

ECS 145 Term Project Report

DES implementations in R and Python

PETER VAN AUSDELN, HENRY JUE, NIKLAS KRAEMER
University of California, Davis
March 20, 2018

Contents

1	Introduction: What is DES?	4
2	Event-Oriented DES	5
3	Process-Oriented DES	9
4	Rposim Package	10
5	Simmer Package	15
6	Who did What?	19

Listings

1	Skip Time Algorithm	5
2	DES.R Code	6
3	Rposim Code	11
4	Simmer Excerpts from MachRep (in order of mention)	17
5	Full Simmer MachRep Implementation	18

1 Introduction: What is DES?

Discrete Event Simulation (DES) is a way of modelling a system and the events that occur within said system. To be a valid system for DES, the events of this system must occur in distinct blocks of time. The average number of points scored by a basketball player over the course of a season would be a valid use of DES. As each game is a discrete event and each point total is a discrete number. The changes in wind speed in a certain location would, on the other hand, not be a valid use of DES as these values are continuous.

To model DES with a programming language package or library, the package must have specific functionality. First, it must be able to simulate a given length of time. This will allow the package to accumulate data from the events as they occur over time. One way of doing this involves storing a time variable and a time limit variable in the class. To simulate passing time, the package increments the time variable at certain points, and ends the simulation once the time variable passes the time limit variable.

The package must also be able to decide when the events occur during that time. Packages usually do this through keeping a schedule of events, often in a queue. When the simulated time of the package reaches the time that the event is scheduled to occur, the event is popped from the queue, handled, and data is gathered from it. This schedule allows the code, in some implementations, to handle the passing of time through a step function. This involves incrementing the stored current time to the time of the next event, after the previous event is completed. Since nothing is scheduled to happen between those two times, that time can be safely skipped over.

The package must also have code to simulate the popped event happening, using any parameters it is given. Without the ability to specify event parameters, such as frequency and magnitude, the user has no flexibility in using the program and likely will not be able to get the data needed for analysis. The package must also be able to calculate and deliver the desired analysis to the user. In the basketball example, that might be the average and standard deviation of points scored per game. If the package can record the data, but cannot do anything productive with it, its usefulness drops.

There are three main methods used to implement the skeleton of DES. These methods focus on different aspects of the DES model. They are activity-based, event-oriented, and process-oriented. In this report we will

discuss the event-oriented and process-oriented methods.

2 Event-Oriented DES

Often, the simplest way of implementing DES is to make it event-oriented. Many of the possible implementations of different parts of DES discussed in the previous section are used in Event-Oriented DES. We see one full implementation of an Event-Oriented DES skeleton in the DES.R example (see code listing below). As implied by the name, Event-Oriented DES, and by extension DES.R, bases its simulation on handling each event in sequence.

One of the major aspects of Event-Oriented DES is its use of shortcuts in its simulated time-frame. It uses these shortcuts to skip to each new event that it must track. This is where the event list comes in handy. A known event is scheduled and put into the list. When this happens in the course of the simulation is left to the user. After one event finishes, the system moves to the time of the next event. Since we know that the given event is the next event to happen, we know that no events of consequence will occur in the times that we skip. Therefore, it is safe to skip those times. The skip-time algorithm works essentially as follows:

Listing 1: Skip Time Algorithm

```
1 nextEventTime = eventQueue.front.eventTime
2 if(nextEventTime < endSimTime)
3   currentSimTime = nextEventTime
4 else
5   exit()
```

We see another example of this in DES.R. See the code labelled *skipTime* for the exact time-skipping code in the package. Sometimes, we need extra functionality regarding the priority of different kinds of events (such as in a natural disaster simulation). In these cases, we use a priority queue instead of a simple list. DES.R specifically uses a matrix to simulate a queue.

Once the Event-Oriented DES is at the next event time, it must handle the event. In many cases, this may involve multiple events, so the system must have a way of knowing which event is about to occur. DES.R makes a framework for this through its `evnt` type variable, in which each type of

event is given a different identifier. As we see in the code labeled *eventAdd*, this information goes into the event queue along with the event itself. It also uses a user-supplied *reactevent* as the specific event handler (as clearly the handler will need to be changed for specific uses of the library, see *eventHandleReact* code section).

