7.8.3 Extended Example: Discrete-Event Simulation in R

Discrete-event simulation (DES) is widely used in business, industry, and government. The term *discrete event* refers to the fact that the state of the system changes only in discrete quantities, rather than changing continuously.

A typical example would involve a queuing system, say people lining up to use an ATM. Let's define the state of our system at time t to be the number of people in the queue at that time. The state changes only by $+1$, when someone arrives, or by -1, when a person finishes an ATM transaction. This is in contrast to, for instance, a simulation of weather, in which temperature, barometric pressure, and so on change continuously.

This will be one of the longer, more involved examples in this book. But it exemplifies a number of important issues in R, especially concerning global variables, and will serve as an example when we discuss appropriate use global variables in the next section. Your patience will turn out to be a good investment of time. (It is not assumed here that the reader has any prior background in DES.)

Central to DES operation is maintenance of the *event list*, which is simply a list of scheduled events. This is a general DES term, so the word *list* here does not refer to the R data type. In fact, we'll represent the event list by a data frame.

In the ATM example, for instance, the event list might at some point in the simulation look like this:

```
customer 1 arrives at time 23.12
customer 2 arrives at time 25.88
customer 3 arrives at time 25.97
customer 1 finishes service at time 26.02
```

Since the earliest event must always be handled next, the simplest form of coding the event list is to store it in time order, as in the example. (Readers with computer science background might notice that a more efficient approach might be to use some kind of binary tree for storage.) Here, we will implement it as a data frame, with the first row containing the earliest scheduled event, the second row containing the second earliest, and so on.

The main loop of the simulation repeatedly iterates. Each iteration pulls the earliest event off of the event list, updates the simulated time to reflect the occurrence of that event, and reacts to this event. The latter action will typically result in the creation of new events. For example, if a customer arrival occurs when the queue is empty, that customer's service will begin—one event triggers setting up another. Our code must determine the customer's service time, and then it will know the time at which service will be finished, which is another event that must be added to the event list.

One of the oldest approaches to writing DES code is the *event-oriented paradigm*. Here, the code to handle the occurrence of one event directly sets up another event, reflecting our preceding discussion.

As an example to guide your thinking, consider the ATM situation. At time 0, the queue is empty. The simulation code randomly generates the time of the first arrival, say 2.3. At this point, the event list is simply

(2.3,"arrival"). This event is pulled off the list, simulated time is updated to 2.3, and we react to the arrival event as follows:

- The queue for the ATM is empty, so we start the service by randomly generating the service time—say it is 1.2 time units. Then the completion of service will occur at simulated time $2.3 + 1.2 = 3.5$.
- We add the completion of service event to the event list, which will now consist of (3.5,"service done")).
- We also generate the time to the next arrival, say 0.6, which means the arrival will occur at time 2.9. Now the event list consists of (2.9,"arrival") and (3.5,"service done").

The code consists of a generally applicable library. We also have an example application, which simulates an M/M/1 queue, which is a single-server queue in which both interarrival time and service time are exponentially distributed.

NOTE *The code in this example is hardly optimal, and the reader is invited to improve it, especially by rewriting some portions in C. (Chapter 15 shows how to interface C to R.) This example does, however, serve to illustrate a number of the issues we have discussed in this chapter.*

Here is a summary of the library functions:

- `schedevnt()`: Inserts a newly created event into the event list.
- `getnextevnt()`: Pulls the earliest event off the event list.
- `dosim()`: Includes the core loop of the simulation. Repeatedly calls `getnextevnt()` to get the earliest of the pending events; updates the current simulated time, `sim$currtime`, to reflect the occurrence of that event; and calls the application-specific function `reactevnt()` to process this newly occurred event.

The code uses the following application-specific functions:

- `initglbls()`: Initializes the application-specific global variables.
- `reactevnt()`: Takes the proper actions when an event occurs, typically generating new events as a result.
- `prntrslts()`: Prints the application-specific results of the simulation.

