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; the 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

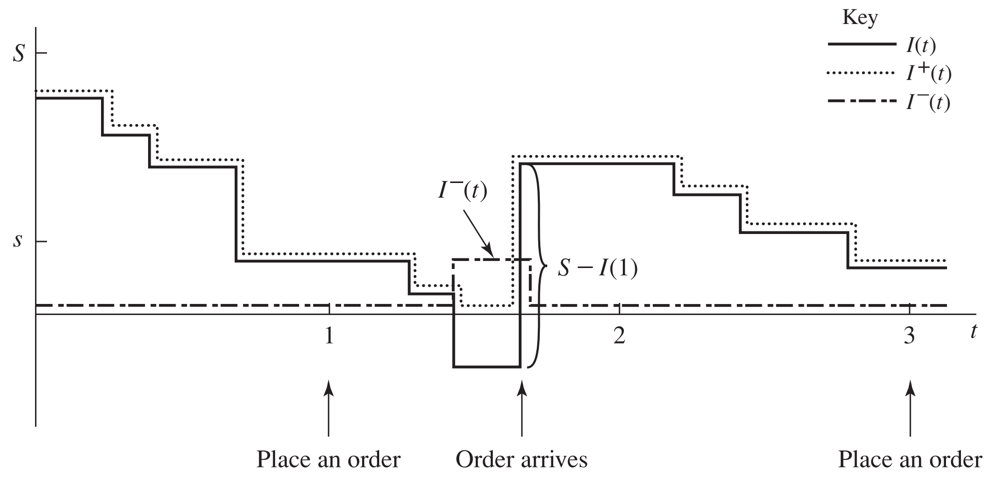

FIGURE 1.28
A realization of $I(t)$, $I^+(t)$, and $I^-(t)$ over time.

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. 1.28. 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 items in backlog is

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

so the average backlog cost per month is $\pi\bar{I}^-$.

Assume that the initial inventory level is $I(0) = 60$ and that no order is outstanding. We simulate the inventory system for $n = 120$ months and use the average total cost per month (which is the sum of the average ordering cost per month, the average holding cost per month, and the average shortage cost per month) to compare the following nine inventory policies:

s	20	20	20	20	40	40	40	60	60
S	40	60	80	100	60	80	100	80	100

We do not address here the issue of how these particular policies were chosen for consideration; statistical techniques for making such a determination are discussed in Chap. 12.

Note that the state variables for a simulation model of this inventory system are the inventory level $I(t)$, the amount of an outstanding order from the company to the supplier, and the time of the last event [which is needed to compute the areas under the $I^+(t)$ and $I^-(t)$ functions].

1.5.2 Program Organization and Logic

Our model of the inventory system uses the following types of events:

Event description	**Event type**
Arrival of an order to the company from the supplier	1
Demand for the product from a customer	2
End of the simulation after n months	3
Inventory evaluation (and possible ordering) at the beginning of a month	4

We have chosen to make the end of the simulation event type 3 rather than type 4, since at time 120 both "end-simulation" and "inventory-evaluation" events will eventually be scheduled and we would like to execute the former event first at this time. (Since the simulation is over at time 120, there is no sense in evaluating the inventory and possibly ordering, incurring an ordering cost for an order that will never arrive.) The execution of event type 3 before event type 4 is guaranteed because the timing routine gives preference to the lowest-numbered event if two or more events are scheduled to occur at the same time. In general, a simulation model should be designed to process events in an appropriate order when time ties occur. An event graph (see Sec. 1.4.7) appears in Fig. 1.29.

There are three types of random variates needed to simulate this system. The interdemand times are distributed exponentially, so the same algorithm (and code) as developed in Sec. 1.4 can be used here. The demand-size random variate D must be discrete, as described above, and can be generated as follows. First divide the unit interval into the contiguous subintervals $C_1 = [0, \frac{1}{6})$, $C_2 = [\frac{1}{6}, \frac{1}{2})$, $C_3 = [\frac{1}{2}, \frac{5}{6})$, and $C_4 = [\frac{5}{6}, 1]$, and obtain a U(0, 1) random variate U from the random-number