The general algorithm for Event-Oriented DES is simple. While there's still an event scheduled to happen within the time limit, we skip to the next event and call the event handler. As opposed to Process-Oriented DES, this is much easier to implement (avoided threads) and often works faster, especially because most of the simulation time is skipped over.

Listing 2: DES.R Code

```

1  newsim <- function(timelim , maxesize , appcols=NULL, aevntset=FALSE,
   dbg=FALSE) {
2    simlist <- new.env()
3    simlist$currtime <- 0.0 # current simulated time
4    simlist$timelim <- timelim
5    simlist$timelim2 <- 2 * timelim
6    simlist$passedtime <- function(z) z > simlist$timelim
7    simlist$evnts <- matrix(nrow=maxesize , ncol=2+length(appcols))
8    colnames(simlist$evnts) <- c('evnttime', 'evnttype', appcols)
9    simlist$evnts[,1] <- simlist$timelim2
10   simlist$aevntset <- aevntset
11   if (aevntset) {
12     simlist$aevnts <- NULL
13     simlist$nextaevnt <- 1
14   }
15   simlist$dbg <- dbg
16   simlist
17 }
18
19 # eventAdd
20 schedevnt <- function(simlist , evnttime , evnttype , appdata=NULL) {
21   evnt <- c(evnttime , evnttype , appdata)
22   fr <- getfreerow(simlist)
23   simlist$evnts[fr ,] <- evnt
24 }
25 # endEventAdd
26
27 getfreerow <- function(simlist) {
28   evtimes <- simlist$evnts[,1]
29   tmp <- Position(simlist$passedtime , evtimes)
30   if (is.na(tmp)) stop('no room for new event')
31   tmp
32 }
33
34 getnextevnt <- function(simlist) {

```

```

35   etimes <- simlist$evnts[,1]
36   whichnexte <- which.min(etimes)
37   nextetime <- etimes[whichnexte]
38   if (simlist$aevntset) {
39     nextatime <- simlist$aevnts[simlist$nextaevnt,1]
40     if (nextatime < nextetime) {
41       oldrow <- simlist$nextaevnt
42       simlist$nextaevnt <- oldrow + 1
43       return(simlist$aevnts[oldrow,])
44     }
45   }
46   head <- simlist$evnts[whichnexte,]
47   simlist$evnts[whichnexte,1] <- simlist$timelim2
48   return(head)
49 }
50
51 mainloop <- function(simlist) {
52   simtimelim <- simlist$timelim
53   while(TRUE) {
54     # skipTime
55     head <- getnextevnt(simlist)
56     etime <- head['evnttime']
57     if (etime > simlist$timelim) return()
58     simlist$currtime <- etime
59     # endSkipTime
60     # eventHandleReact
61     simlist$reactevent(head, simlist)
62     # endEventHandleReact
63     if (simlist$dbg) {
64       print("event occurred:")
65       print(head)
66       print("events list now")
67       print(simlist$evnts)
68       browser()
69     }
70   }
71 }
72
73 cancelvnt <- function(rownum, simlist) {
74   simlist$evnts[rownum,1] <- simlist$timelim2
75 }
76
77 newqueue <- function(simlist) {
78   if (is.null(simlist$evnts)) stop('no event set')
79   q <- new.env()
80   q$m <- matrix(nrow=0, ncol=ncol(simlist$evnts))
81   q
82 }
83
84 appendfcfs <- function(queue, jobtoqueue) {
85   if (is.null(queue$m)) {

