ECS 145 Term Project Report
DES implementations in R and Python

PETER VAN AUSDELN, HENRY JUE, NIKLAS KRAEMER
University of California, Davis
March 22, 2018

Contents

Listings

1 Introduction: What is DES?

Discrete Event Simulation (DES) is a way of modeling 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 changing continuously.

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 interval 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 a DES system. Each method focuses on a different aspect of the DES model. These aspects are either activity-based, event-oriented, or process-oriented. In this report we will discuss and analyze the similarities and differences of the event-oriented and process-oriented methods. A brief look at popular DES implementations as well as a process-oriented implementation are also presented.

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 Listing 2). 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

```
nextEventTime = eventQueue.front.eventTime
if(nextEventTime < endSimTime)
   currentSimTime = nextEventTime
else
   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 `evnttype` 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 is 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

```
newsim <- function(timelim,maxesize,appcols=NULL,aevntset=FALSE,
   dbg=FALSE) {
   simlist <- new.env()
   simlist$currtime <- 0.0  # current simulated time
   simlist$timelim <- timelim
   simlist$timelim2 <- 2 * timelim
   simlist$passedtime <- function(z) z > simlist$timelim
   simlist$evnts <- matrix(nrow=maxesize,ncol=2+length(appcols))
   colnames(simlist$evnts) <- c('evnttime','evnttype',appcols)
   simlist$evnts[,1] <- simlist$timelim2
   simlist$aevntset <- aevntset
   if (aevntset) {
      simlist$aevnts <- NULL
      simlist$nextaevnt <- 1
   }
   simlist$dbg <- dbg
   simlist
}

# eventAdd
schedevnt <- function(simlist,evnttime,evnttype,appdata=NULL) {
   evnt <- c(evnttime,evnttype,appdata)
   fr <- getfreerow(simlist)
   simlist$evnts[fr,] <- evnt
}
# endEventAdd

getfreerow <- function(simlist) {
   evtimes <- simlist$evnts[,1]
   tmp <- Position(simlist$passedtime,evtimes)
   if (is.na(tmp)) stop('no room for new event')
   tmp
```

```
}

getnextevnt <- function(simlist) {
   etimes <- simlist$evnts[,1]
   whichnexte <- which.min(etimes)
   nextetime <- etimes[whichnexte]
   if (simlist$aevntset) {
      nextatime <- simlist$aevnts[simlist$nextaevnt,1]
      if (nextatime < nextetime) {
         oldrow <- simlist$nextaevnt
         simlist$nextaevnt <- oldrow + 1
         return(simlist$aevnts[oldrow,])
      }
   }
   head <- simlist$evnts[whichnexte,]
   simlist$evnts[whichnexte,1] <- simlist$timelim2
   return(head)
}

mainloop <- function(simlist) {
   simtimelim <- simlist$timelim
   while(TRUE) {
      # skipTime
      head <- getnextevnt(simlist)
      etime <- head['evnttime']
      if (etime > simlist$timelim) return()
      simlist$currtime <- etime
      # endSkipTime
      # eventHandleReact
      simlist$reactevent(head,simlist)
      # endEventHandleReact
      if (simlist$dbg) {
         print("event occurred:")
         print(head)
         print("events list now")
         print(simlist$evnts)
         browser()
      }
   }
}

cancelevnt <- function(rownum,simlist) {
   simlist$evnts[rownum,1] <- simlist$timelim2
}

newqueue <- function(simlist) {
   if (is.null(simlist$evnts)) stop('no event set')
   q <- new.env()
   q$m <- matrix(nrow=0,ncol=ncol(simlist$evnts))
   q
}
```

```
appendfcfs <- function(queue,jobtoqueue) {
   if (is.null(queue$m)) {
      queue$m <- matrix(jobtoqueue,nrow=1)
      return()
   }
   queue$m <- rbind(queue$m,jobtoqueue)
}

delfcfs <- function(queue) {
   if (is.null(queue$m)) return(NULL)
   qhead <- queue$m[1,]
   queue$m <- queue$m[-1,,drop=F]
   qhead
}