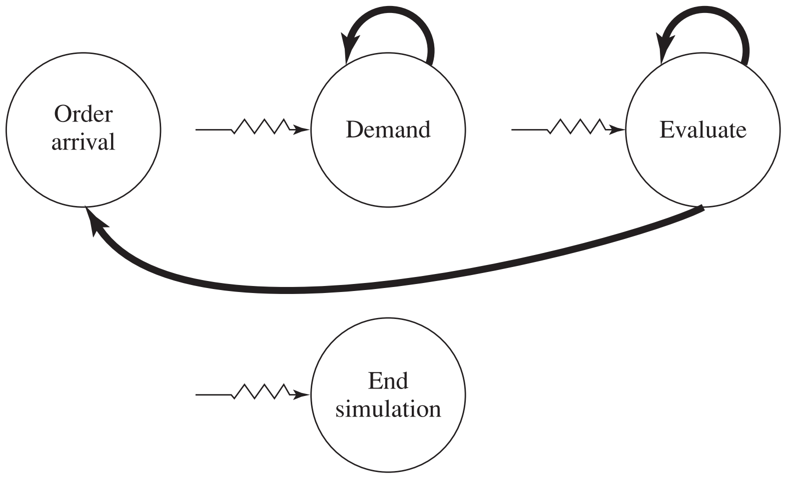

FIGURE 1.29
Event graph, inventory model.

generator. If U falls in C_1, return $D = 1$; if U falls in C_2, return $D = 2$; and so on. Since the width of C_1 is $\frac{1}{6} - 0 = \frac{1}{6}$, and since U is uniformly distributed over [0, 1], the probability that U falls in C_1 (and thus that we return $D = 1$) is $\frac{1}{6}$; this agrees with the desired probability that $D = 1$. Similarly, we return $D = 2$ if U falls in C_2, having probability equal to the width of C_2, $\frac{1}{2} - \frac{1}{6} = \frac{1}{3}$, as desired; and so on for the other intervals. The subprogram to generate the demand sizes uses this principle and takes as input the cutoff points defining the above subintervals, which are the *cumulative* probabilities of the distribution of D.

The delivery lags are uniformly distributed, but not over the unit interval [0,1]. In general, we can generate a random variate distributed uniformly over any interval $[a,b]$ by generating a U(0, 1) random number U, and then returning $a + U(b - a)$. That this method is correct seems intuitively clear, but will be formally justified in Sec. 8.3.1.

We now describe the logic for event types 1, 2, and 4, which actually involve state changes.

The order-arrival event is flowcharted in Fig. 1.30, and must make the changes necessary when an order (which was previously placed) arrives from the supplier. The inventory level is increased by the amount of the order, and the order-arrival event must be eliminated from consideration. (See Prob. 1.12 for consideration of the issue of whether there could be more than one order outstanding at a time for this model with these parameters.)

A flowchart for the demand event is given in Fig. 1.31, and processes the changes necessary to represent a demand's occurrence. First, the demand size is generated, and the inventory is decremented by this amount. Finally, the time of the next demand is scheduled into the event list. Note that this is the place where the inventory level might become negative.

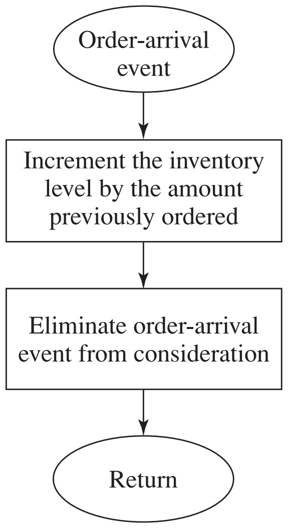
Order-arrival event
Increment the inventory level by the amount previously ordered
Eliminate order-arrival event from consideration
Return

FIGURE 1.30
Flowchart for order-arrival routine, inventory model.

