), but only to be sure that all time quantities are expressed in the *same* units. In an actual simulation (see Secs. 1.4.4 and 1.4.5), the A_i's and the S_i's would be generated from their corresponding probability distributions, as needed, during the course of the simulation. The numerical values for the A_i's and the S_i's given above have been artificially chosen so as to generate the same simulation realization as depicted in Figs. 1.5 and 1.6 illustrating the $Q(t)$ and $B(t)$ processes.

Figure 1.7 gives a snapshot of the system itself and of a computer representation of the system at each of the times $e_0 = 0$, $e_1 = 0.4, \ldots, e_{13} = 8.6$. In the "system" pictures, the square represents the server, and circles represent customers; the numbers inside the customer circles are the times of their arrivals. In the "computer representation" pictures, the values of the variables shown are *after* all processing has been completed at that event. Our discussion will focus on how the computer representation changes at the event times.

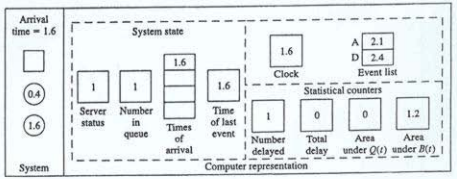
$t = 0$: *Initialization.* The simulation begins with the main program invoking the initialization routine. Our modeling assumption was that initially the system is empty of customers and the server is idle, as depicted in the "system" picture of Fig. 1.7*a*. The model state variables are initialized to represent this: Server status is 0 [we use 0 to represent an idle server and 1 to represent a busy server, similar to the definition of the $B(t)$ function], and the number of customers in the queue is 0. There is a one-dimensional array to store the times of arrival of customers *currently in the queue*; this array is initially empty, and as the simulation progresses, its length will grow and shrink. The time of the last (most recent) event is initialized to 0, so that at the time of the first event (when it is used), it will have its correct value. The simulation clock is set to 0, and the *event list*, giving the times of the next



(*a*)

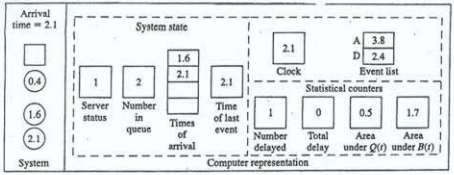


(*b*)

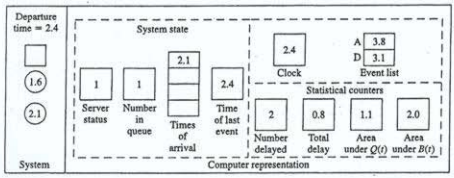


(*c*)

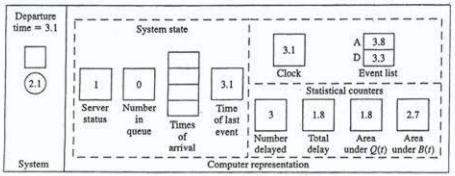
FIGURE 1.7
Snapshots of the system and of its computer representation at time 0 and at each of the 13 succeeding event times.



(*d*)

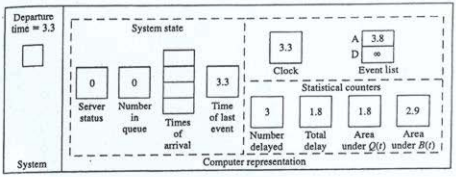


(*e*)

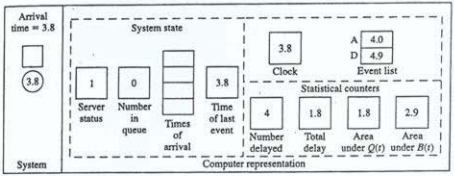


(*f*)

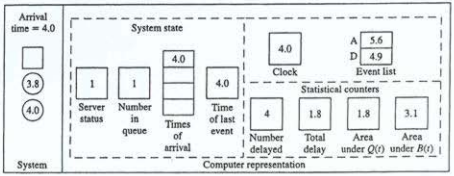
FIGURE 1.7
(*continued*)



(*g*)

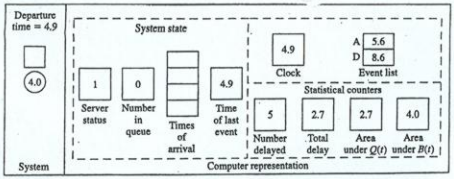


(*h*)

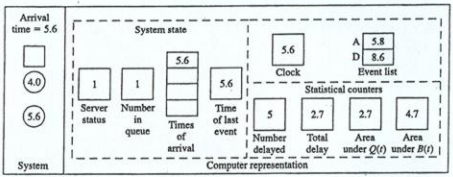


(*i*)

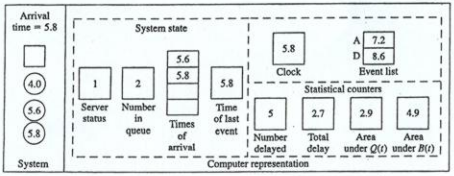
FIGURE 1.7
(*continued*)



(*j*)

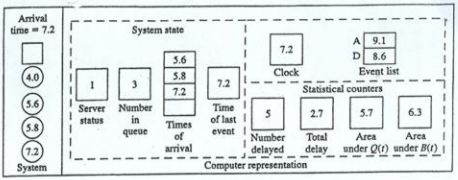


(*k*)

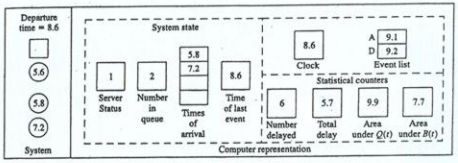


(*l*)

FIGURE 1.7
(*continued*)



(*m*)



(*n*)

FIGURE 1.7
(*continued*)

occurrence of each of the event types, is initialized as follows. The time of the first arrival is $0 + A_1 = 0.4$, and is denoted by "A" next to the event list. Since there is no customer in service, it does not even make sense to talk about the time of the next departure ("D" by the event list), and we know that the first event will be the initial customer arrival at time 0.4. However, the simulation progresses in general by looking at the event list and picking the smallest value from it to determine what the next event will be, so by scheduling the next departure to occur at time ∞ (or a very large number in a computer program), we effectively eliminate the departure event from consideration and force the next event to be an arrival. (This is sometimes called *poisoning* the departure event.) Finally, the four statistical counters are initialized to 0. When all initialization is done, control is returned to the main program, which then calls the timing routine to determine the next event. Since $0.4 < \infty$, the next event will be an arrival at time 0.4, and the timing routine advances the clock to this

time, then passes control back to the main program with the information that the next event is to be an arrival.

$t = 0.4$: *Arrival of customer 1.* At time 0.4, the main program passes control to the arrival routine to process the arrival of the first customer. Figure 1.7*b* shows the system and its computer representation *after* all changes have been made to process this arrival. Since this customer arrived to find the server idle (status equal to 0), he begins service immediately and has a delay in queue of $D_1 = 0$ (which *does* count as a delay). The server status is set to 1 to represent that the server is now busy, but the queue itself is still empty. The clock has been advanced to the current time, 0.4, and the event list is updated to reflect this customer's arrival: The next arrival will be $A_2 = 1.2$ time units from now, at time $0.4 + 1.2 = 1.6$, and the next departure (the service completion of the customer now arriving) will be $S_1 = 2.0$ time units from now, at time $0.4 + 2.0 = 2.4$. The number delayed is incremented to 1 (when this reaches $n = 6$, the simulation will end), and $D_1 = 0$ is added into the total delay (still at zero). The area under $Q(t)$ is updated by adding in the product of the *previous* value (i.e., the level it had between the last event and now) of $Q(t)$ (0 in this case) times the width of the interval of time from the last event to now, $t -$ (time of last event) $= 0.4 - 0$ in this case. Note that the time of the last event used here is its *old* value (0), before it is updated to its new value (0.4) in this event routine. Similarly, the area under $B(t)$ is updated by adding in the product of its previous value (0) times the width of the interval of time since the last event. [Look back at Figs. 1.5 and 1.6 to trace the accumulation of the areas under $Q(t)$ and $B(t)$.] Finally, the time of the last event is brought up to the current time, 0.4, and control is passed back to the main program. It invokes the timing routine, which scans the event list for the smallest value, and determines that the next event will be another arrival at time 1.6; it updates the clock to this value and passes control back to the main program with the information that the next event is an arrival.

$t = 1.6$: *Arrival of customer 2.* At this time we again enter the arrival routine, and Fig. 1.7*c* shows the system and its computer representation after all changes have been made to process this event. Since this customer arrives to find the server busy (status equal to 1 upon her arrival), she must queue up in the first location in the queue, her time of arrival is stored in the first location in the array, and the number-in-queue variable rises to 1. The time of the next arrival in the event list is updated to $A_3 = 0.5$ time unit from now, $1.6 + 0.5 = 2.1$; the time of the next departure is not changed, since its value of 2.4 is the departure time of customer 1, who is still in service at this time. Since we are not observing the end of anyone's delay in queue, the number-delayed and total-delay variables are unchanged. The area under $Q(t)$ is increased by 0 [the previous value of $Q(t)$] times the time since the last event, $1.6 - 0.4 = 1.2$. The area under $B(t)$ is increased by 1 [the previous value of $B(t)$] times this same interval of time, 1.2. After updating the time of the last event to now, control is passed back to the main program and then to the timing routine, which determines that the next event will be an arrival at time 2.1.

$t = 2.1$: *Arrival of customer 3.* Once again the arrival routine is invoked, as depicted in Fig. 1.7*d*. The server stays busy, and the queue grows by one customer, whose time of arrival is stored in the queue array's second location. The next arrival is updated to $t + A_4 = 2.1 + 1.7 = 3.8$, and the next departure is still the same, as we are still waiting for the service completion of customer 1. The delay counters are unchanged, since this is not the end of anyone's delay in queue, and the two area accumulators are updated by adding in 1 [the previous values of both $Q(t)$ and $B(t)$] times the time since the last event, $2.1 - 1.6 = 0.5$. After bringing the time of the last event up to the present, we go back to the main program and invoke the timing routine, which looks at the event list to determine that the next event will be a departure at time 2.4, and updates the clock to that time.