exparrivals <- function(simlist,meaninterarr,batchsize=10000) {
   if (!simlist$aevntset)
      stop("newsim() wasn't called with aevntset TRUE")
   es <- simlist$evnts
   cn <- colnames(es)
   if (cn[3] != 'arrvtime') stop('col 3 must be "arrvtime"')
   if (cn[4] != 'jobnum') stop('col 3 must be "jobnum"')
   erate <- 1 / meaninterarr
   s <- 0
   allarvs <- NULL
   while(s < simlist$timelim) {
      arvs <- rexp(batchsize,erate)
      s <- s + sum(arvs)
      allarvs <- c(allarvs,arvs)
   }
   cuallarvs <- cumsum(allarvs)
   allarvs <- allarvs[cuallarvs <= simlist$timelim]
   nallarvs <- length(allarvs)
   if (nallarvs == 0) stop('no arrivals before timelim')
   cuallarvs <- cuallarvs[1:nallarvs]
   maxesize <- nallarvs + nrow(es)
   newes <- matrix(nrow=maxesize,ncol=ncol(es))
   nonempty <- 1:nallarvs
   newes[nonempty,1] <- cuallarvs
   if (is.null(simlist$arrvevnt)) stop('simlist$arrvevnt undefined
    ')
   newes[nonempty,2] <- simlist$arrvevnt
   newes[nonempty,3] <- newes[nonempty,1]
   newes[nonempty,4] <- 1:nallarvs
   newes[-nonempty,1] <- simlist$timelim2
   colnames(newes) <- cn
   simlist$aevnts <- newes
}
```

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 functinality 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 tha `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 integer for each event works best, as is done in the DES.R examples.

Given this brief overview of `Rposim`'s core functionality and implementation, there are various aspects which require more in-depth attention. Of foremost interest is how `Rposim` is able to overcome R's lack of generators in implementing a Process-Oriented DES system. Generally, this would be a simple issue of defining threads for the different processes in the system, yet R also lacks any native threading capabilities. Thus, in order to implement `Rposim`'s desired functionality some creativity is necessary.

The simplest way of forgoing R's lack of generators and, furthmore, R's lack of threading, was to implement pseudo-threads by running multiple instances of R concurrently. This would allow one instance of R to function as the manager for the system while subsequent instances can run code simulating the various processes. This, however, is not as simple as it seems and thus represented the chief challenge of `Rposim`'s implementation.

While using R to create new instances of R is rather simple (using the `system2` function) it is a far greater challenge to setup these instances so that they are running the desired functions. Not only that, the new R instances must also be connected to the manager and each other. While `bigmemory` is a nice R package that offers data sharing between different R instances, actually linking this shared `bigmemory` matrix to the different pseudo-threads is very challenging.

Having invoked new R instances there are two options for how to link them with the manager. The simplest option is to make the user of `Rposim` responsible for this linkage. This however would require the user to understand the underlying implementation of `Rposim` and could thus prove an annoyance and disuade any potential `Rposim` users. This means the second option has to be the simpler from the user's point of view. This option calls for `Rposim` to handle the linking automatically. This is done by having `Rposim` generate R code in these new R instances. This code would need to load the shared `bigmemory` matrix from the harddrive. Once the matrix is successfully loaded into the process R instance, there is no further worry about syncing the matrix between the R instances, as this is handled automatically by the `bigmemory` package.

The `bigmemory` package is one way of circumventing R's limitations in generators and threading. Another potentially viable option would be to use TCP and IP protocols. In this potential implementation, machines on any possible internet connected networks could join to run the simulation. One machine would act as the manager, while subsequent machines would simulate the processes in the DES. All coordination and information sharing would be handled over the internet, by sending information between the comupters of the network. This of course poses various pros and cons.

Namely the potential amount of traffic necessary to simulate a complex system could cause severe network congestion causing the implementation to lag and run slow. The simulation would also need to be strictly syncronized between the computers on the network, which would require a lot of additional code that in principle has nothing to do with discrete event simulation.

Potential pros of this TCP/IP approach would be increased collaboration possibilities, as anybody with an internet connected computer would be able to join the simulation. Furthermore, the results of the simulation could be shared and stored in multiple locations with greater ease. This could allow different locations which partook in the simulation to analyze the resulting data differently, without any interference. A live analysis of the data could also be done by a separate computer connected to the simulation, as a secondary manager computer could provide live data analysis.

Listing 3: Rposim Code

```
library(bigmemory) # used to share memory between the manager and
    the different processes

# newsim
#
# Inputs: processFun - vector of application specific processes
#         processObj - vector of objects necessary for the
    processes, i.e. machines/repairmen
#         endOfSim - time limit for the simulation
#         ncols - based off the number of instances in each
    process, this is the max number
#                 of cols necessary in the shared matrix
#
# Creates a new manager object, links the processes to the manager
    , and runs the manager

newsim <- function(processFun, processObj, endOfSim) {
    # creation of manager, which keeps track of time and waking up
     of each process
    mgr <- managerInit(processFun, endOfSim)