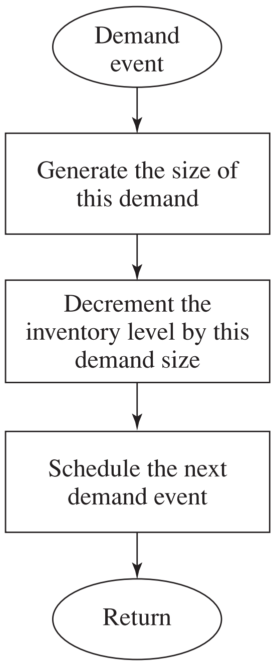
Demand event
Generate the size of this demand
Decrement the inventory level by this demand size
Schedule the next demand event
Return

FIGURE 1.31
Flowchart for demand routine, inventory model.

The inventory-evaluation event, which takes place at the beginning of each month, is flowcharted in Fig. 1.32. If the inventory level $I(t)$ at the time of the evaluation is at least s, then no order is placed, and nothing is done except to schedule the next evaluation into the event list. On the other hand, if $I(t) < s$, we want to place an order for $S - I(t)$ items. This is done by storing the amount of the order $[S - I(t)]$ until the order arrives, and scheduling its arrival time. In this case as well, we want to schedule the next inventory-evaluation event.

As in the single-server queueing model, it is convenient to write a separate nonevent routine to update the continuous-time statistical accumulators. For this model, however, doing so is slightly more complicated, so we give a flowchart for this activity in Fig. 1.33. The principal issue is whether we need to update the area under $I^-(t)$ or $I^+(t)$ (or neither). If the inventory level since the last event has been negative, then we have been in backlog, so the area under $I^-(t)$ only should be updated. On the other hand, if the inventory level has been positive, we need only update the area under $I^+(t)$. If the inventory level has been zero (a possibility), then neither update is needed. The code for this routine also brings the variable for the time of the last event up to the present time. This routine will be invoked from the main program just after returning from the timing routine, regardless of the event type or whether the inventory level is actually changing at this point. This provides a simple (if not the most computationally efficient) way of updating integrals for continuous-time statistics.

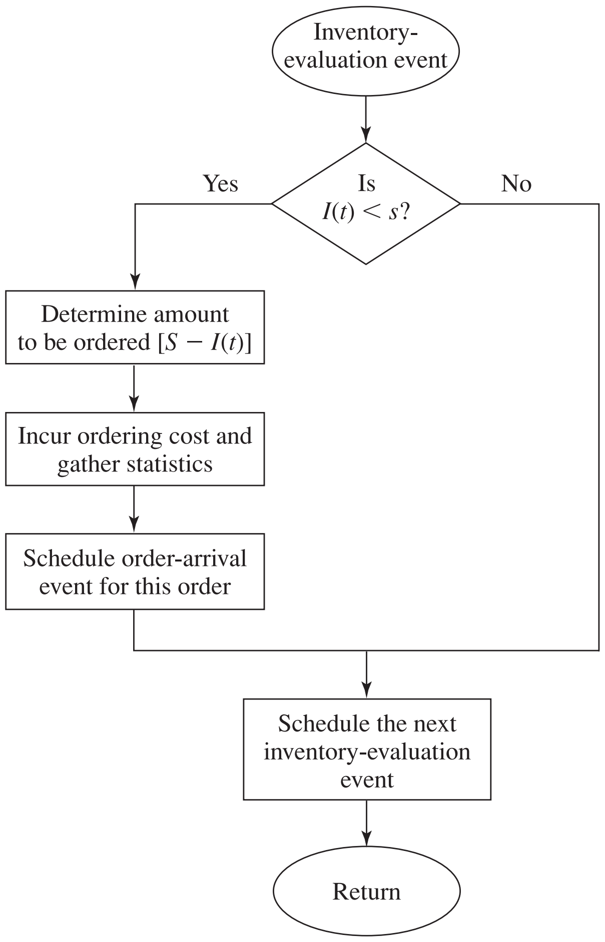
Inventory-evaluation event
Is $I(t) < s$?
Yes
No
Determine amount to be ordered $[S - I(t)]$
Incur ordering cost and gather statistics
Schedule order-arrival event for this order
Schedule the next inventory-evaluation event
Return

FIGURE 1.32
Flowchart for inventory-evaluation routine, inventory model.

Section 1.5.3 contains a program to simulate this model in C. Neither the timing nor exponential-variate-generation subprograms will be shown, as they are the same as for the single-server queueing model in Sec. 1.4. The reader should also note the considerable similarity between the main programs of the queueing and inventory models.