$t = 2.4$: *Departure of customer 1.* Now the main program invokes the departure routine, and Fig. 1.7*e* shows the system and its representation after this occurs. The server will maintain its busy status, since customer 2 moves out of the first place in queue and into service. The queue shrinks by 1, and the time-of-arrival array is moved up one place, to represent that customer 3 is now first in line. Customer 2, now entering service, will require $S_2 = 0.7$ time unit, so the time of the next departure (that of customer 2) in the event list is updated to S_2 time units from now, or to time $2.4 + 0.7 = 3.1$; the time of the next arrival (that of customer 4) is unchanged, since this was scheduled earlier at the time of customer 3's arrival, and we are still waiting at this time for customer 4 to arrive. The delay statistics are updated, since at this time customer 2 is entering service and is completing her delay in queue. Here we make use of the time-of-arrival array, and compute the second delay as the current time minus the second customer's time of arrival, or $D_2 = 2.4 - 1.6 = 0.8$. (Note that the value of 1.6 was stored in the first location in the time-of-arrival array *before* it was changed, so this delay computation would have to be done before advancing the times of arrival in the array.) The area statistics are updated by adding in $2 \times (2.4 - 2.1)$ for $Q(t)$ [note that the previous value of $Q(t)$ was used], and $1 \times (2.4 - 2.1)$ for $B(t)$. The time of the last event is updated, we return to the main program, and the timing routine determines that the next event is a departure at time 3.1.

$t = 3.1$: *Departure of customer 2.* The changes at this departure are similar to those at the departure of customer 1 at time 2.4 just discussed. Note that we observe another delay in queue, and that after this event is processed the queue is again empty, but the server is still busy.

$t = 3.3$: *Departure of customer 3.* Again, the changes are similar to those in the above two departure events, with one important exception: Since

the queue is now empty, the server becomes idle and we must set the next departure time in the event list to ∞, since the system now looks the same as it did at time 0 and we want to force the next event to be the arrival of customer 4.

$t = 3.8$: *Arrival of customer 4.* Since this customer arrives to find the server idle, he has a delay of 0 (i.e., $D_4 = 0$) and goes right into service. Thus, the changes here are very similar to those at the arrival of the first customer at time $t = 0.4$.

The remaining six event times are depicted in Figs. 1.7*i* through 1.7*n*, and readers should work through these to be sure they understand why the variables and arrays are as they appear; it may be helpful to follow along in the plots of $Q(t)$ and $B(t)$ in Figs. 1.5 and 1.6. With the departure of customer 5 at time $t = 8.6$, customer 6 leaves the queue and enters service, at which time the number delayed reaches 6 (the specified value of n) and the simulation ends. At this point, the main program invokes the report generator to compute the final output measures [$\hat{d}(6) = 5.7/6 = 0.95$, $\hat{q}(6) = 9.9/8.6 = 1.15$, and $\hat{u}(6) = 7.7/8.6 = 0.90$] and write them out.

A few specific comments about the above example illustrating the logic of a simulation should be made:

- Perhaps the key element in the dynamics of a simulation is the interaction between the simulation clock and the event list. The event list is maintained, and the clock jumps to the next event, as determined by scanning the event list at the end of each event's processing for the smallest (i.e., next) event time. This is how the simulation progresses through time.
- While processing an event, no "simulated" time passes. However, even though time is standing still for the model, care must be taken to process updates of the state variables and statistical counters in the appropriate order. For example, it would be incorrect to update the number in queue before updating the area-under-$Q(t)$ counter, since the height of the rectangle to be used is the *previous* value of $Q(t)$ [before the effect of the current event on $Q(t)$ has been implemented]. Similarly, it would be incorrect to update the time of the last event before updating the area accumulators. Yet another type of error would result if the queue list were changed at a departure before the delay of the first customer in queue were computed, since his time of arrival to the system would be lost.
- It is sometimes easy to overlook contingencies that seem out of the ordinary but that nevertheless must be accommodated. For example, it would be easy to forget that a departing customer could leave behind an empty queue, necessitating that the server be idled and the departure event again be eliminated from consideration. Also, termination conditions are often more involved than they might seem at first sight; in the above example, the simulation stopped in what seems to be the "usual" way, after a departure of one customer, allowing another to enter service and contribute the last delay needed, but the simulation *could* actually have ended instead with an arrival event—how?
- In some simulations it can happen that two (or more) entries in the event list are tied for smallest, and a decision rule must be incorporated to break such *time ties* (this happens with the inventory simulation considered later in Sec. 1.5). The tie-breaking rule can affect the results of the simulation, so must be chosen in accordance with how the system is to be modeled. In many simulations, however, we can ignore the possibility of ties, since the use of continuous random variables may make their occurrence an event with probability 0. In the above model, for example, if the interarrival-time or service-time distribution is continuous, then a time tie in the event list is a probability-zero event (though it could still happen during the computer simulation due to finite accuracy in representation of real numbers).

The above exercise is intended to illustrate the changes and data structures involved in carrying out a discrete-event simulation from the event-scheduling point of view, and contains most of the important ideas needed for more complex simulations of this type. The interarrival and service times used could have been drawn from a random-number table of some sort, constructed to reflect the desired probability distributions; this would result in what might be called a *hand simulation*, which in principle could be carried out to any length. The tedium of doing this should now be clear, so we will next turn to the use of computers (which are not easily bored) to carry out the arithmetic and bookkeeping involved in longer or more complex simulations.

1.4.3 Program Organization and Logic

In this section we set up the necessary ingredients for the programs to simulate the single-server queueing system in FORTRAN (Sec. 1.4.4) and C (Sec. 1.4.5). The organization and logic described in this section apply for both languages, so the reader need only go through one of Sec. 1.4.4 or 1.4.5, according to language preference.

There are several reasons for choosing a general-purpose language such as FORTRAN or C, rather than more powerful high-level simulation software, for introducing computer simulation at this point:

- By learning to simulate in a general-purpose language, in which one must pay attention to every detail, there will be a greater understanding of how simulations actually operate, and thus less chance of conceptual errors if a switch is later made to high-level simulation software.
- Despite the fact that there is now very good and powerful simulation software available (see Chap. 3), it is sometimes necessary to write at least parts of complex simulations in a general-purpose language if the specific, detailed logic of complex systems is to be represented faithfully.
- General-purpose languages are widely available, and entire simulations are sometimes still written in this way.

It is not our purpose in this book to teach any particular simulation software in detail, although we survey several packages in Chap. 3. With the understanding promoted by our more general approach and by going through our simulations in this and the next chapter, the reader should find it easier to learn a specialized simulation software product.

The single-server queueing model that we will simulate in the following two sections differs in two respects from the model used in the previous section:

- The simulation will end when $n = 1000$ delays in queue have been completed, rather than $n = 6$, in order to collect more data (and maybe to impress the reader with the patience of computers, since we have just slugged it out by hand in the $n = 6$ case in the preceding section). It is important to note that this change in the stopping rule changes the model itself, in that the output measures are defined relative to the stopping rule; hence the presence of the "n" in the notation for the quantities $d(n)$, $q(n)$, and $u(n)$ being estimated.
- The interarrival and service times will now be modeled as independent random variables from exponential distributions with mean 1 minute for the interarrival times and mean 0.5 minute for the service times. The exponential distribution with mean β (any positive real number) is continuous, with probability density function

$$f(x) = \frac{1}{\beta} e^{-x/\beta} \qquad \text{for } x \geq 0$$

(See Chaps. 4 and 6 for more information on density functions in general, and on the exponential distribution in particular.) We make this change here since it is much more common to generate input quantities (which drive the simulation) such as interarrival and service times from specified distributions than to assume that they are "known" as we did in the preceding section. The choice of the exponential distribution with the above particular values of β is essentially arbitrary, and is made primarily because it is easy to generate exponential random variates on a computer. (Actually, the assumption of exponential interarrival times is often quite realistic; assuming exponential service times, however, is less plausible.) Chapter 6 addresses in detail the important issue of how one chooses distribution forms and parameters for modeling simulation input random variables.

The single-server queue with exponential interarrival and service times is commonly called the *M/M/1 queue*, as discussed in App. 1B.

To simulate this model, we need a way to generate random variates from an exponential distribution. The subprograms used by the FORTRAN and C codes both operate in the same way, which we will now develop. First, a *random-number generator* (discussed in detail in Chap. 7) is invoked to generate a variate U that is distributed (continuously) uniformly between 0 and 1; this distribution will henceforth be referred to as U(0, 1) and has probability density function

$$f(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

It is easy to show that the probability that a U(0, 1) random variable falls in any subinterval $[x, x + \Delta x]$ contained in the interval [0, 1] is (uniformly) Δx (see Sec. 6.2.2). The U(0, 1) distribution is fundamental to simulation modeling because, as we shall see in Chap. 8, a random variate from any distribution can be generated by first generating one or more U(0, 1) random variates and then performing some kind of transformation. After obtaining U, we shall take the natural logarithm of it, multiply the result by β, and finally change the sign, to return what we will show to be an exponential random variate with mean β, that is, $-\beta \ln U$.

To see why this algorithm works, recall that the (*cumulative*) *distribution function* of a random variable X is defined, for any real x, to be $F(x) = P(X \leq x)$ (Chap. 4 contains a review of basic probability theory). If X is exponential with mean β, then

$$\begin{aligned} F(x) &= \int_0^x \frac{1}{\beta} e^{-t/\beta}\, dt \\ &= 1 - e^{-x/\beta} \end{aligned}$$

for any real $x \geq 0$, since the probability density function of the exponential distribution at the argument $t \geq 0$ is $(1/\beta)e^{-t/\beta}$. To show that our method is correct, we can try to verify that the value it returns will be less than or equal to x (any nonnegative real number), with probability $F(x)$ given above:

$$\begin{aligned} P(-\beta \ln U \leq x) &= P\left(\ln U \geq -\frac{x}{\beta}\right) \\ &= P(U \geq e^{-x/\beta}) \\ &= P(e^{-x/\beta} \leq U \leq 1) \\ &= 1 - e^{-x/\beta} \end{aligned}$$

The first line in the above is obtained by dividing through by $-\beta$ (recall that $\beta > 0$, so $-\beta < 0$ and the inequality reverses), the second line is obtained by exponentiating both sides (the exponential function is monotone increasing, so the inequality is preserved), the third line is just rewriting, together with knowing that U is in [0, 1] anyway, and the last line follows since U is U(0, 1), and the interval $[e^{-x/\beta}, 1]$ is contained within the interval [0, 1]. Since the last line is $F(x)$ for the exponential distribution, we have verified that our algorithm is correct. Chapter 8 discusses how to generate random variates and processes in general.

In our programs, we will use a particular method for random-number generation to obtain the variate U described above, as expressed in the FORTRAN and C codes of Figs. 7.5 through 7.7 in App. 7A of Chap. 7. While most compilers do have some kind of built-in random-number generator, many of these are of extremely poor quality and should not be used; this issue is discussed fully in Chap. 7.