    # creates shared matrix
    # size based off number of processes and max number of objects
     in processes
    nrows <- length(processFun)
    ncols <- length(apply(as.matrix(processObj), 1, length))
    data <- big.matrix(nrows, ncols, type='integer', shared=T)
    mgr$data <- data

    # creation of a thread for each item in processFun and manages
     each process
    id <- 1
    for (process in processFun) {
        makeThread(process, id, data)
        id <- id + 1
    }

    # now that everything is set up, we will run the simulation
    # returns a times object, which includes all necessary
    analysis
    run(mgr)
}

# managerInit
#
# Inputs: processFun - vector of application specific processes
#         endOfSim - time limit for the simulation
#
# Attributes: processes - vector containing all processes for the
    simulation, these will
```

```
#                          end up being run in different instances
    of R (pseudo-threads)
#              numProcesses - total number of processes in the
    simulation
#              curTime - current time in the simulation
#              maxTime - maximum duration of simulation
#              events - vector containing all unreacted events in
    the simulation
#
# Creates a manager object which controls the simulation - i.e.
    keeps track of current time,
# max time, processes, number of processes, and the pending events
     in the simulation

managerInit <- function(processFun, endOfSim) {
    me <- list()

    me$processes <- processFun
    me$numProcesses <- length(processFun)
    me$curTime <- 0
    me$maxTime <- endOfSim
    me$events <- c()

    class(me) <- 'manager'
    return(me)
}

# makeThread
#
# Inputs: process - function which will be called in the R
    instance, this is a function
#                   set-up to listen for the manager as it
    represents a process in the DES
#         processID - each process has an id in order to keep
    track of it
#         data - the shared matrix, which needs to be linked to
    the process
#
# Creates a new terminal instance of R calling the appropriate
    process function

makeThread <- function(process, processID, data) {
    # open new instance of R
    me <- list()

    me$id <- processID
    me$data <- data

    system2(command="R", args="--vanilla", wait=F) # is there a
    better way to create a new R terminal??

```

```
    # need to figure out how to call the process function in that
    new instance of R
    # this will hopefully be correctly linked with the bigmemory
    matrix

    class(me) <- 'thread'
    return(me)
}

# newEvent
#
# Inputs:  time - given time, make event for that time
#          eventType - given event type, used when event is pulled
    out of list at execution time
# Outputs: me - new event object to add to list
#
# creates event object for list

newEvent <- function(time, eventType) {
   me <- list()
   me$time <- time
   me$eventType <- eventType
   class(me) <- "listEntry"
   return(me)
}

# addEvent
#
# Inputs:  eventList - mgr$list, list to add new event to
#          time - given time, add event for that time
#          eventType - given event type, used when event is pulled
    out of list at execution time
# Outputs: eventList - list which contains new event
#
# Adds new event to event list

addEvent <- function(eventList,time,eventType) {
   newEntry <- newEvent(time,eventType)
   eventList.append(me)
   return(eventList)
}

# getTimedEvents
#
# Inputs:  eventList - mgr$list, list of all scheduled events
#          time - current time, check for events at time
# Outputs: events - list of events at this time, usually only one
#
# gets the events scheduled for the current time

getTimedEvents <- function(eventList,time) {
```

```
    events <- list()
    for (i in eventList) {
        if(i$time == time)
            events.append(i)
            #remove from eventList after end of function
    }
    return(events)
}

# run
#
# Inputs:  mgr - manager object which has all the neccessary attributes of the simulation
# Outputs: results - list which contains all of the results of the simulation
#
# Runs the simulation until the maxTime is reached or there are no more events to be processed

run <- function(mgr) {
     results <- list()
    # set remove condition for events to be triggered
     removeCondition <- sapply(mgr$events, function(x) x$time != mgr$curTime)
     while (mgr$curTime < mgr$maxTime) {
         # ...
       triggeredEvents <- getTimedEvents(mgr$events, mgr$curTime)
       mgr$events <- mgr$events[removeCondition]
         for (i in triggeredEvents){ # loop only relevant if multiple events at same time
            # handle event function from sim implementation as well as stats updating
            results <- handleEvent(i$eventType, results)
       }
         mgr$curTime <- mgr$curTime + 1
     }

     class(results) <- 'times'
     return(results)
}

# yield
#
# Function necessary for applications, this waits for a signal from the manager that
# the process may proceed to the next step in its flowchart

yield <- function() {
     while (T) {
         # figure out how to signal from the manager that the process may proceed
```