1.5.3 C Program

The external definitions are shown in Fig. 1.34. The array prob_distrib_demand will be used to hold the cumulative probabilities for the demand sizes, and is passed into the random-integer-generation function random_integer. As for the queueing model, we must include the header file lcgrand.h (in Fig. 7.6) for the random-number generator of Fig. 7.5. All code is available at www.mhhe.com/law.

The code for the main function is given in Fig. 1.35. After opening the input and output files, the number of events is set to 4. The input parameters (except s and S) are then read in and written out, and a report heading is produced; for each (s, S)

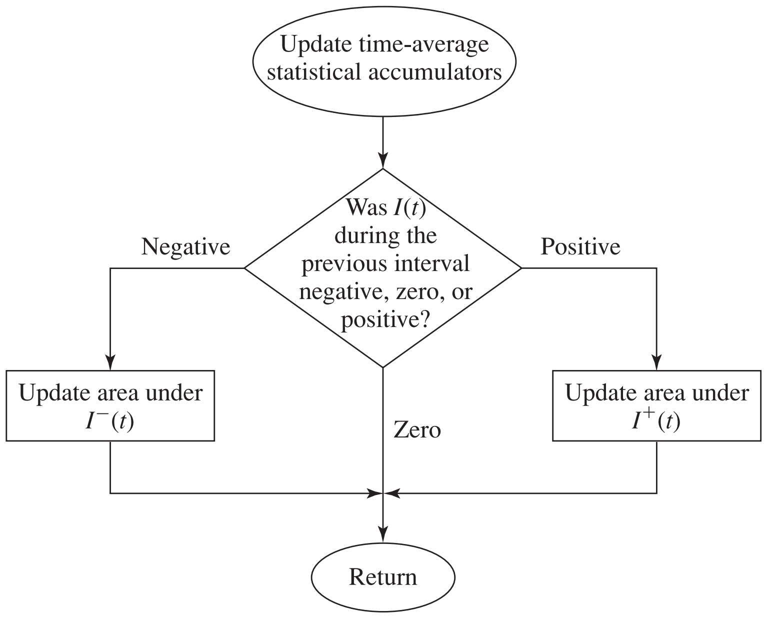
Update time-average statistical accumulators
Was $I(t)$ during the previous interval negative, zero, or positive?
Negative
Positive
Zero
Update area under $I^{-}(t)$
Update area under $I^{+}(t)$
Return

FIGURE 1.33
Flowchart for routine to update the continuous-time statistical accumulators, inventory model.

```
/* External definitions for inventory system. */

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

int   amount, bigs, initial_inv_level, inv_level, next_event_type, num_events,
      num_months, num_values_demand, smalls;
float area_holding, area_shortage, holding_cost, incremental_cost, maxlag,
      mean_interdemand, minlag, prob_distrib_demand[26], setup_cost,
      shortage_cost, sim_time, time_last_event, time_next_event[5],
      total_ordering_cost;
FILE  *infile, *outfile;

void  initialize(void);
void  timing(void);
void  order_arrival(void);
void  demand(void);
void  evaluate(void);
void  report(void);
void  update_time_avg_stats(void);
float expon(float mean);
int   random_integer(float prob_distrib []);
float uniform(float a, float b);
```