It is convenient (if not the most computationally efficient) to modularize the programs into several subprograms to clarify the logic and interactions, as discussed in general in Sec. 1.3.2. In addition to a main program, the simulation program includes routines for initialization, timing, report generation, and generating exponential random variates, as in Fig. 1.3. It also simplifies matters if we write a separate routine to update the continuous-time statistics, being the accumulated areas under the $Q(t)$ and $B(t)$ curves. The most important action, however, takes place in the routines for the events, which we number as follows:

Event description	Event type
Arrival of a customer to the system	1
Departure of a customer from the system after completing service	2

As the logic of these event routines is independent of the particular language to be used, we shall discuss it here. Figure 1.8 contains a flowchart for the arrival event. First, the time of the next arrival in the future is generated and placed in the event list. Then a check is made to determine whether the server is busy. If so, the number of customers in the queue is incremented by 1, and we ask whether the storage space allocated to hold the queue is already full (see the code in Sec. 1.4.4 or 1.4.5 for details). If the queue is already full, an error message is produced and the simulation is stopped; if there is still room in the queue, the arriving customer's time

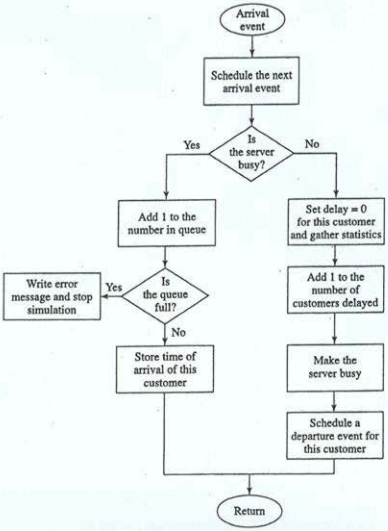


FIGURE 1.8
Flowchart for arrival routine, queueing model.

of arrival is put at the (new) end of the queue. (This queue-full check could be eliminated if using dynamic storage allocation in a programming language that supports this.) On the other hand, if the arriving customer finds the server idle, then this customer has a delay of 0, which *is* counted as a delay, and the number of customer delays completed is incremented by 1. The server must be made busy, and the time of departure from service of the arriving customer is scheduled into the event list.

The departure event's logic is depicted in the flowchart of Fig. 1.9. Recall that this routine is invoked when a service completion (and subsequent departure)

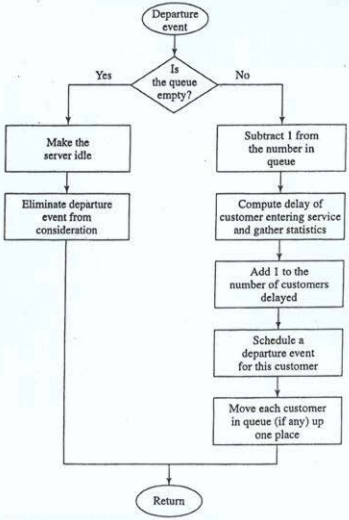


FIGURE 1.9
Flowchart for departure routine, queueing model.

occurs. If the departing customer leaves no other customers behind in queue, the *server is idled and the departure event is eliminated* from consideration, since the next event must be an arrival. On the other hand, if one or more customers are left behind by the departing customer, the first customer in queue will leave the queue and enter service, so the queue length is reduced by 1, and the delay in queue of this customer is computed and registered in the appropriate statistical counter. The number delayed is increased by 1, and a departure event for the customer now entering service is scheduled. Finally, the rest of the queue (if any) is advanced one place. Our implementation of the list for the queue will be very simple in this chapter, and is certainly not the most efficient; Chap. 2 discusses better ways of handling lists to model such things as queues.

In the next two sections we give examples of how the above setup can be used to write simulation programs in FORTRAN and C. Again, only one of these sections need be studied, depending on language familiarity or preference; the logic and organization are essentially identical, except for changes dictated by a particular language's features or shortcomings. The results (which were identical for both languages) are discussed in Sec. 1.4.6. These programs are neither the simplest nor most efficient possible, but were instead designed to illustrate how one might organize programs for more complex simulations.

1.4.4 FORTRAN Program

This section presents and describes a FORTRAN 77 program for the *M/M/*1 queue simulation. A general reference on the FORTRAN 77 language is Koffman and Friedman (1996). In this and all FORTRAN 77 programs in this book we have obeyed the ANSI standards so far as possible in an attempt to make the programs general and portable. The only exception to the ANSI standard for FORTRAN 77 is use of the INCLUDE statement, discussed later in this subsection, which can have slightly different syntax from what we show in our figures below. All code is available at http://www.mhhe.com/lawkelton.

We have also run the programs on a variety of different machines and compilers. The numerical results differed in some cases for a given model run with different compilers or on different machines, due to inaccuracies in floating-point operations. This can matter if, for example, at some point in the simulation two events are scheduled very close together in time, and roundoff error results in a different sequencing of the events' occurrences.

The subroutines and functions shown in Table 1.1 make up the FORTRAN program for this model. The table also shows the FORTRAN variables used (modeling variables include state variables, statistical counters, and variables used to facilitate coding).

The code for the main program is shown in Fig. 1.10, and begins with the INCLUDE statement to bring in the lines in the file mm1.dcl, which is shown in Fig. 1.11. The action the INCLUDE statement takes is to copy the file named between the quotes (in this case, mm1.dcl) into the main program in place of the INCLUDE statement. The file mm1.dcl contains "decla...tions" of the variables

TABLE 1.1
Subroutines, functions, and FORTRAN variables for the queueing model

Subprogram	Purpose
INIT	Initialization routine
TIMING	Timing routine
ARRIVE	Event routine to process type 1 events
DEPART	Event routine to process type 2 events
REPORT	Generates report when simulation ends
UPTAVG	Updates continuous-time area-accumulator statistics just before each event occurrence
EXPON(RMEAN)	Function to generate an exponential random variate with mean RMEAN
RAND(1)	Function to generate a uniform random variate between 0 and 1 (shown in Fig. 7.5)

Variable	Definition
Input parameters:	
MARRVT	Mean interarrival time (=1.0 here)
MSERVT	Mean service time (=0.5)
TOTCUS	Total number, n, of customer delays to be observed (=1000)
Modeling variables:	
ANIQ	Area under the number-in-queue function $[Q(t)]$ so far
AUTIL	Area under the server-status function $[B(t)]$ so far
BUSY	Mnemonic for server busy (=1)
DELAY	Delay in queue of a customer
IDLE	Mnemonic for server idle (=0)
MINTNE	Used by TIMING to determine which event is next
NEVNTS	Number of event types for this model, used by TIMING routine (=2 here)
NEXT	Event type (1 or 2 here) of the next event to occur (determined by TIMING routine)
NIQ	Number of customers currently in queue
NUMCUS	Number of customers who have completed their delays so far
QLIMIT	Number of storage locations for the queue TARRVL (=100)
RMEAN	Mean of the exponential random variate to be generated (used by EXPON)
SERVER	Server status (0 for idle, 1 for busy)
TARRVL(I)	Time of arrival of the customer now Ith in queue (dimensioned to have 100 places)
TIME	Simulation clock
TLEVNT	Time of the last (most recent) event
TNE(I)	Time of the next event of type I (I = 1, 2), part of event list
TOTDEL	Total of the delays completed so far
TSLE	Time since the last event (used by UPTAVG)
Output variables:	
AVGDEL	Average delay in queue $[\hat{d}(n)]$
AVGNIQ	Time-average number in queue $[\hat{q}(n)]$
UTIL	Server utilization $[\hat{u}(n)]$

```
      Main program for single-server queueing system.

      Bring in declarations file.

      INCLUDE 'mm1.dcl'

      Open input and output files.

      OPEN (5, FILE = 'mm1.in')
      OPEN (6, FILE = 'mm1.out')

      Specify the number of event types for the timing routine.

      NEVNTS = 2

      Set mnemonics for server's being busy and idle.

      BUSY = 1
      IDLE = 0

      Read input parameters.

      READ (5,*) MARRVT, MSERVT, TOTCUS

      Write report heading and input parameters.

      WRITE (6,2010) MARRVT, MSERVT, TOTCUS
 2010 FORMAT (' Single-server queueing system'//
     &        ' Mean interarrival time',F11.3,' minutes'//
     &        ' Mean service time',F16.3,' minutes'//
     &        ' Number of customers',I14//)

      Initialize the simulation.

      CALL INIT

      Determine the next event.

   10 CALL TIMING

      Update time-average statistical accumulators.

      CALL UPTAVG

      Call the appropriate event routine.

      GO TO (20, 30), NEXT
   20    CALL ARRIVE
         GO TO 40
   30    CALL DEPART

      If the simulation is over, call the report generator and end the
      simulation. If not, continue the simulation.

   40 IF (NUMCUS .LT. TOTCUS) GO TO 10
      CALL REPORT

      CLOSE (5)
      CLOSE (6)

      STOP
      END
```

FIGURE 1.10
FORTRAN code for the main program, queueing model.


```
      INTEGER QLIMIT
      PARAMETER (QLIMIT = 100)
      INTEGER BUSY,IDLE,NEVNTS,NEXT,NIQ,NUMCUS,SERVER,TOTCUS
      REAL ANIQ,AUTIL,MARRVT,MSERVT,TARRVL(QLIMIT),TIME,TLEVNT,TNE(2),
     &     TOTDEL
      REAL EXPON
      COMMON /MODEL/ ANIQ,AUTIL,BUSY,IDLE,MARRVT,MSERVT,NEVNTS,NEXT,NIQ,
     &               NUMCUS,SERVER,TARRVL,TIME,TLEVNT,TNE,TOTCUS,TOTDEL
```

FIGURE 1.11
FORTRAN code for the declarations file (mm1.dcl), queueing model.

and arrays to be INTEGER or REAL, the COMMON block MODEL, and the PARAMETER value QLIMIT = 100, our guess (which may have to be adjusted by trial and error) as to the longest the queue will ever get. (As mentioned earlier, using dynamic storage allocation in a language that supports this would eliminate the need for such a guess. While FORTRAN 77 does not support dynamic storage allocation, FORTRAN 90 does.) All of the statements in the file mm1.dcl must appear at the beginning of almost all subprograms in the simulation, and using the INCLUDE statement simply makes it easier to do this, and to make any necessary changes. The variables in the COMMON block MODEL are those we want to be global; i.e., variables in the block will be known and accessible to all subprograms that contain this COMMON statement. Variables not in COMMON will be *local* to the subprogram in which they appear. Also, we have adopted the convention of explicitly declaring the type (REAL or INTEGER) of all variables, arrays, and functions regardless of whether the first letter of the variable would default to its desired type according to the FORTRAN convention. Next, the input data file (called mm1.in) is opened and assigned to unit 5, and the file to contain the output (called mm1.out) is opened and assigned to unit 6. (We do not show the contents of the file mm1.in, since it is only a single line consisting of the numbers 1.0, 0.5, and 1000, separated by any number of blanks.) The number of event types for the simulation, NEVNTS, is initialized to 2 for this model, and the mnemonic constants BUSY and IDLE are set to use with the SERVER status variable, for code readability. The input parameters are then read in free format. After writing a report heading and echoing the input parameters (as a check that they were read correctly), the initialization routine INIT is called. The timing routine, TIMING, is then called to determine the event type, NEXT, of the next event to occur and to advance the simulation clock, TIME, to its time. Before processing this event, subroutine UPTAVG is called to update the areas under the $Q(t)$ and $B(t)$ curves, for the continuous-time statistics; UPTAVG also brings the time of the last event, TLEVNT, up to the present. By doing this at this time we automatically update these areas before processing each event. Then a "computed GO TO statement," based on NEXT, is used to pass control to the appropriate event routine. If NEXT = 1, event routine ARRIVE is called (at statement label 20) to process the arrival of a customer. If NEXT = 2, event routine DEPART (at statement label 30) is called to process the departure of a customer after completing service. After control is returned to the main program from ARRIVE or DEPART, a check is made (at statement label 40) to see whether the number of customers who have completed their delays, NUMCUS (which is

incremented by 1 after each customer completes his or her delay), is still (strictly) less than the number of customers whose delays we want to observe, TOTCUS. If so, TIMING is called to continue the simulation. If the specified number of delays has been observed, the report generator, REPORT, is called to compute and write estimates of the desired measures of performance. Finally, the input and output files are closed (a precautionary measure that is not required on many systems), and the simulation run is terminated.