```

```

86     queue$m <- matrix(jobtoqueue,nrow=1)
87     return()
88 }
89 queue$m <- rbind(queue$m,jobtoqueue)
90 }
91
92 delfcfs <- function(queue) {
93     if (is.null(queue$m)) return(NULL)
94     qhead <- queue$m[1,]
95     queue$m <- queue$m[-1,,drop=F]
96     qhead
97 }
98
99 exparrivals <- function(simlist,meaninterarr,batchsize=10000) {
100     if (!simlist$aevntset)
101         stop("newsim() wasn't called with aevntset TRUE")
102     es <- simlist$evnts
103     cn <- colnames(es)
104     if (cn[3] != 'arrvtime') stop('col 3 must be "arrvtime"')
105     if (cn[4] != 'jobnum') stop('col 3 must be "jobnum"')
106     erate <- 1 / meaninterarr
107     s <- 0
108     allarvs <- NULL
109     while(s < simlist$timelim) {
110         arvs <- rexp(batchsize,erate)
111         s <- s + sum(arvs)
112         allarvs <- c(allarvs,arvs)
113     }
114     cuallarvs <- cumsum(allarvs)
115     allarvs <- allarvs[cuallarvs <= simlist$timelim]
116     nallarvs <- length(allarvs)
117     if (nallarvs == 0) stop('no arrivals before timelim')
118     cuallarvs <- cuallarvs[1:nallarvs]
119     maxesize <- nallarvs + nrow(es)
120     newes <- matrix(nrow=maxesize,ncol=ncol(es))
121     nonempty <- 1:nallarvs
122     newes[nonempty,1] <- cuallarvs
123     if (is.null(simlist$arrvevnt)) stop('simlist$arrvevnt undefined')
124     newes[nonempty,2] <- simlist$arrvevnt
125     newes[nonempty,3] <- newes[nonempty,1]
126     newes[nonempty,4] <- 1:nallarvs
127     newes[-nonempty,1] <- simlist$timelim2
128     colnames(newes) <- cn
129     simlist$aevnts <- newes
130 }

```


3 Process-Oriented DES

Process-Oriented DES is exactly what it sounds like, a DES implementation using similar concepts to UNIX-style processes. In Process-Oriented DES, we use a process (often a thread) for each object in the simulation, as well as another one for the handling system. One might alternatively call it “Object-Oriented DES” were the term not already claimed in the programming world. For languages like R, that do not support multithreading, generally each process is a separate invocation of the language.

For a single MM1 queue simulation, a Process-Oriented DES involves four different processes, one for adding to the queue, one for processing the items in the queue, one to manage the simulation, and a main thread to start them all. If there were multiple servers, each one would need its own process. Any Process-Oriented DES implementation will have to have a manager thread to keep track of the simulated time, among other things. In R, this manager thread will be the one that specifically invokes the DES library in use, while the object threads will invoke the simulation code itself.

In dealing with simulated time, Process-Oriented DES works somewhat similarly to Event-Oriented DES. If the event queue is nonempty, the manager will jump to the next time, skipping uneventful time. It will then yield to whichever thread handles that event. If the event queue is empty, however, the manager will sleep until either a) it is nonempty, or b) time runs out.

The Python library *Simpy* is a famous example of Process-Oriented DES. *Simpy* has a clever way of jumping between event and manager processes, which themselves are all held in a *Simpy environment*. To make these jumps, it assigns a generator functions (usually called *Run*) to each of its event threads. Generator functions are iterators that yield back values after each iteration. These are the functions that a *Simpy* program’s main process will use to activate the event processes. This happens by passing the event process’s *Run()* function as an argument to *Simpy*’s *activate()* function.

The generator in an event process will, at a certain point, yield to another process for a specific amount of time. It generally does this through either a time out or by passing a number back to the manager thread via the *yield*. Once that time has passed, the yielded event thread will resume. *Simpy* takes advantage of a generator’s ability to yield to its calling function to get out of the event process and wake up another process. *Simpy* can also

use the generator ability to return values with a yield to help deal with advancing its simulated time. Through these uses of Python generators, Simpy can function as a process-oriented implementation of DES.

4 Rposim Package

Rposim is short for R Process-Oriented Simulation. Its objective is to be a simple implementation of Process-Oriented DES in using the functionality of the bigmemory package in place of threads or generators. Through this construct one is able to simulate a discrete environment by running multiple instances of R at once. One instance will act as a manager, while subsequent instances will represent the various processes in the environment.

In order to get a better overview of how Rposim works, it is beneficial to examine the `newsim` function. This is how one would create and run a new DES simulation. Various information needs to be passed into this function, in order to tell Rposim how to create the manager and the psuedo-threads for the corresponding processes of the simulation.

Firstly, the `newsim` function creates a new manager object by calling the `managerInit` constructor function. This function takes in a vector of process functions and an integer value which represents the time at the end of the simulation. Then it initializes various attributes important to the manager object, such as a vector of the processes, the number of processes, the current simulated time, the maximum simulated time, and a vector of events. These attributes are all vital for keeping track of simulated time or which event is scheduled to happen next.