FIGURE 1.34
C code for the external definitions, inventory model.

```
main()  /* Main function. */
{
    int i, num_policies;

    /* Open input and output files. */

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

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

    num_events = 4;

    /* Read input parameters. */

    fscanf(infile, "%d %d %d %d %f %f %f %f %f %f %f",
           &initial_inv_level, &num_months, &num_policies, &num_values_demand,
           &mean_interdemand, &setup_cost, &incremental_cost, &holding_cost,
           &shortage_cost, &minlag, &maxlag);
    for (i = 1; i <= num_values_demand; ++i)
        fscanf(infile, "%f", &prob_distrib_demand[i]);

    /* Write report heading and input parameters. */

    fprintf(outfile, "Single-product inventory system\n\n");
    fprintf(outfile, "Initial inventory level%24d items\n\n",
            initial_inv_level);
    fprintf(outfile, "Number of demand sizes%25d\n\n", num_values_demand);
    fprintf(outfile, "Distribution function of demand sizes  ");
    for (i = 1; i <= num_values_demand; ++i)
        fprintf(outfile, "%8.3f", prob_distrib_demand[i]);
    fprintf(outfile, "\n\nMean interdemand time%26.2f\n\n", mean_interdemand);
    fprintf(outfile, "Delivery lag range%29.2f to%10.2f months\n\n", minlag,
            maxlag);
    fprintf(outfile, "Length of the simulation%23d months\n\n", num_months);
    fprintf(outfile, "K =%6.1f   i =%6.1f   h =%6.1f   pi =%6.1f\n\n",
            setup_cost, incremental_cost, holding_cost, shortage_cost);
    fprintf(outfile, "Number of policies%29d\n\n", num_policies);
    fprintf(outfile, "                 Average        Average");
    fprintf(outfile, "        Average        Average\n");
    fprintf(outfile, "  Policy       total cost    ordering cost");
    fprintf(outfile, "  holding cost   shortage cost");

    /* Run the simulation varying the inventory policy. */

    for (i = 1; i <= num_policies; ++i) {

        /* Read the inventory policy, and initialize the simulation. */

        fscanf(infile, "%d %d", &smalls, &bigs);
        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. */
```

FIGURE 1.35
C code for the main function, inventory model.

```
            switch (next_event_type) {
                case 1:
                    order_arrival();
                    break;
                case 2:
                    demand();
                    break;
                case 4:
                    evaluate();
                    break;
                case 3:
                    report();
                    break;
            }

        /* If the event just executed was not the end-simulation event (type 3),
           continue simulating.  Otherwise, end the simulation for the current
           (s,S) pair and go on to the next pair (if any). */

        } while (next_event_type != 3);
    }

    /* End the simulations. */

    fclose(infile);
    fclose(outfile);

    return 0;
}
```

FIGURE 1.35
(*continued*)

pair the simulation will then produce in the report function a single line of output corresponding to this heading. Then a "for" loop is begun, each iteration of which performs an entire simulation for a given (s, S) pair; the first thing done in the loop is to read the next (s, S) pair. The model is initialized, and a "do while" loop is used to keep simulating as long as the type 3 (end-simulation) event does not occur, as in Sec. 1.4.6. Inside this loop, the timing function is used to determine the next event type and to update the simulation clock. After returning from timing with the next event type, the continuous-time statistics are updated before executing the event routine itself. A "switch" statement is then used as before to transfer control to the appropriate event routine. Unlike the fixed-time stopping rule of Sec. 1.4.6, when the "do while" loop ends here, we do not stop the program, but go to the next step of the enclosing "for" loop to read in the next (s, S) pair and do a separate simulation; the entire program stops only when the "for" loop is over and there are no more (s, S) pairs to consider.

The initialization function appears in Fig. 1.36. Observe that the first inventory evaluation is scheduled at time 0 since, in general, the initial inventory level could be less than s. Note also that event type 1 (order arrival) is eliminated from consideration, since our modeling assumption was that there are no outstanding orders initially.

The event functions order_arrival, demand, and evaluate are shown in Figs. 1.37 through 1.39, and correspond to the general discussion given in Sec. 1.5.2, and

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

    sim_time = 0.0;

    /* Initialize the state variables. */

    inv_level       = initial_inv_level;
    time_last_event = 0.0;

    /* Initialize the statistical counters. */

    total_ordering_cost = 0.0;
    area_holding        = 0.0;
    area_shortage       = 0.0;

    /* Initialize the event list.  Since no order is outstanding, the order-
       arrival event is eliminated from consideration. */

    time_next_event[1] = 1.0e+30;
    time_next_event[2] = sim_time + expon(mean_interdemand);
    time_next_event[3] = num_months;
    time_next_event[4] = 0.0;
}
```

FIGURE 1.36
C code for function initialize, inventory model.

```
void order_arrival(void)  /* Order arrival event function. */
{
    /* Increment the inventory level by the amount ordered. */

    inv_level += amount;

    /* Since no order is now outstanding, eliminate the order-arrival event from
       consideration. */

    time_next_event[1] = 1.0e+30;
}
```

FIGURE 1.37
C code for function order_arrival, inventory model.