Code for subroutine INIT is given in Fig. 1.12. Note that the same declarations file, mm1.dcl, is brought in here by the INCLUDE statement. Each statement in INIT corresponds to an element of the computer representation in Fig. 1.7*a*. Note that the time of the first arrival, TNE(1), is determined by adding an exponential random variate with mean MARRVT, namely, EXPON(MARRVT), to the simulation clock, TIME = 0. (We explicitly used TIME in this statement, although it has a value of 0, to show the general form of a statement to determine the time of a future event.) Since no customers are present at TIME = 0, the time of the next departure, TNE(2), is set to 1.0E + 30 (FORTRAN notation for 10^{30}), guaranteeing that the first event will be an arrival.

Subroutine TIMING is given in Fig. 1.13. The program compares TNE(1), TNE(2), . . . , TNE(NEVNTS) and sets NEXT equal to the event type whose time of occurrence is the smallest. (Note that NEVNTS is set in the main program.) In case of ties, the lowest-numbered event type is chosen. Then the simulation clock is advanced to the time of occurrence of the chosen event type, MINTNE. The program is complicated slightly by an error check for the event list's being empty,

```
      SUBROUTINE INIT
      INCLUDE 'mm1.dcl'

*     Initialize the simulation clock.

      TIME   = 0.0

*     Initialize the state variables.

      SERVER = IDLE
      NIQ    = 0
      TLEVNT = 0.0

*     Initialize the statistical counters.

      NUMCUS = 0
      TOTDEL = 0.0
      ANIQ   = 0.0
      AUTIL  = 0.0

*     Initialize event list.  Since no customers are present, the
*     departure (service completion) event is eliminated from
*     consideration.

      TNE(1) = TIME + EXPON(MARRVT)
      TNE(2) = 1.0E+30

      RETURN
      END
```

FIGURE 1.12
FORTRAN code for subroutine INIT, queueing model.


```
      SUBROUTINE TIMING
      INCLUDE 'mm1.dcl'
      INTEGER I
      REAL MINTNE

      MINTNE = 1.0E+29
      NEXT   = 0

*     Determine the event type of the next event to occur.

      DO 10 I = 1, NEVNTS
         IF (TNE(I) .LT. MINTNE) THEN
            MINTNE = TNE(I)
            NEXT   = I
         END IF
   10 CONTINUE

*     Check to see whether the event list is empty.

      IF (NEXT .EQ. 0) THEN

*        The event list is empty, so stop the simulation.

         WRITE (6,2010) TIME
 2010    FORMAT (' Event list empty at time',F10.3)
         STOP

      END IF

*     The event list is not empty, so advance the simulation clock.

      TIME = MINTNE

      RETURN
      END
```

FIGURE 1.13
FORTRAN code for subroutine TIMING, queueing model.

which we define to mean that all events are scheduled to occur at TIME = 10^{30}. If this is ever the case (as indicated by NEXT = 0), an error message is produced along with the current clock time (as a possible debugging aid), and the simulation is terminated.

The code for event routine ARRIVE is in Fig. 1.14, and follows the language-independent discussion as given in Sec. 1.4.3 and in the flowchart of Fig. 1.8. Note that TIME is the time of arrival of the customer who is just now arriving, and that the queue-overflow check (required since we cannot do dynamic storage allocation in FORTRAN 77) is made by asking whether NIQ is now greater than QLIMIT, the length for which TARRVL was dimensioned.

Event routine DEPART, whose code is shown in Fig. 1.15, is called from the main program when a service completion (and subsequent departure) occurs; the logic for it was discussed in Sec. 1.4.3, with a flowchart in Fig. 1.9. Note that if the statement TNE(2) = 1.0E + 30 just before the ELSE were omitted, the program would get into an infinite loop. (Why?) Advancing the rest of the queue (if any) one place by DO loop 10 ensures that the arrival time of the next customer entering service (after being delayed in queue) will always be stored in TARRVL(1). Note that if the queue were now empty (i.e., if the customer who just left the queue and entered service had been the only one in queue), then NIQ would be equal to 0, and

```
      SUBROUTINE ARRIVE
      INCLUDE 'mm1.dcl'
      REAL DELAY

*     Schedule next arrival.

      TNR(1) = TIME + EXPON(MARRVT)

*     Check to see whether server is busy.

      IF (SERVER .EQ. BUSY) THEN

*        Server is busy, so increment number of customers in queue.

         NIQ = NIQ + 1

*        Check to see whether an overflow condition exists.

         IF (NIQ .GT. QLIMIT) THEN

*           The queue has overflowed, so stop the simulation.

            WRITE (6,2010) TIME
 2010       FORMAT (' Overflow of the array TARRVL at time',F10.3)
            STOP

         END IF

*        There is still room in the queue, so store the time of arrival
*        of the arriving customer at the (new) end of TARRVL.

         TARRVL(NIQ) = TIME

      ELSE

*        Server is idle, so arriving customer has a delay of zero.  (The
*        following two statements are for program clarity and do not
*        affect the results of the simulation.)

         DELAY  = 0.0
         TOTDEL = TOTDEL + DELAY

*        Increment the number of customers delayed, and make server
*        busy.

         NUMCUS = NUMCUS + 1
         SERVER = BUSY

*        Schedule a departure (service completion).

         TNR(2) = TIME + EXPON(MSERVT)

      END IF

      RETURN
      END
```

FIGURE 1.14
FORTRAN code for subroutine ARRIVE, queueing model.

this DO loop would not be executed at all since the beginning value of the DO loop index, I, starts out at a value (1) that would already exceed its final value (NIQ = 0); this is a feature of FORTRAN 77. (Managing the queue in this simple way, by moving the arrival times up physically, is certainly inefficient; we return to this issue in Chap. 2.) A final comment about DEPART concerns the subtraction


```
      SUBROUTINE DEPART
      INCLUDE 'mm1.dcl'
      INTEGER I
      REAL DELAY

*     Check to see whether the queue is empty.

      IF (NIQ .EQ. 0) THEN

*        The queue is empty so make the server idle and eliminate the
*        departure (service completion) event from consideration.

         SERVER = IDLE
         TNR(2) = 1.0E+30

      ELSE

*        The queue is nonempty, so decrement the number of customers in
*        queue.

         NIQ = NIQ - 1

*        Compute the delay of the customer who is beginning service and
*        update the total delay accumulator.

         DELAY  = TIME - TARRVL(1)
         TOTDEL = TOTDEL + DELAY

*        Increment the number of customers delayed, and schedule
*        departure.

         NUMCUS = NUMCUS + 1
         TNR(2) = TIME + EXPON(MSERVT)

*        Move each customer in queue (if any) up one place.

         DO 10 I = 1, NIQ
   10    TARRVL(I) = TARRVL(I + 1)

      END IF

      RETURN
      END
```

FIGURE 1.15
FORTRAN code for subroutine DEPART, queueing model.

simulation is to run for a long period of (simulated) time, both TIME and TARRVL(1) would become very large numbers in comparison with the difference between them; thus, since they are both stored as floating-point (REAL) numbers with finite accuracy, there is potentially a serious loss of precision when doing this subtraction. For this reason, it may be necessary to make both TIME and the TARRVL array DOUBLE PRECISION if we want to run this simulation out for a long period of time.

The code for subroutine REPORT, called when the termination check in the main program determines that the simulation is over, is given in Fig. 1.16. The average delay, AVGDEL, is computed by dividing the total of the delays by the number of customers whose delays were observed, and the time-average number in queue, AVGNIQ, is obtained by dividing the area under $Q(t)$, now updated to the end of the simulation (since UPTAVG is called from the main program before processing either an

```
      SUBROUTINE REPORT
      INCLUDE 'mm1.dcl'
      REAL AVGDEL,AVGNIQ,UTIL

*     Compute and write estimates of desired measures of performance.

      AVGDEL = TOTDEL / NUMCUS
      AVGNIQ = ANIQ / TIME
      UTIL   = AUTIL / TIME
      WRITE (6,2010) AVGDEL, AVGNIQ, UTIL, TIME
 2010 FORMAT (//' Average delay in queue',F11.3,' minutes'//
     &         ' Average number in queue',F10.3//
     &         ' Server utilization',F15.3//
     &         ' Time simulation ended',F12.3,' minutes')

      RETURN
      END
```

FIGURE 1.16
FORTRAN code for subroutine REPORT, queueing model.

termination. The server utilization, UTIL, is computed by dividing the area under $B(t)$ by the final clock time, and all three measures are written out. We also write out the final clock value itself, to see how long it took to observe the 1000 delays.

Subroutine UPTAVG is shown in Fig. 1.17. This subroutine is called just before processing each event (of any type) and updates the areas under the two functions needed for the continuous-time statistics; this routine is separate for coding convenience only, and is *not* an event routine. The time since the last event, TSLE, is first computed, and the time of the last event, TLEVNT, is brought up to the current time in order to be ready for the next entry into UPTAVG. Then the area under the number-in-queue function is augmented by the area of the rectangle under $Q(t)$ during the interval since the previous event, which is of width TSLE and height NIQ; remember, UPTAVG is called *before* processing an event, and state variables such as NIQ still have their previous values. The area under $B(t)$ is then augmented by the

```
      SUBROUTINE UPTAVG
      INCLUDE 'mm1.dcl'
      REAL TSLE

*     Compute time since last event, and update last-event-time marker.

      TSLE   = TIME - TLEVNT
      TLEVNT = TIME

*     Update area under number-in-queue function.

      ANIQ   = ANIQ + NIQ * TSLE

*     Update area under server-busy indicator function.

      AUTIL  = AUTIL + SERVER * TSLE

      RETURN
      END
```

FIGURE 1.17
FORTRAN code for subroutine UPTAVG, queueing model.


```
      REAL FUNCTION EXPON(RMEAN)
      REAL RMEAN
      REAL RAND

*     Return an exponential random variate with mean RMEAN.

      EXPON = -RMEAN * LOG(RAND(1))

      RETURN
      END
```

FIGURE 1.18
FORTRAN code for function EXPON.

area of a rectangle of width TSLE and height SERVER; this is why it is convenient to define SERVER to be either 0 or 1. Note that this routine, like DEPART, contains a subtraction of two floating-point numbers (TIME − TLEVNT), both of which could become quite large relative to their difference if we were to run the simulation for a long time; in this case it may be necessary to declare both TIME and TLEVNT to be DOUBLE PRECISION variables.