Note that `initglbls()`, `reactevnt()`, and `prntrslts()` are written by the application programmer and then passed to `dosim()` as arguments. In the M/M/1 queue example included here, these functions are named `mm1initglbls()`, `mm1reactevnt()`, and `mm1prntrslts()`, respectively. Thus, in correspondence with the definition of `dosim()`,

```
dosim <- function(initglbls,reactevnt,prntrslts,maxsimtime,apppars=NULL,dbg=FALSE){
```

our call is as follows:

```
dosim(mm1initglbls,mm1reactevnt,mm1prntrslts,10000.0,
   list(arrvrate=0.5,srvrate=1.0))
```

Here's the library code:

```
# DES.R:  R routines for discrete-event simulation (DES)

# each event will be represented by a data frame row consisting of the
# following components:  evnttime, the time the event is to occur;
# evnttype, a character string for the programmer-defined event type;
# optional application-specific components, e.g.
# the job's arrival time in a queuing app

# a global list named "sim" holds the events data frame, evnts, and
# current simulated time, currtime; there is also a component dbg, which
# indicates debugging mode

# forms a row for an event of type evntty that will occur at time
# evnttm; see comments in schedevnt() regarding appin
evntrow <- function(evnttm,evntty,appin=NULL) {
   rw <- c(list(evnttime=evnttm,evnttype=evntty),appin)
   return(as.data.frame(rw))
}

# insert event with time evnttm and type evntty into event list;
# appin is an optional set of application-specific traits of this event,
# specified in the form a list with named components
schedevnt <- function(evnttm,evntty,appin=NULL) {
   newevnt <- evntrow(evnttm,evntty,appin)
   # if the event list is empty, set it to consist of evnt and return
   if (is.null(sim$evnts)) {
      sim$evnts <<- newevnt
      return()
   }
   # otherwise, find insertion point
   inspt <- binsearch((sim$evnts)$evnttime,evnttm)
   # now "insert," by reconstructing the data frame; we find what
   # portion of the current matrix should come before the new event and
   # what portion should come after it, then string everything together
   before <-
      if (inspt == 1) NULL else sim$evnts[1:(inspt-1),]
   nr <- nrow(sim$evnts)
   after <- if (inspt <= nr) sim$evnts[inspt:nr,] else NULL
   sim$evnts <<- rbind(before,newevnt,after)
}

```

```
# binary search of insertion point of y in the sorted vector x; returns
# the position in x before which y should be inserted, with the value
# length(x)+1 if y is larger than x[length(x)]; could be changed to C
# code for efficiency
binsearch <- function(x,y) {
   n <- length(x)
   lo <- 1
   hi <- n
   while(lo+1 < hi) {
      mid <- floor((lo+hi)/2)
      if (y == x[mid]) return(mid)
      if (y < x[mid]) hi <- mid else lo <- mid
   }
   if (y <= x[lo]) return(lo)
   if (y < x[hi]) return(hi)
   return(hi+1)
}

# start to process next event (second half done by application
# programmer via call to reactevnt())
getnextevnt <- function() {
   head <- sim$evnts[1,]
   # delete head
   if (nrow(sim$evnts) == 1) {
      sim$evnts <<- NULL
   } else sim$evnts <<- sim$evnts[-1,]
   return(head)
}

# simulation body
# arguments:
#    initglbls:  application-specific initialization function; inits
#      globals to statistical totals for the app, etc.; records apppars
#      in globals; schedules the first event
#    reactevnt: application-specific event handling function, coding the
#       proper action for each type of event
#    prntrslts:  prints application-specific results, e.g. mean queue
#       wait
#    apppars:  list of application-specific parameters, e.g.
#      number of servers in a queuing app
#    maxsimtime:  simulation will be run until this simulated time
#    dbg:  debug flag; if TRUE, sim will be printed after each event
dosim <- function(initglbls,reactevnt,prntrslts,maxsimtime,apppars=NULL,
      dbg=FALSE) {
   sim <<- list()
   sim$currtime <<- 0.0  # current simulated time
   sim$evnts <<- NULL  # events data frame
```

```
   sim$dbg <<- dbg
   initglbls(apppars)
   while(sim$currtime < maxsimtime) {
      head <- getnextevnt()
      sim$currtime <<- head$evnttime  # update current simulated time
      reactevnt(head)  # process this event
      if (dbg) print(sim)
   }
   prntrslts()
}
```