```
void demand(void)  /* Demand event function. */
{
    /* Decrement the inventory level by a generated demand size. */

    inv_level -= random_integer(prob_distrib_demand);

    /* Schedule the time of the next demand. */

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

FIGURE 1.38
C code for function demand, inventory model.

```
void evaluate(void)  /* Inventory-evaluation event function. */
{
    /* Check whether the inventory level is less than smalls. */

    if (inv_level < smalls) {

        /* The inventory level is less than smalls, so place an order for the
           appropriate amount. */

        amount               = bigs - inv_level;
        total_ordering_cost += setup_cost + incremental_cost * amount;

        /* Schedule the arrival of the order. */

        time_next_event[1] = sim_time + uniform(minlag, maxlag);
    }

    /* Regardless of the place-order decision, schedule the next inventory
       evaluation. */

    time_next_event[4] = sim_time + 1.0;
}
```

FIGURE 1.39
C code for function evaluate, inventory model.

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

    float avg_holding_cost, avg_ordering_cost, avg_shortage_cost;

    avg_ordering_cost = total_ordering_cost / num_months;
    avg_holding_cost  = holding_cost * area_holding / num_months;
    avg_shortage_cost = shortage_cost * area_shortage / num_months;
    fprintf(outfile, "\n\n(%3d,%3d)%15.2f%15.2f%15.2f%15.2f",
            smalls, bigs,
            avg_ordering_cost + avg_holding_cost + avg_shortage_cost,
            avg_ordering_cost, avg_holding_cost, avg_shortage_cost);
}
```

FIGURE 1.40
C code for function report, inventory model.

to the flowcharts in Figs. 1.30 through 1.32. In evaluate, note that the variable total_ordering_cost is increased by the ordering cost for any order that might be placed here.

The report generator is listed in Fig. 1.40, and computes the three components of the total cost separately, adding them together to get the average total cost per month. The current values of s and S are written out for identification purposes, along with the average total cost and its three components (ordering, holding, and shortage costs).

Function update_time_avg_stats, which was discussed in general in Sec. 1.5.2 and flowcharted in Fig. 1.33, is shown in Fig. 1.41. Note that if the inventory level inv_level is zero, neither the "if" nor the "else if" condition is satisfied, resulting in no update at all, as desired. As in the single-server queueing model of Sec. 1.4, it

```
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;

    /* Determine the status of the inventory level during the previous interval.
       If the inventory level during the previous interval was negative, update
       area_shortage.  If it was positive, update area_holding.  If it was zero,
       no update is needed. */

    if (inv_level < 0)
        area_shortage -= inv_level * time_since_last_event;
    else if (inv_level > 0)
        area_holding  += inv_level * time_since_last_event;
}
```

FIGURE 1.41
C code for function update_time_avg_stats, inventory model.

```
int random_integer(float prob_distrib[])  /* Random integer generation
                                             function. */
{
    int   i;
    float u;

    /* Generate a U(0,1) random variate. */

    u = lcgrand(1);

    /* Return a random integer in accordance with the (cumulative) distribution
       function prob_distrib. */

    for (i = 1; u >= prob_distrib[i]; ++i)
        ;
    return i;
}
```

FIGURE 1.42
C code for function random_integer.

might be necessary to make both the sim_time and time_last_event variables be of type double to avoid severe roundoff error in their subtraction at the top of the routine if the simulation is to be run for a long period of simulated time.

The code for function random_integer is given in Fig. 1.42, and is general in that it will generate an integer according to distribution function prob_distrib[I], provided that the values of prob_distrib[I] are specified. (In our case, prob_distrib[1] $= \frac{1}{6}$, prob_distrib[2] $= \frac{1}{2}$, prob_distrib[3] $= \frac{5}{6}$, and prob_distrib[4] $= 1$, all specified to three-decimal accuracy on input.) The logic agrees with the discussion in Sec. 1.5.2; note that the input array prob_distrib must contain the *cumulative* distribution function rather than the probabilities that the variate takes on its possible values.

The function uniform is given in Fig. 1.43, and is as described in Sec. 1.5.2.