The function EXPON, which generates an exponential random variate with mean β = RMEAN (passed into EXPON), is shown in Fig. 1.18, and follows the algorithm discussed in Sec. 1.4.3. The random-number generator RAND, used here with an INTEGER argument of 1, is discussed fully in Chap. 7, and is shown specifically in Fig. 7.5. The FORTRAN built-in function LOG returns the natural logarithm of its argument, and agrees in type with its argument.

The program described here must be combined with the random-number-generator code from Fig. 7.5. This could be done by separate compilations, followed by linking the object codes together in an installation-dependent way.

1.4.5 C Program

This section presents a C program for the *M/M/*1 queue simulation. We use the ANSI-standard version of the language, as defined by Kernighan and Ritchie (1988), and in particular use function prototyping. We have also taken advantage of C's facility to give variables and functions fairly long names, which should thus be self-explanatory (for instance, the current value of simulated time is in a variable called sim_time). As with our FORTRAN 77 programs, we have run all our C programs on several different computers and compilers to ensure good portability, and for the same reasons given at the beginning of Sec. 1.4.4, numerical results can differ across computers and compilers. The C math library must be linked, which might require setting an option, depending on the compiler (on UNIX systems this option is often -lm in the compilation statement). All code is available at http://www.mhhe.com/lawkelton.

The external definitions are given in Fig. 1.19. The header file lcgrand.h (listed in Fig. 7.7) is included to declare the functions for the random-number generator. The symbolic constant Q_LIMIT is set to 100, our guess (which may have to be adjusted by trial and error) as to the longest the queue will ever get. (As mentioned

```
/* External definitions for single-server queueing system. */

#include <stdio.h>
#include <math.h>
#include "lcgrand.h"  /* Header file for random-number generator. */

#define Q_LIMIT 100  /* Limit on queue length. */
#define BUSY      1  /* Mnemonics for server's being busy */
#define IDLE      0  /* and idle. */

int   next_event_type, num_custs_delayed, num_delays_required, num_events,
      num_in_q, server_status;
float area_num_in_q, area_server_status, mean_interarrival, mean_service,
      sim_time, time_arrival[Q_LIMIT + 1], time_last_event, time_next_event[3],
      total_of_delays;
FILE  *infile, *outfile;

void  initialize(void);
void  timing(void);
void  arrive(void);
void  depart(void);
void  report(void);
void  update_time_avg_stats(void);
float expon(float mean);
```

FIGURE 1.19
C code for the external definitions, queueing model.

earlier, this guess could be eliminated if we were using dynamic storage allocation; while C supports this, we have not used it in our examples in order to make them comparable to the corresponding FORTRAN 77 examples.) The symbolic constants BUSY and IDLE are defined to be used with the server_status variable, for code readability. File pointers *infile and *outfile are defined to allow us to open the input and output files from within the code, rather than at the operating-system level. Note also that the event list, as we have discussed it so far, will be implemented in an array called time_next_event, whose 0th entry will be ignored in order to make the index agree with the event type.

The code for the main function is shown in Fig. 1.20. The input and output files are opened, and the number of event types for the simulation is initialized to 2 for this model. The input parameters then are read in from the file mm1.in, which contains a single line with the numbers 1.0, 0.5, and 1000, separated by blanks. After writing a report heading and echoing the input parameters (as a check that they were read correctly), the initialization function is invoked. The "while" loop then executes the simulation as long as more customer delays are needed to fulfill the 1000-delay stopping rule. Inside the "while" loop, the timing function is first invoked to determine the type of the next event to occur and to advance the simulation clock to its time. Before processing this event, the function to update the areas under the $Q(t)$ and $B(t)$ curves is invoked; by doing this at this time we automatically update these areas before processing each event. Then a switch statement, based on next_event_type (=1 for an arrival and 2 for a departure), passes control to the appropriate event function. After the "while" loop is done, the report function is invoked, the input and output files are closed, and the simulation ends.


```
main()  /* Main function. */
{
    /* Open input and output files. */

    infile  = fopen("mm1.in",  "r");
    outfile = fopen("mm1.out", "w");

    /* Specify the number of events for the timing function. */

    num_events = 2;

    /* Read input parameters. */

    fscanf(infile, "%f %f %d", &mean_interarrival, &mean_service,
           &num_delays_required);

    /* Write report heading and input parameters. */

    fprintf(outfile, "Single-server queueing system\n\n");
    fprintf(outfile, "Mean interarrival time%11.3f minutes\n\n",
            mean_interarrival);
    fprintf(outfile, "Mean service time%16.3f minutes\n\n", mean_service);
    fprintf(outfile, "Number of customers%14d\n\n", num_delays_required);

    /* Initialize the simulation. */

    initialize();

    /* Run the simulation while more delays are still needed. */

    while (num_custs_delayed < num_delays_required) {

        /* Determine the next event. */

        timing();

        /* Update time-average statistical accumulators. */

        update_time_avg_stats();

        /* Invoke the appropriate event function. */

        switch (next_event_type) {
            case 1:
                arrive();
                break;
            case 2:
                depart();
                break;
        }
    }

    /* Invoke the report generator and end the simulation. */

    report();

    fclose(infile);
    fclose(outfile);

    return 0;
}
```

FIGURE 1.20
C code for the main function, queueing model.

```
void initialize(void)  /* Initialization function. */
{
    /* Initialize the simulation clock. */

    sim_time = 0.0;

    /* Initialize the state variables. */

    server_status   = IDLE;
    num_in_q        = 0;
    time_last_event = 0.0;

    /* Initialize the statistical counters. */

    num_custs_delayed  = 0;
    total_of_delays    = 0.0;
    area_num_in_q      = 0.0;
    area_server_status = 0.0;

    /* Initialize event list.  Since no customers are present, the departure
       (service completion) event is eliminated from consideration. */

    time_next_event[1] = sim_time + expon(mean_interarrival);
    time_next_event[2] = 1.0e+30;
}
```

FIGURE 1.21
C code for function initialize, queueing model.

Code for the initialization function is given in Fig. 1.21. Each statement here corresponds to an element of the computer representation in Fig. 1.7*a*. Note that the time of the first arrival, time_next_event[1], is determined by adding an exponential random variate with mean mean_interarrival, namely, expon(mean_interarrival), to the simulation clock, sim_time = 0. (We explicitly used "sim_time" in this statement, although it has a value of 0, to show the general form of a statement to determine the time of a future event.) Since no customers are present at time sim_time = 0, the time of the next departure, time_next_event[2], is set to 1.0e + 30 (C notation for 10^{30}), guaranteeing that the first event will be an arrival.

The timing function, which is given in Fig. 1.22, is used to compare time_next_event[1], time_next_event[2], . . . , time_next_event[num_events] (recall that num_events was set in the main function) and to set next_event_type equal to the event type whose time of occurrence is the smallest. In case of ties, the lowest-numbered event type is chosen. Then the simulation clock is advanced to the time of occurrence of the chosen event type, min_time_next_event. The program is complicated slightly by an error check for the event list's being empty, which we define to mean that all events are scheduled to occur at time = 10^{30}. If this is ever the case (as indicated by next_event_type = 0), an error message is produced along with the current clock time (as a possible debugging aid), and the simulation is terminated.

The code for event function arrive is in Fig. 1.23, and follows the language-independent discussion as given in Sec. 1.4.3 and in the flowchart of Fig. 1.8. Note that "sim_time" is the time of arrival of the customer who is just now arriving, and that the queue-overflow check is made by asking whether num_in_q is now greater than Q_LIMIT, the length for which the array time_arriv. was dimensioned.


```
void timing(void)  /* Timing function. */
{
    int   i;
    float min_time_next_event = 1.0e+29;

    next_event_type = 0;

    /* Determine the event type of the next event to occur. */

    for (i = 1; i <= num_events; ++i)
        if (time_next_event[i] < min_time_next_event) {
            min_time_next_event = time_next_event[i];
            next_event_type     = i;
        }

    /* Check to see whether the event list is empty. */

    if (next_event_type == 0) {

        /* The event list is empty, so stop the simulation. */

        fprintf(outfile, "\nEvent list empty at time %f", sim_time);
        exit(1);
    }

    /* The event list is not empty, so advance the simulation clock. */

    sim_time = min_time_next_event;
}
```

FIGURE 1.22
C code for function timing, queueing model.

Event function depart, whose code is shown in Fig. 1.24, is invoked from the main program when a service completion (and subsequent departure) occurs; the logic for it was discussed in Sec. 1.4.3, with the flowchart in Fig. 1.9. Note that if the statement "time_next_event[2] = 1.0e + 30;" just before the "else" were omitted, the program would get into an infinite loop. (Why?) Advancing the rest of the queue (if any) one place by the "for" loop near the end of the function ensures that the arrival time of the next customer entering service (after being delayed in queue) will always be stored in time_arrival[1]. Note that if the queue were now empty (i.e., the customer who just left the queue and entered service had been the only one in queue), then num_in_q would be equal to 0, and this loop would not be executed at all since the beginning value of the loop index, i, starts out at a value (1) that would already exceed its final value (num_in_q = 0). (Managing the queue in this simple way is certainly inefficient, and could be improved by using pointers; we return to this issue in Chap. 2.) A final comment about depart concerns the subtraction of time_arrival[1] from the clock value, sim_time, to obtain the delay in queue. If the simulation is to run for a long period of (simulated) time, both sim_time and time_arrival[1] would become very large numbers in comparison with the difference between them; thus, since they are both stored as floating-point (float) numbers with finite accuracy, there is potentially a serious loss of precision when doing this subtraction. For this reason, it may be necessary to make both sim_time and the time_arrival ar- of type double if we are to run this simulation out for a long period of time.

```
void arrive(void)  /* Arrival event function. */
{
    float delay;

    /* Schedule next arrival. */

    time_next_event[1] = sim_time + expon(mean_interarrival);

    /* Check to see whether server is busy. */

    if (server_status == BUSY) {

        /* Server is busy, so increment number of customers in queue. */

        ++num_in_q;

        /* Check to see whether an overflow condition exists. */

        if (num_in_q > Q_LIMIT) {

            /* The queue has overflowed, so stop the simulation. */

            fprintf(outfile, "\nOverflow of the array time_arrival at");
            fprintf(outfile, " time %f", sim_time);
            exit(2);
        }

        /* There is still room in the queue, so store the time of arrival of the
           arriving customer at the (new) end of time_arrival. */

        time_arrival[num_in_q] = sim_time;
    }

    else {

        /* Server is idle, so arriving customer has a delay of zero.  (The
           following two statements are for program clarity and do not affect
           the results of the simulation.) */

        delay            = 0.0;
        total_of_delays += delay;

        /* Increment the number of customers delayed, and make server busy. */

        ++num_custs_delayed;
        server_status = BUSY;

        /* Schedule a departure (service completion). */

        time_next_event[2] = sim_time + expon(mean_service);
    }
}
```

FIGURE 1.23
C code for function arrive, queueing model.