Once this manager is setup `newsim` creates a bigmemory matrix which will help coordinate the different R instances that are required in order to simulate the various processes. We create this matrix now as it will need to be linked to the pseudo-threads as they are started. We start these threads within the `makeThreads` command. This function uses the `system2` command to make a new thread (in this case a new R instance) for each application process.

The event queue is implemented as a list of `listEntry` objects. Each of these objects includes an event time (the time at which the event will occur) and an event type (represented by an integer). The event time, for

simplicity's sake, is represented by an integer. Events are built and put into the list as they are made (via the `addEvent` function). As time passes, events whose time comes up via the main loop are taken from the list via the `getTimedEvents` function, and then they are sent to the event handler for whatever DES implementation is using Rposim, along with the `results` object from the manager (which will have the objects and stats within altered as necessary by the handler).

Since this package can only deal with time in integer values, it is very possible (even probable in some implementations) for multiple events to be scheduled for the same time. It is for this reason that the `getTimedEvents` function returns a list, and the `run` function handles the list in a loop. This way, we can deal with the currently scheduled events in the same way regardless of how many events are scheduled for the given time.

Since there can be many different types of events within a DES simulation, the event type is also handled by the event handler. This also allows for greater flexibility in how each specific DES simulation is implemented with Rposim. How each type of event in a simulation is represented is up to those implementing it. Generally, a different int for each event works best, as is done in the DES.R examples.

Listing 3: Rposim Code

```

1  library(bigmemory) # used to share memory between the manager and
   the different processes
2
3  # newsim
4  #
5  # Inputs: processVec - vector of application specific processes
6  #         endOfSim - time limit for the simulation
7  #         ncols - based off the number of instances in each
   process, this is the max number
8  #         of cols necessary in the shared matrix
9  #
10 # Creates a new manager object, links the processes to the manager
   , and runs the manager
11
12 newsim <- function(processVec, endOfSim) {
13   # creation of manager, which keeps track of time and waking up
   of each process
14   mgr <- managerInit(processVec, endOfSim)
15
16   # creates shared matrix
17   data <- big.matrix(10, 10, type='integer') # 10x10 big matrix
18   mgr$data <- data
19

```

```

20     # creation of a thread for each item in processVec and manages
      each process
21     id <- 1
22     for (process in processVec) {
23         makeThread(process, id, data)
24         id <- id + 1
25     }
26
27     # now that everything is set up, we will run the simulation
28     # returns a times object, which includes all necessary
      analysis
29     run(mgr)
30 }
31
32 # managerInit
33 #
34 # Inputs: processVec - vector of application specific processes
35 #         endOfSim - time limit for the simulation
36 #
37 # Attributes: processes - vector containing all processes for the
      simulation, these will
38 #               end up being run in different instances
      of R (pseudo-threads)
39 #               numProcesses - total number of processes in the
      simulation
40 #               curTime - current time in the simulation
41 #               maxTime - maximum duration of simulation
42 #               events - vector containing all unreacted events in
      the simulation
43 #
44 # Creates a manager object which controls the simulation - i.e.
      keeps track of current time,
45 # max time, processes, number of processes, and the pending events
      in the simulation
46
47 managerInit <- function(processVec, endOfSim) {
48     me <- list()
49
50     me$processes <- processVec
51     me$numProcesses <- length(processVec)
52     me$curTime <- 0
53     me$maxTime <- endOfSim
54     me$events <- c()
55
56     class(me) <- 'manager'
57     return(me)
58 }
59
60 # makeThread
61 #
62 # Inputs: process - function which will be called in the R