```
float uniform(float a, float b)  /* Uniform variate generation function. */
{
    /* Return a U(a,b) random variate. */

    return a + lcgrand(1) * (b - a);
}
```

FIGURE 1.43
C code for function uniform.

1.5.4 Simulation Output and Discussion

The simulation report (in file inv.out) is given in Fig. 1.44. For this model, there were some differences in the results across different compilers and computers, even though the same random-number-generator algorithm was being used; see the discussion at the beginning of Sec. 1.4.4 for an explanation of this discrepancy.

```
Single-product inventory system

Initial inventory level                      60 items

Number of demand sizes                        4

Distribution function of demand sizes     0.167   0.500   0.833   1.000

Mean interdemand time                       0.10 months

Delivery lag range                          0.50 to      1.00 months

Length of the simulation                     120 months

K =  32.0    i =    3.0    h =   1.0    pi =   5.0

Number of policies                            9

                    Average        Average        Average        Average
  Policy           total cost    ordering cost  holding cost   shortage cost

( 20, 40)           126.61          99.26           9.25          18.10

( 20, 60)           122.74          90.52          17.39          14.83

( 20, 80)           123.86          87.36          26.24          10.26

( 20,100)           125.32          81.37          36.00           7.95

( 40, 60)           126.37          98.43          25.99           1.95

( 40, 80)           125.46          88.40          35.92           1.14

( 40,100)           132.34          84.62          46.42           1.30

( 60, 80)           150.02         105.69          44.02           0.31

( 60,100)           143.20          89.05          53.91           0.24
```

FIGURE 1.44
Output report, inventory model.

The three separate components of the average total cost per month were reported to see how they respond individually to changes in s and S, as a possible check on the model and the code. For example, fixing $s = 20$ and increasing S from 40 to 100 increases the holding cost steadily from \$9.25 per month to \$36.00 per month, while reducing shortage cost at the same time; the effect of this increase in S on the ordering cost is to reduce it, evidently since ordering up to larger values of S implies that these larger orders will be placed less frequently, thereby avoiding the fixed ordering cost more often. Similarly, fixing S at, say, 100, and increasing s from 20 to 60 leads to a decrease in shortage cost (\$7.95, \$1.30, \$0.24) but an increase in holding cost (\$36.00, \$46.42, \$53.91), since increases in s translate into less willingness to let the inventory level fall to low values. While we could probably have predicted the *direction* of movement of these components of cost without doing the simulation, it would not have been possible to say much about their *magnitude* without the aid of the simulation output.

Since the overall criterion of *total* cost per month is the sum of three components that move in sometimes different directions in reaction to changes in s and S, we cannot predict even the direction of movement of this criterion without the simulation. Thus, we simply look at the values of this criterion, and it would *appear* that the (20, 60) policy is the best, having an average total cost of \$122.74 per month. However, in the present context where the length of the simulation is fixed (the company wants a planning horizon of 10 years), what we *really* want to estimate for each policy is the *expected* average total cost per month for the first 120 months. The numbers in Fig. 1.44 are *estimates* of these expected values, each estimate based on a sample of size *1* (simulation run or replication). Since these estimates may have large variances, the ordering of them may differ considerably from the ordering of the expected values, which is the desired information. In fact, if we reran the nine simulations using different U(0, 1) random variates, the estimates obtained might differ greatly from those in Fig. 1.44. Furthermore, the ordering of the new estimates might also be different.

We conclude from the above discussion that when the simulation run length is fixed by the problem context, it will generally not be sufficient to make a single simulation run of each policy or system of interest. In Chap. 9 we address the issue of just how many runs are required to get a good estimate of a desired expected value. Chapters 10 and 12 consider related problems when we are concerned with several different expected values arising from alternative system designs.

1.6 PARALLEL/DISTRIBUTED SIMULATION AND THE HIGH LEVEL ARCHITECTURE

The simulations in Secs. 1.4 and 1.5 (as well as those to be considered in Chap. 2) all operate in basically the same way. A simulation clock and an event list interact to determine which event will be processed next, the simulation clock is advanced to the time of this event, and the computer executes the event logic, which may include updating state variables, updating the event list, and collecting statistics. This