The code for the report function, invoked when the "while" loop in the main program is over, is given in Fig. 1.25. The average delay is computed by dividing the total of the delays by the number of customers whose delays were observed, and the time-average number in queue is obtained by dividing the area under $Q(t)$, now updated to the end of the simulation (since the function to update the areas is called from the main program before processing either an arrival or departure, one of which will end the simulation), by the clock value at termination. The server


```
void depart(void)  /* Departure event function. */
{
    int   i;
    float delay;

    /* Check to see whether the queue is empty. */

    if (num_in_q == 0) {

        /* The queue is empty so make the server idle and eliminate the
           departure (service completion) event from consideration. */

        server_status      = IDLE;
        time_next_event[2] = 1.0e+30;
    }

    else {

        /* The queue is nonempty, so decrement the number of customers in
           queue. */

        --num_in_q;

        /* Compute the delay of the customer who is beginning service and update
           the total delay accumulator. */

        delay            = sim_time - time_arrival[1];
        total_of_delays += delay;

        /* Increment the number of customers delayed, and schedule departure. */

        ++num_custs_delayed;
        time_next_event[2] = sim_time + expon(mean_service);

        /* Move each customer in queue (if any) up one place. */

        for (i = 1; i <= num_in_q; ++i)
            time_arrival[i] = time_arrival[i + 1];
    }
}
```

FIGURE 1.24
C code for function depart, queueing model.

```
void report(void)  /* Report generator function. */
{
    /* Compute and write estimates of desired measures of performance. */

    fprintf(outfile, "\n\nAverage delay in queue%11.3f minutes\n\n",
            total_of_delays / num_custs_delayed);
    fprintf(outfile, "Average number in queue%10.3f\n\n",
            area_num_in_q / sim_time);
    fprintf(outfile, "Server utilization%15.3f\n\n",
            area_server_status / sim_time);
    fprintf(outfile, "Time simulation ended%12.3f minutes", sim_time);
}
```

FIGURE 1.25
C code for function report, queueing model.

```
void update_time_avg_stats(void)  /* Update area accumulators for time-average
                                     statistics. */
{
    float time_since_last_event;

    /* Compute time since last event, and update last-event-time marker. */

    time_since_last_event = sim_time - time_last_event;
    time_last_event       = sim_time;

    /* Update area under number-in-queue function. */

    area_num_in_q      += num_in_q * time_since_last_event;

    /* Update area under server-busy indicator function. */

    area_server_status += server_status * time_since_last_event;
}
```

FIGURE 1.26
C code for function update_time_avg_stats, queueing model.

utilization is computed by dividing the area under $B(t)$ by the final clock time, and all three measures are written out directly. We also write out the final clock value itself, to see how long it took to observe the 1000 delays.

Function update_time_avg_stats is shown in Fig. 1.26. This function is invoked just before processing each event (of any type) and updates the areas under the two functions needed for the continuous-time statistics; this routine is separate for coding convenience only, and is *not* an event routine. The time since the last event is first computed, and then the time of the last event is brought up to the current time in order to be ready for the next entry into this function. Then the area under the number-in-queue function is augmented by the area of the rectangle under $Q(t)$ during the interval since the previous event, which is of width time_since_last_event and of height num_in_q; remember, this function is invoked *before* processing an event, and state variables such as num_in_q still have their previous values. The area under $B(t)$ is then augmented by the area of a rectangle of width time_since_last_event and height server_status; this is why it is convenient to define server_status to be either 0 or 1. Note that this function, like depart, contains a subtraction of two floating-point numbers (sim_time − time_last_event), both of which could become quite large relative to their difference if we were to run the simulation for a long time; in this case it might be necessary to declare both sim_time and time_last_event to be of type double.

The function expon, which generates an exponential random variate with mean β = mean (passed into expon), is shown in Fig. 1.27, and follows the algorithm

```
float expon(float mean)  /* Exponential variate generation function. */
{
    /* Return an exponential random variate with mean "mean". */

    return -mean * log(lcgrand(1));
}
```

FIGURE 1.27
C code for function expon.

discussed in Sec. 1.4.3. The random-number generator lcgrand, used here with an int argument of 1, is discussed fully in Chap. 7, and is shown specifically in Fig. 7.6. The C predefined function log returns the natural logarithm of its argument.

The program described here must be combined with the random-number-generator code from Fig. 7.6. This could be done by separate compilations, followed by linking the object codes together in an installation-dependent way.

1.4.6 Simulation Output and Discussion

The output (in a file named mm1.out) is shown in Fig. 1.28; since the same method for random-number generation was used for the programs in both languages, they produced identical results. In this run, the average delay in queue was 0.430 minute, there was an average of 0.418 customer in the queue, and the server was busy 46 percent of the time. It took 1027.915 simulated minutes to run the simulation to the completion of 1000 delays, which seems reasonable since the expected time between customer arrivals was 1 minute. (It is not a coincidence that the average delay, average number in queue, and utilization are all so close together for this model; see App. 1B.)

Note that these particular numbers in the output were determined, at root, by the numbers the random-number generator happened to come up with this time. If a different random-number generator were used, or if this one were used in another way (with another "seed" or "stream," as discussed in Chap. 7), then different numbers would have been produced in the output. Thus, these numbers are not to be regarded as "The Answers," but rather as estimates (and perhaps poor ones) of the expected quantities we want to know about, $d(n)$, $q(n)$, and $u(n)$; the statistical analysis of simulation output data is discussed in Chaps. 9 through 12. Also, the results are functions of the input parameters, in this case the mean interarrival and service times, and the $n = 1000$ stopping rule; they are also affected by the way we initialized the simulation (empty and idle).

In some simulation studies, we might want to estimate *steady-state* characteristics of the model (see Chap. 9), i.e., characteristics of a model after the simulation has been running a very long (in theory, an infinite amount of) time. For the simple

```
Single-server queueing system

Mean interarrival time          1.000 minutes

Mean service time               0.500 minutes

Number of customers              1000


Average delay in queue          0.430 minutes

Average number in queue         0.418

Server utilization              0.460

Time simulation ended       1027.915 minutes
```

FIGURE 1.28
Output report, queueing model.

*M/M/*1 queue we have been considering, it is possible to compute *analytically* the steady-state average delay in queue, the steady-state time-average number in queue, and the steady-state server utilization, all of these measures of performance being 0.5 [see, e.g., Ross (1997, pp. 419–420)]. Thus, if we wanted to determine these steady-state measures, our estimates based on the stopping rule $n = 1000$ delays were not too far off, at least in absolute terms. However, we were somewhat lucky, since $n = 1000$ was chosen arbitrarily! In practice, the choice of a stopping rule that will give good estimates of steady-state measures is quite difficult. To illustrate this point, suppose for the *M/M/*1 queue that the arrival rate of customers were increased from 1 per minute to 1.98 per minute (the mean interarrival time is now 0.505 minute), that the mean service time is unchanged, and that we wish to estimate the steady-state measures from a run of length $n = 1000$ delays, as before. We performed this simulation run and got values for the average delay, average number in queue, and server utilization of 17.404 minutes, 34.831, and 0.997, respectively. Since the true steady-state values of these measures are 49.5 minutes, 98.01, and 0.99 (respectively), it is clear that the stopping rule cannot be chosen arbitrarily. We discuss how to specify the run length for a steady-state simulation in Chap. 9.

The reader may have wondered why we did not estimate the expected average waiting time in the system of a customer, $w(n)$, rather than the expected average delay in queue, $d(n)$, where the waiting time of a customer is defined as the time interval from the instant the customer arrives to the instant the customer completes service and departs. There were two reasons. First, for many queueing systems we believe that the customer's delay in queue while waiting for other customers to be served is the most troublesome part of the customer's wait in the system. Moreover, if the queue represents part of a manufacturing system where the "customers" are actually parts waiting for service at a machine (the "server"), then the delay in queue represents a loss, whereas the time spent in service is "necessary." Our second reason for focusing on the delay in queue is one of statistical efficiency. The usual estimator of $w(n)$ would be

$$\hat{w}(n) = \frac{\sum_{i=1}^{n} W_i}{n} = \frac{\sum_{i=1}^{n} D_i}{n} + \frac{\sum_{i=1}^{n} S_i}{n} = \hat{d}(n) + \bar{S}(n) \qquad (1.7)$$

where $W_i = D_i + S_i$ is the waiting time in the system of the ith customer and $\bar{S}(n)$ is the average of the n customers' service times. Since the service-time distribution would have to be known to perform a simulation in the first place, the expected or mean service time, $E(S)$, would also be known and an alternative estimator of $w(n)$ is

$$\tilde{w}(n) = \hat{d}(n) + E(S)$$

[Note that $\bar{S}(n)$ is an unbiased estimator of $E(S)$ in Eq. (1.7).] In almost all queueing simulations, $\tilde{w}(n)$ will be a more efficient (less variable) estimator of $w(n)$ than $\hat{w}(n)$ and is thus preferable (both estimators are unbiased). Therefore, if one wants an estimate of $w(n)$, estimate $d(n)$ and add the known expected service time, $E(S)$. In general, the moral is to replace estimators by their expected values whenever possible (see the discussion of indirect estimators in Sec. 11.5).

1.4.7 Alternative Stopping Rules

In the above queueing example, the simulation was terminated when the number of customers delayed became equal to 1000; the final value of the simulation clock was thus a random variable. However, for many real-world models, the simulation is to stop after some fixed amount of time, say 8 hours. Since the interarrival and service times for our example are continuous random variables, the probability of the simulation's terminating after exactly 480 minutes is 0 (neglecting the finite accuracy of a computer). Therefore, to stop the simulation at a specified time, we introduce a dummy "end-simulation" event (call it an event of type 3), which is scheduled to occur at time 480. When the time of occurrence of this event (being held in the third spot of the event list) is less than all other entries in the event list, the report generator is called and the simulation is terminated. The number of customers delayed is now a random variable.