The following is an example application of the code. Again, the simulation models an M/M/1 queue, which is a single-server queuing system in which service times and times between job arrivals are exponentially distributed.

```
# DES application:  M/M/1 queue, arrival rate 0.5, service rate 1.0

# the call
# dosim(mm1initglbls,mm1reactevnt,mm1prntrslts,10000.0,
#    list(arrvrate=0.5,srvrate=1.0))
# should return a value of about 2 (may take a while)

# initializes global variables specific to this app
mm1initglbls <- function(apppars) {
   mm1glbls <<- list()
   # simulation parameters
   mm1glbls$arrvrate <<- apppars$arrvrate
   mm1glbls$srvrate <<- apppars$srvrate
   # server queue, consisting of arrival times of queued jobs
   mm1glbls$srvq <<- vector(length=0)
   # statistics
   mm1glbls$njobsdone <<- 0  # jobs done so far
   mm1glbls$totwait <<- 0.0  # total wait time so far
   # set up first event, an arrival; the application-specific data for
   # each event will consist of its arrival time, which we need to
   # record in order to later calculate the job's residence time in the
   # system
   arrvtime <- rexp(1,mm1glbls$arrvrate)
   schedevnt(arrvtime,"arrv",list(arrvtime=arrvtime))
}

# application-specific event processing function called by dosim()
# in the general DES library
mm1reactevnt <- function(head) {
   if (head$evnttype == "arrv") {  # arrival
      # if server free, start service, else add to queue (added to queue
      # even if empty, for convenience)
```

```
33        if (length(mm1glbls$srvq) == 0) {
34           mm1glbls$srvq <<- head$arrvtime
35           srvdonetime <- sim$currtime + rexp(1,mm1glbls$srvrate)
36           schedevnt(srvdonetime,"srvdone",list(arrvtime=head$arrvtime))
37        } else mm1glbls$srvq <<- c(mm1glbls$srvq,head$arrvtime)
38        # generate next arrival
39        arrvtime <- sim$currtime + rexp(1,mm1glbls$arrvrate)
40        schedevnt(arrvtime,"arrv",list(arrvtime=arrvtime))
41     } else {  # service done
42        # process job that just finished
43        # do accounting
44        mm1glbls$njobsdone <<- mm1glbls$njobsdone + 1
45        mm1glbls$totwait <<-
46           mm1glbls$totwait + sim$currtime - head$arrvtime
47        # remove from queue
48        mm1glbls$srvq <<- mm1glbls$srvq[-1]
49        # more still in the queue?
50        if (length(mm1glbls$srvq) > 0) {
51           # schedule new service
52           srvdonetime <- sim$currtime + rexp(1,mm1glbls$srvrate)
53           schedevnt(srvdonetime,"srvdone",list(arrvtime=mm1glbls$srvq[1]))
54        }
55     }
56  }
57
58  mm1prntrslts <- function() {
59     print("mean wait:")
60     print(mm1glbls$totwait/mm1glbls$njobsdone)
61  }
```

To see how all this works, take a look at the M/M/1 application code. There, we have set up a global variable, `mm1glbls`, which contains variables relevant to the M/M/1 code, such as `mm1glbls$totwait`, the running total of the wait time of all jobs simulated so far. As you can see, the superassignment operator is used to write to such variables, as in this statement:

```
mm1glbls$srvq <<- mm1glbls$srvq[-1]
```