```

```

instance, this is a function
63 #           set-up to listen for the manager as it
    represents a process in the DES
64 #           processID – each process has an id in order to keep
    track of it
65 #           data – the shared matrix, which needs to be linked to
    the process
66 #
67 # Creates a new terminal instance of R calling the appropriate
    process function
68
69 makeThread <- function(process, processID, data) {
70     # open new instance of R
71     system2(command="xterm", args=print("R")) # is there a better
    way to create a new R terminal??
72
73     # need to figure out how to call the process function in that
    new instance of R
74     # this will hopefully be correctly linked with the bigmemory
    matrix
75
76     # one possible approach to this would be to create a .R file
    for each process
77     # and open the xterm with the command Rscript process_n.R
    where the file would
78     # load Rposim.R and the implementation (such as BaristaDES.R)
    and then simply
79     # run the processFlow function for the particular process
80 }
81
82 # newEvent
83 #
84 # Inputs:  time – given time, make event for that time
85 #           eventType – given event type, used when event is pulled
    out of list at execution time
86 # Outputs: me – new event object to add to list
87 #
88 # creates event object for list
89
90 newEvent(time, eventType) {
91     me <- list()
92     me$time <- time
93     me$eventType <- eventType
94     class(me) <- "listEntry"
95     return(me)
96 }
97
98 # addEvent
99 #
100 # Inputs: eventList – mgr$list, list to add new event to
101 #           time – given time, add event for that time

```

```

102 #           eventType – given event type, used when event is pulled
           out of list at execution time
103 #
104 # Outputs: eventList – list which contains new event
105 #
106 # Adds new event to event list
107
108 addEvent <- function(eventList,time,eventType) {
109     newEntry <- newEvent(time,eventType)
110     eventList.append(me)
111     return(eventList)
112 }
113
114 # getTimedEvents
115 #
116 # Inputs: eventList – mgr$list, list of all scheduled events
117 #         time – current time, check for events at time
118 #
119 # Outputs: events – list of events at this time, usually only one
120 #
121 # gets the events scheduled for the current time
122
123 getTimedEvents <- function(eventList,time) {
124     events <- list()
125     for (i in eventList) {
126         if(i$time == time)
127             events.append(i)
128         #remove from eventList after end of function
129     }
130     return(events)
131 }
132
133 # run
134 #
135 # Inputs: mgr – manager object which has all the necessary
           attributes of the simulation
136 #
137 # Outputs: results – list which contains all of the results of the
           simulation
138 #
139 # Runs the simulation until the maxTime is reached or there are no
           more events to be processed
140
141 run <- function(mgr) {
142     results <- list()
143     # set remove condition for events to be triggered
144     removeCondition <- sapply(mgr$events, function(x) x$time !=
mgr$curTime)
145     while (mgr$curTime < mgr$maxTime) {
146         # ...
147         triggeredEvents <- getTimedEvents(mgr$events, mgr$curTime)

```

```

148     mgr$events <- mgr$events[removeCondition]
149     for (i in triggeredEvents){ # loop only relevant if
multiple events at same time
150         # handle event function from sim implementation as well
as stats updating
151         results <- handleEvent(i$eventType, results)
152     }
153     mgr$curTime <- mgr$curTime + 1
154 }
155
156 class(results) <- 'times'
157 return(results)
158 }
159
160 # yield
161 #
162 # Function necessary for applications, this waits for a signal
from the manager that
163 # the process may proceed to the next step in its flowchart
164
165 yield <- function() {
166     while (True) {
167         # figure out how to signal from the manager that the
process may proceed
168         # probably easiest with a boolean variable in the shared
matrix
169     }
170 }

```

5 Simmer Package

Simmer is another R package that has the same basic functionality as Rposim. It is also designed as a Process-Oriented DES package. It is advertised as being like Simpy. However, Simpy uses generators as the backbone of its process handling. R does not have generators, or even multithreading capability. How then does Simmer work as a process-oriented package? As we look through the different ways that Simmer makes Process-Oriented DES work, we will go over an example of the package in use, using a very similar simulation model to MachRep.R. The code will be listed at the end of this section.

There are some parts of Simmer that work effectively in an identical manner to Simpy. First, Simmer encapsulates the entire simulation within a single environment, built by the `simmer()` function (line 12 in the example

code). As we can see in the rest of the code, everything else that we build for the simulation is attached to that environment (simply named *env* in the example). *Simmer* also uses a `run()` function to simulate the passing of time, accepting a number parameter to serve as the stopping point (line 35). Note that we attached this `run` function to the already-built *env* environment, as with everything else built for the simulation. However, though there are similarities between *Simpy* and *Simmer*, there are still bound to be major differences since *Simmer* cannot exploit Python's generators and multithreading capabilities.