These ideas can be implemented in the computer programs by making changes to the main program, the initialization routine, and the report generator, as described below. The reader need go through the changes for only one of the languages, but should review carefully the corresponding code.

FORTRAN Program. Changes must be made in the main program, the declarations file (renamed mm1alt.dcl), INIT, and REPORT, as shown in Figs. 1.29 through 1.32. The only changes in TIMING, ARRIVE, DEPART, and UPTAVG are in the file name in the INCLUDE statements, and there are no changes at all in EXPON. In Figs. 1.29 and 1.30, note that we now have 3 events, that the desired simulation run length, TEND, is now an input parameter and a member of the COMMON block MODEL (TOTCUS has been removed), and that the statements after the "computed GO TO" statement have been changed. In the main program (as before), we call UPTAVG before entering an event routine, so that in particular the areas will be updated to the end of the simulation, here when the type 3 event (end simulation) is next. The only change to INIT (other than the file name to INCLUDE) is the addition of the statement TNE(3) = TEND, which schedules the end of the simulation. The only change to REPORT in Fig. 1.32 is to write the number of customers delayed instead of the time the simulation ends, since in this case we know that the ending time will be 480 minutes but will not know how many customer delays will have been completed during that time.

C Program. Changes must be made in the external definitions, the main function, and the initialize and report functions, as shown in Figs. 1.33 through 1.36; the rest of the program is unaltered. In Figs. 1.33 and 1.34, note that we now have 3 events, that the desired simulation run length, time_end, is now an input parameter (num_delays_required has been removed), and that the "switch" statement has

```
*     Main program for single-server queueing system, fixed run length.
*     Bring in declarations file.
      INCLUDE 'mm1alt.dcl'
*     Open input and output files.
      OPEN (5, FILE = 'mm1alt.in')
      OPEN (6, FILE = 'mm1alt.out')
*     Specify the number of event types for the timing routine.
      NEVNTS = 3
*     Set mnemonics for server's being busy and idle.
      BUSY = 1
      IDLE = 0
*     Read input parameters.
      READ (5,*) MARRVT, MSERVT, TEND
*     Write report heading and input parameters.
      WRITE (6,2010) MARRVT, MSERVT, TEND
2010  FORMAT (' Single-server queueing system with fixed run length'//
     &        ' Mean interarrival time',F11.3,' minutes'//
     &        ' Mean service time',F16.3,' minutes'//
     &        ' Length of the simulation',F9.3,' minutes'//)
*     Initialize the simulation.
      CALL INIT
*     Determine the next event.
   10 CALL TIMING
*     Update time-average statistical accumulators.
      CALL UPTAVG
*     Call the appropriate event routine.
      GO TO (20, 30, 40), NEXT
   20    CALL ARRIVE
         GO TO 10
   30    CALL DEPART
         GO TO 10
*        Simulation is over; call report generator and end
*        simulation.
   40    CALL REPORT
      CLOSE (5)
      CLOSE (6)
      STOP
      END
```

FIGURE 1.29
FORTRAN code for the main program, queueing model with fixed run length.


```
      INTEGER QLIMIT
      PARAMETER (QLIMIT = 100)
      INTEGER BUSY, IDLE, NEVNTS, NEXT, NIQ, NUMCUS, SERVER
      REAL ANIQ, AUTIL, MARRVT, MSERVT, TARRVL(QLIMIT), TEND, TIME, TLEVNT,
     &     TNE(3), TOTDEL
      REAL EXPON
      COMMON /MODEL/ ANIQ, AUTIL, BUSY, IDLE, MARRVT, MSERVT, NEVNTS, NEXT, NIQ,
     &               NUMCUS, SERVER, TARRVL, TEND, TIME, TLEVNT, TNE, TOTDEL
```

FIGURE 1.30
FORTRAN code for the declarations file (mm1alt.dcl), queueing model with fixed run length.

been changed. To stop the simulation, the original "while" loop has been replaced by a "do while" loop in Fig. 1.34, where the loop keeps repeating itself as long as the type of event just executed is not 3 (end simulation); after a type 3 event is chosen for execution, the loop ends and the simulation stops. In the main program (as before), we invoke update_time_avg_stats before entering an event function, so that in particular the areas will be updated to the end of the simulation here when the type 3 event (end simulation) is next. The only change to the initialization function in Fig. 1.35 is the addition of the statement time_next_event[3] = time_end, which schedules the end of the simulation. The only change to the report function in Fig. 1.36 is to write the number of customers delayed instead of the time the

```
      SUBROUTINE INIT
      INCLUDE 'mm1alt.dcl'
*     Initialize the simulation clock.
      TIME   = 0.0
*     Initialize the state variables.
      SERVER = IDLE
      NIQ    = 0
      TLEVNT = 0.0
*     Initialize the statistical counters.
      NUMCUS = 0
      TOTDEL = 0.0
      ANIQ   = 0.0
      AUTIL  = 0.0
*     Initialize event list. Since no customers are present, the
*     departure (service completion) event is eliminated from
*     consideration.  The end-simulation event (type 3) is scheduled for
*     time TEND.
      TNE(1) = TIME + EXPON(MARRVT)
      TNE(2) = 1.0E+30
      TNE(3) = TEND
      RETURN
      END
```

FIGURE 1.31
FORTRAN code for subroutine INIT, queueing model with fixed run length.

```
      SUBROUTINE REPORT
      INCLUDE 'mm1alt.dcl'
      REAL AVGDEL,AVGNIQ,UTIL

*     Compute and write estimates of desired measures of performance.

      AVGDEL = TOTDEL / NUMCUS
      AVGNIQ = ANIQ / TIME
      UTIL   = ATTIL / TIME
      WRITE (6,2010) AVGDEL, AVGNIQ, UTIL, NUMCUS
2010  FORMAT (/' Average delay in queue',F11.3,' minutes'//
     &        ' Average number in queue',F10.3//
     &        ' Server utilization',F15.3//
     &        ' Number of delays completed',I7)

      RETURN
      END
```

FIGURE 1.32
FORTRAN code for subroutine REPORT, queueing model with fixed run length.

simulation ends, since in this case we know that the ending time will be 480 minutes but will not know how many customer delays will have been completed during that time.

The output file (named mm1alt.out if either the FORTRAN or C program was run) is shown in Fig. 1.37. The number of customer delays completed was 475 in this run, which seems reasonable in a 480-minute run where customers are arriving at an average rate of 1 per minute. The same three measures of performance are again numerically close to each other, but are all somewhat less than their earlier values in the 1000-delay simulation. A possible reason for this is that the current run is roughly only half as long as the earlier one, and since the initial conditions for the simulation are empty and idle (an uncongested state), the model in this shorter run

```
/* External definitions for single-server queueing system, fixed run length. */

#include <stdio.h>
#include <math.h>
#include "lcgrand.h"  /* Header file for random-number generator. */

#define Q_LIMIT 100  /* Limit on queue length. */
#define BUSY      1  /* Mnemonics for server's being busy */
#define IDLE      0  /* and idle. */

int   next_event_type, num_custs_delayed, num_events, num_in_q, server_status;
float area_num_in_q, area_server_status, mean_interarrival, mean_service,
      sim_time, time_arrival[Q_LIMIT + 1], time_end, time_last_event,
      time_next_event[4], total_of_delays;
FILE  *infile, *outfile;

void  initialize(void);
void  timing(void);
void  arrive(void);
void  depart(void);
void  report(void);
void  update_time_avg_stats(void);
float expon(float mean);
```

FIGURE 1.33
C code for the external definitions, queueing model with fixed run length.


```
main()  /* Main function. */
{
    /* Open input and output files. */

    infile  = fopen("mm1alt.in",  "r");
    outfile = fopen("mm1alt.out", "w");

    /* Specify the number of events for the timing function. */

    num_events = 3;

    /* Read input parameters. */

    fscanf(infile, "%f %f %f", &mean_interarrival, &mean_service, &time_end);

    /* Write report heading and input parameters. */

    fprintf(outfile, "Single-server queueing system with fixed run");
    fprintf(outfile, " length\n\n");
    fprintf(outfile, "Mean interarrival time%11.3f minutes\n\n",
            mean_interarrival);
    fprintf(outfile, "Mean service time%16.3f minutes\n\n", mean_service);
    fprintf(outfile, "Length of the simulation%9.3f minutes\n\n", time_end);

    /* Initialize the simulation. */

    initialize();

    /* Run the simulation until it terminates after an end-simulation event
       (type 3) occurs. */

    do {

        /* Determine the next event. */

        timing();

        /* Update time-average statistical accumulators. */

        update_time_avg_stats();

        /* Invoke the appropriate event function. */

        switch (next_event_type) {
            case 1:
                arrive();
                break;
            case 2:
                depart();
                break;
            case 3:
                report();
                break;
        }

    /* If the event just executed was not the end-simulation event (type 3),
       continue simulating.  Otherwise, end the simulation. */

    } while (next_event_type != 3);

    fclose(infile);
    fclose(outfile);

    return 0;
}
```

FIGURE 1.34
C code for the main function, queueing model with fixed run length.

```
void initialize(void)  /* Initialization function. */
{
    /* Initialize the simulation clock. */

    sim_time = 0.0;

    /* Initialize the state variables. */

    server_status   = IDLE;
    num_in_q        = 0;
    time_last_event = 0.0;

    /* Initialize the statistical counters. */

    num_custs_delayed  = 0;
    total_of_delays    = 0.0;
    area_num_in_q      = 0.0;
    area_server_status = 0.0;

    /* Initialize event list.  Since no customers are present, the departure
       (service completion) event is eliminated from consideration.  The end-
       simulation event (type 3) is scheduled for time time_end. */

    time_next_event[1] = sim_time + expon(mean_interarrival);
    time_next_event[2] = 1.0e+30;
    time_next_event[3] = time_end;
}
```

FIGURE 1.35
C code for function initialize, queueing model with fixed run length.

has less chance to become congested. Again, however, this is just a single run and is thus subject to perhaps considerable uncertainty; there is no easy way to assess the degree of uncertainty from only a single run.

If the queueing system being considered had actually been a one-operator barbershop open from 9 A.M. to 5 P.M., stopping the simulation after exactly 8 hours might leave a customer with hair partially cut. In such a case, we might want to close the door of the barbershop after 8 hours but continue to run the simulation until all customers present when the door closes (if any) have been served. The reader is asked in Prob. 1.10 to supply the program changes necessary to implement this stopping rule (see also Sec. 2.6).

```
void report(void)  /* Report generator function. */
{
    /* Compute and write estimates of desired measures of performance. */

    fprintf(outfile, "\n\nAverage delay in queue%11.3f minutes\n\n",
            total_of_delays / num_custs_delayed);
    fprintf(outfile, "Average number in queue%10.3f\n\n",
            area_num_in_q / sim_time);
    fprintf(outfile, "Server utilization%15.3f\n\n",
            area_server_status / sim_time);
    fprintf(outfile, "Number of delays completed%7d",
            num_custs_delayed);
}
```

FIGURE 1.36
C code for function report, queueing model with fixed run length.


```
Single-server queueing system with fixed run length

Mean interarrival time       1.000 minutes

Mean service time            0.500 minutes

Length of the simulation   480.000 minutes


Average delay in queue       0.399 minutes

Average number in queue      0.394

Server utilization           0.464

Number of delays completed     475
```

FIGURE 1.37
Output report, queueing model with fixed run length.