Let's look at `mm1reactevnt()` to see how the simulation works, focusing on the code portion in which a "service done" event is handled.

```
} else {  # service done
   # process job that just finished
   # do accounting
   mm1glbls$njobsdone <<- mm1glbls$njobsdone + 1
   mm1glbls$totwait <<-
      mm1glbls$totwait + sim$currtime - head$arrvtime
   # remove this job from queue
```

```
   mm1glbls$srvq <<- mm1glbls$srvq[-1]
   # more still in the queue?
   if (length(mm1glbls$srvq) > 0) {
      # schedule new service
      srvdonetime <- sim$currtime + rexp(1,mm1glbls$srvrate)
      schedevnt(srvdonetime,"srvdone",list(arrvtime=mm1glbls$srvq[1]))
   }
}
```

First, this code does some bookkeeping, updating the totals of number of jobs completed and wait time. It then removes this newly completed job from the server queue. Finally, it checks if there are still jobs in the queue and, if so, calls `schedevnt()` to arrange for the service of the one at the head.

What about the DES library code itself? First note that the simulation state, consisting of the current simulated time and the event list, has been placed in an R list structure, `sim`. This was done in order to encapsulate all the main information into one package, which in R, typically means using a list. The `sim` list has been made a global variable.

As mentioned, a key issue in writing a DES library is the event list. This code implements it as a data frame, `sim$evnts`. Each row of the data frame corresponds to one scheduled event, with information about the event time, a character string representing the event type (say arrival or service completion), and any application-specific data the programmer wishes to add. Since each row consists of both numeric and character data, it was natural to choose a data frame for representing this event list. The rows of the data frame are in ascending order of event time, which is contained in the first column.

The main loop of the simulation is in `dosim()` of the DES library code, beginning at line 91:

```
while(sim$currtime < maxsimtime) {
   head <- getnextevnt()
   sim$currtime <<- head$evnttime  # update current simulated time
   reactevnt(head)  # process this event
   if (dbg) print(sim)
}
```

First, `getnextevnt()` is called to remove the head (the earliest event) from the event list. (Note the side effect: The event list changes.) Then the current simulated time is updated according to the scheduled time in the head event. Finally, the programmer-supplied function `reactevnt()` is called to process the event (as seen in the M/M/1 code discussed earlier).

The main potential advantage of using a data frame as our structure here is that it enables us to maintain the event list in ascending order by

time via a binary search operation by event time. This is done in line 31 within `schedevnt()`, the function that inserts a newly created event into the event list:

```
inspt <- binsearch((sim$evnts)$evnttime,evnttm)
```

Here, we wish to insert a newly created event into the event list, and the fact that we are working with a vector enables the use of a fast binary search. (As noted in the comments in the code, though, this really should be implemented in C for good performance.)

A later line in `schedevnt()`is a good example of the use of `rbind()`:

```
sim$evnts <<- rbind(before,newevnt,after)
```

Now, we have extracted the events in the event list whose times are earlier than that of `evnt` and stored them in `before`. We also constructed a similar set in `after` for the events whose times are later than that of `newevnt`. We then use `rbind()` to put all these together in the proper order.

7.8.4 When Should You Use Global Variables?

Use of global variables is a subject of controversy in the programming community. Obviously, the question raised by the title of this section cannot be answered in any formulaic way, as it is a matter of personal taste and style. Nevertheless, most programmers would probably consider the outright banning of global variables, which is encouraged by many teachers of programming, to be overly rigid. In this section, we will explore the possible value of globals in the context of the structures of R. Here, the term *global variable*, or just *global*, will be used to include any variable located higher in the environment hierarchy than the level of the given code of interest.

The use of global variables in R is more common than you may have guessed. You might be surprised to learn that R itself makes very substantial use of globals internally, both in its C code and in its R routines. The superassignment operator `<<-`, for instance, is used in many of the R library functions (albeit typically in writing to a variable just one level up in the environment hierarchy). *Threaded* code and *GPU* code, which are used for writing fast programs (as described in Chapter 16), tend to make heavy use of global variables, which provide the main avenue of communication between parallel actors.

Now, to make our discussion concrete, let's return to the earlier example from Section 7.7:

```
f <- function(lxxyy) {  # lxxyy is a list containing x and y
   ...
   lxxyy$x <- ...
   lxxyy$y <- ...
   return(lxxyy)
}
```