There are a few methods that *Simmer* uses to overcome these issues in building its package. The first one is rather simple. It builds its own version of Python generators. Each generator defines the inner workings of one object in each simulation. We can see that in lines 28-29 of our sample code, each generator is hooked up to one of the machines that will be running, along with what time it will start (`at(0)`), how much monitoring will be done (`mon=2`), and the exact function that each machine will perform during the simulation (`runMach(i)`). We build one generator for each object in the loop.

The `runMach` function is the meat of this simulation. It is where all the interesting stuff happens and where all the data is collected. To make these functions work, *Simmer* exploits the concept of a trajectory. A trajectory is a set of actions that are linked together into a chain, using the `%>%` functionality from the *magrittr* package, where they build off each other. By combining this chain of actions with the `rollback` function, we can both tell the machine generator what to do, and to repeat it ad nauseum until the simulated time is up. The `rollback` function takes two numbers. The first tells the trajectory how many commands to roll back, while the second one tells when to repeat that rollback. In our use (line 24), we roll back 6 lines (to the first timeout) and repeat that an infinite number of times. The rollbacks end when the simulated time is up.

The trajectory also contains *Simmer*'s methods of data collection. We decide what data we want to collect in the trajectory and then record it as necessary using attributes. Specifically, we use the `set_attribute` function. In our machine repair, we track two variables, the total up time of a machine, and the number of repairs. At the end of the program, in lines 37-40, we print out the monitored attributes, using the `get_mon_attributes` function. Each generator has its own copies of (and values assigned to) the used attributes. *Simmer* likes to take advantage of the *dplyr* data manipulation

package to format some of its prints.

Listing 4: Simmer Excerpts from MachRep (in order of mention)

```
1 # Line 12: Simmer Environment
2   env <- simmer()
3
4 # Lines 34–35: Runtime for Environment
5   env %>%
6     run(SIM_TIME) %>% invisible
7
8 # Lines 28–29: Machine Generator Loop
9   for (i in machines) env %>%
10     add_generator(i, runMach(i), at(0), mon = 2)
11
12 # Line 24: Rollback
13   rollback(6, Inf)
14
15 # Lines 37–40: Printing Attributes
16   get_mon_attributes(env) %>%
17     dplyr::group_by(name) %>%
18     dplyr::slice(n()) %>%
19     dplyr::arrange(name)
20
21 # Line 19: Seize Repairman Resource
22   seize("repairman", 1) %>%
23
24 # Line 21: Release Repairman Resource
25   release("repairman", 1) %>%
```

Listing 5: Full Simmer MachRep Implementation

```

1  library(simmer)
2
3  MTTF <- 300.0          # Mean time to failure in minutes
4  BREAK_MEAN <- 1 / MTTF # Param. for exponential distribution
5  REPAIR_TIME <- 30.0    # Time it takes to repair a machine in
                           minutes
6  JOB_DURATION <- 30.0   # Duration of other jobs in minutes
7  NUM_MACHINES <- 10     # Number of machines in the machine shop
8  SIM_TIME <- 50000      # Simulation time in stuff
9
10 # setup
11 set.seed(42)
12 env <- simmer()
13
14 runMach <- function(machine)
15   trajectory() %>%
16     set_attribute("upTime", 0) %>%
17     set_attribute("repairs", 0) %>%
18     timeout(function() rexp(1, BREAK_MEAN)) %>%
19     seize("repairman", 1) %>%
20     timeout(REPAIR_TIME) %>%
21     release("repairman", 1) %>%
22     set_attribute("repairs", 1, mod="+") %>%
23     set_attribute("upTime", (function() now(env) - get_attribute
24       (env, "upTime") - get_attribute(env, "repairs") * REPAIR_TIME), mod
25       = "+") %>%
26     rollback(6, Inf) # go to 'timeout' over and over
27
28 machines <- paste0("machine", 1:NUM_MACHINES-1)
29
30 for (i in machines) env %>%
31   add_generator(i, runMach(i), at(0), mon = 2)
32
33 env %>%
34   add_resource("repairman", 1, Inf, preemptive = TRUE) %>%
35   invisible
36
37 env %>%
38   run(SIM_TIME) %>% invisible
39
40 get_mon_attributes(env) %>%
41   dplyr::group_by(name) %>%
42   dplyr::slice(n()) %>%
43   dplyr::arrange(name)

```

6 Who did What?

Incomplete