1.4.8 Determining the Events and Variables

We defined an event in Sec. 1.3 as an instantaneous occurrence that may change the system state, and in the simple single-server queue of Sec. 1.4.1 it was not too hard to identify the events. However, the question sometimes arises, especially for complex systems, of how one determines the number and definition of events in general for a model. It may also be difficult to specify the state variables needed to keep the simulation running in the correct event sequence and to obtain the desired output measures. There is no completely general way to answer these questions, and different people may come up with different ways of representing a model in terms of events and variables, all of which may be correct. But there are some principles and techniques to help simplify the model's structure and to avoid logical errors.

Schruben (1983b) presented an *event-graph* method, which was subsequently refined and extended by Sargent (1988) and Som and Sargent (1989). In this approach proposed events, each represented by a *node*, are connected by *directed arcs* (arrows) depicting how events may be scheduled from other events and from themselves. For example, in the queueing simulation of Sec. 1.4.3, the arrival event schedules another future occurrence of itself and (possibly) a departure event, and the departure event may schedule another future occurrence of itself; in addition, the arrival event must be initially scheduled in order to get the simulation going. Event graphs connect the proposed set of events (nodes) by arcs indicating the type of event scheduling that can occur. In Fig. 1.38 we show the event graph for our

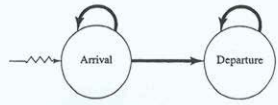


FIGURE 1.38
Event graph, queueing model.

single-server queueing system, where the heavy, smooth arrows indicate that an event at the end of the arrow *may* be scheduled from the event at the beginning of the arrow in a (possibly) *nonzero* amount of time, and the thin jagged arrow indicates that the event at its end is scheduled initially. Thus, the arrival event reschedules itself and may schedule a departure (in the case of an arrival who finds the server idle), and the departure event may reschedule itself (if a departure leaves behind someone else in queue).

For this model, it could be asked why we did not explicitly account for the act of a customer's entering service (either from the queue or upon arrival) as a separate event. This certainly can happen, and it could cause the state to change (i.e., the queue length to fall by 1). In fact, this could have been put in as a separate event without making the simulation incorrect, and would give rise to the event diagram in Fig. 1.39. The two thin smooth arrows each represent an event at the beginning of an arrow potentially scheduling an event at the end of the arrow without any intervening time, i.e., immediately; in this case the straight thin smooth arrow refers to a customer who arrives to an empty system and whose "enter-service" event is thus scheduled to occur immediately, and the curved thin smooth arrow represents a customer departing with a queue left behind, and so the first customer in the queue would be scheduled to enter service immediately. The number of events has now increased by 1, and so we have a somewhat more complicated representation of our model. One of the uses of event graphs is to simplify a simulation's event structure by eliminating unnecessary events. There are several "rules" that allow for simplification, and one of them is that if an event node has incoming arcs that are all thin and smooth (i.e., the only way this event is scheduled is by other events and without any intervening time), then this event can be eliminated from the model and its action built into the events that schedule it in zero time. Here, the "enter-service" event could be eliminated, and its action put partly into the arrival event (when a customer arrives to an idle server and begins service immediately) and partly into the departure event (when a customer finishes service and there is a queue from which the next customer is taken to enter service); this takes us back to the simpler event graph in Fig. 1.38. Basically, "events" that can happen only in conjunction with other events do not need to be in the model. Reducing the number of events not only simplifies model conceptualization, but may also speed its execution. Care must be taken, however, when "collapsing" events in this way to handle priorities and time ties appropriately.

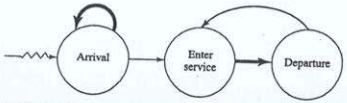


FIGURE 1.39
Event graph, queueing model with separate "enter-service" event.

Another rule has to do with initialization. The event graph is decomposed into *strongly connected* components, within each of which it is possible to "travel" from every node to every other node by following the arcs in their indicated directions. The graph in Fig. 1.38 decomposes into two strongly connected components (with a single node in each), and that in Fig. 1.39 has two strongly connected components (one of which is the arrival node by itself, and the other of which consists of the enter-service and departure nodes). The initialization rule states that in any strongly connected component of nodes that has no incoming arcs from other event nodes outside the component, there must be at least one node that is initially scheduled; if this rule were violated, it would never be possible to execute any of the events in the component. In Figs. 1.38 and 1.39, the arrival node is such a strongly connected component since it has no incoming arcs from other nodes, and so it must be initialized. Figure 1.40 shows the event graph for the queueing model of Sec. 1.4.7 with the fixed run length, for which we introduced the dummy "end-simulation" event. Note that this event is itself a strongly connected component without any arcs coming in, and so it must be initialized; i.e., the end of the simulation is scheduled as part of the initialization. Failure to do so would result in erroneous termination of the simulation.

We have presented only a partial and simplified account of the event-graph technique. There are several other features, including event-canceling relations, ways to combine similar events into one, refining the event-scheduling arcs to include conditional scheduling, and incorporating the state variables needed; see the original paper by Schruben (1983b). Sargent (1988) and Som and Sargent (1989) extend and refine the technique, giving comprehensive illustrations involving a flexible manufacturing system and computer network models. Event graphs can also be used to test whether two apparently different models might in fact be equivalent [Yücesan and Schruben (1992)], as well as to forecast how computationally intensive a model will be when it is executed [Yücesan and Schruben (1998)]. Schruben (1995) provides a software package, SIGMA, that allows on-screen building of an event-graph representation of a simulation model, and then generates code and runs the model. A general event-graph review and tutorial are given by Buss (1996).

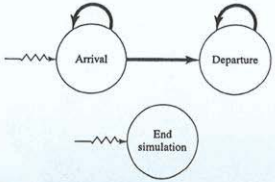


FIGURE 1.40
Event graph, queueing model with fixed run length.

In modeling a system, the event-graph technique can be used to simplify the structure and to detect certain kinds of errors, and is especially useful in complex models involving a large number of interrelated events. Other considerations should also be kept in mind, such as continually asking why a particular state variable is needed; see Prob. 1.4.

1.5 SIMULATION OF AN INVENTORY SYSTEM

We shall now see how simulation can be used to compare alternative ordering policies for an inventory system. Many of the elements of our model are representative of those found in actual inventory systems.

1.5.1 Problem Statement

A company that sells a single product would like to decide how many items it should have in inventory for each of the next n months (n is a fixed input parameter). The times between demands are IID exponential random variables with a mean of 0.1 month. The sizes of the demands, D, are IID random variables (independent of when the demands occur), with

$$D = \begin{cases} 1 & \text{w.p. } \frac{1}{6} \\ 2 & \text{w.p. } \frac{1}{3} \\ 3 & \text{w.p. } \frac{1}{3} \\ 4 & \text{w.p. } \frac{1}{6} \end{cases}$$

where w.p. is read "with probability."

At the beginning of each month, the company reviews the inventory level and decides how many items to order from its supplier. If the company orders Z items, it incurs a cost of $K + iZ$, where $K = \$32$ is the *setup cost* and $i = \$3$ is the *incremental cost* per item ordered. (If $Z = 0$, no cost is incurred.) When an order is placed, the time required for it to arrive (called the *delivery lag* or *lead time*) is a random variable that is distributed uniformly between 0.5 and 1 month.

The company uses a stationary (s, S) policy to decide how much to order, i.e.,

$$Z = \begin{cases} S - I & \text{if } I < s \\ 0 & \text{if } I \geq s \end{cases}$$

where I is the inventory level at the beginning of the month.

When a demand occurs, it is satisfied immediately if the inventory level is at least as large as the demand. If the demand exceeds the inventory level, the excess of demand over supply is backlogged and satisfied by future deliveries. (In this case, the new inventory level is equal to the old inventory level minus the demand size, resulting in a negative inventory level.) When an order arrives, it is first used to eliminate as much of the backlog (if any) as possible; remainder of the order (if any) is added to the inventory.

So far, we have discussed only one type of cost incurred by the inventory system, the ordering cost. However, most real inventory systems also have two additional types of costs, *holding* and *shortage* costs, which we discuss after introducing some additional notation. Let $I(t)$ be the inventory level at time t [note that $I(t)$ could be positive, negative, or zero]; let $I^+(t) = \max\{I(t), 0\}$ be the number of items physically on hand in the inventory at time t [note that $I^+(t) \geq 0$]; and let $I^-(t) = \max\{-I(t), 0\}$ be the backlog at time t [$I^-(t) \geq 0$ as well]. A possible realization of $I(t)$, $I^+(t)$, and $I^-(t)$ is shown in Fig. 1.41. The time points at which $I(t)$ decreases are the ones at which demands occur.

For our model, we shall assume that the company incurs a holding cost of $h = \$1$ per item per month held in (positive) inventory. The holding cost includes such costs as warehouse rental, insurance, taxes, and maintenance, as well as the opportunity cost of having capital tied up in inventory rather than invested elsewhere. We have ignored in our formulation the fact that some holding costs are still incurred when $I^+(t) = 0$. However, since our goal is to *compare* ordering policies, ignoring this factor, which after all is independent of the policy used, will not affect our assessment of which policy is best. Now, since $I^+(t)$ is the number of items held in inventory at time t, the time-average (per month) number of items held in inventory for the n-month period is

$$\bar{I}^+ = \frac{\int_0^n I^+(t)\, dt}{n}$$

which is akin to the definition of the time-average number of customers in queue given in Sec. 1.4.1. Thus, the average holding cost per month is $h\bar{I}^+$.

Similarly, suppose that the company incurs a backlog cost of $\pi = \$5$ per item per month in backlog; this accounts for the cost of extra record keeping when a backlog exists, as well as loss of customers' goodwill. The time-average number of

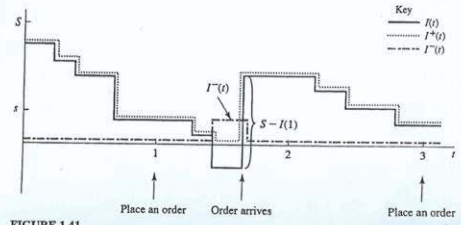


FIGURE 1.41
A realization of $I(t)$, $I^+(t)$, and $I^-(t)$ over time.