```
        # probably easiest with a boolean variable in the shared
    matrix
    }
}
```

This overview and analysis of the underlying challenges of `Rposim` allow us to examine a potential user application of the package. With the code in Listing 4, BaristaDES.R implements a simulation of costumers visiting a coffee shop. Here there are a number of baristas waiting to take and make various drink orders. This simulation could be beneficial in analyzing staffing requirements at a café or examining peak operational hours for different café locations.

Here the user simply imports the `Rposim` package using the `source` function. Then the user defines variables for the simulation, such as maximum time, average event time durations, as well as vectors for the baristas and customers, where the length of the vector represents the number of each. These vectors have integers, which represent the current state of the baristas or customers in their process flow. Then the user must implement functions which describe the process of a barista or customer. This entails the barista waiting for an order by a customer, making the order, and then cleaning the machine. Implementing these functions requires minimal understanding of `Rposim`'s underlying implementation, which is a big advantage of the package. Finally the user must run the simulation, which is done by passing the flow functions into the `newsim` function. From here the user can save the results of the simulation and proceed with any data analysis desired.

Listing 4: Barista Rposim Implementation

```
# This is a sample application of Rposim's Process-Oriented DES
#
# These are just the functions that need to be defined by the user
    . Rposim
# will automatically set up the three necessary instances of R to
    successfully
# run the simulation. One instance will act as the manager, while
    a second
# and third will act as the baristas and the customers,
    respectively.
# This is the case as the processes need to be actively listening
    and
# interacting with the manager throughout the simulation.
#
# Instructions for running simulation:
#    1. Open a new terminal window and invoke R
```

```
#    2. Load Rposim.R
#    3. Load BaristaDES.R
#       a. this file contains executable code, which will be called
         upon load
#    4. This will automatically launch new R instances, calling
        each
#       process function (thus setting up the listening)
#    5. The simulation will be run
#    6. Corresponding information will be displayed

source('Rposim.R')

# Here there are two processes: (1) the Barista making the coffee
#                               (2) the customers placing the
        orders
#
# These are described in a continuous flowchart (user defined
        functions)

# initialize simulation variables
maxTime <- 28800 # 28,800 seconds in 8 hours, or a typical cafe
        opening duration

baristas <- c(1, 1)               # represents 2 baristas  and
        their initial states
customers <- c(1, 1, 1, 1, 1, 1) # represents 6 customers and
        their initial states

# average times for the events in the barista and customer
        processes
DECISION_TIME <- 120 # time it takes the customer to decide upon
        an order
ORDER_TIME <- 80      # time it takes to place an order
DRINK_PREP <- 200     # represents the amount of time the barista
        needs to make the drink
MACH_CLEAN <- 45      # time it takes the barista to clean the
        machine

# Flowchart to describe the process of a barista, as follows:
#    1. Wait for order from customer
#    2. Recieve order and make coffee
#    3. Clean machine
# This cycle repeats until the end of the simulation
#
# There can be any number of baristas working, these are
        represented
# by a vector of size n, which stores either a 1, 2 or 3 for the
        current
# state the barista is in
#
# As the DES progresses the manager will tell the baristaFlow
```

```
process when
# to update, thus allowing any number of baristas to move to the next step
# in their flowchart

baristaFlow <- function(baristas) {
  # while currentTime < timeLimit
    # barista is currently waiting for an order
    # yield() - listen for an update from the manager aka an order
    # addEvent() - take the order and make the drink
    # addEvent() - clean the machine
}


# Flowchart to describe the process of a customer, as follows:
#   1. Read the menu and decide
#   2. Place coffee order
# This cycle repeats until the end of the simulation
#
# There can be any number of customers at the cafe, they are represented
# by a vector of size n, which stores either a 1 or 2 for the current
# state the customer is in
#
# As the DES progresses the manager will tell the customerFlow process when
# to update, thus allowing any number of customers to move to the next step
# in their flowchart

customerFlow <- function(customers) {
  # while currentTime < timeLimit
    # addEvent() - read menu and decide on an order
    # yield() - listen for an update from the manager
    # addEvent() - place order
}

# run simulation
times <- newsim(c(baristaFlow, customerFlow), c(baristas, customers), maxTime)
```

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 full code is provided in Listing 6.

Listing 5: Excerpts from Simmer MachRep (in order of mention)

```
# Line 12: Simmer Environment
    env <- simmer()

# Lines 34-35: Runtime for Environment
    env %>%
        run(SIM_TIME) %>% invisible

# Lines 28-29: Machine Generator Loop
    for (i in machines) env %>%
        add_generator(i, runMach(i), at(0), mon = 2)

# Line 24: Rollback
    rollback(6, Inf)

# Lines 37-40: Printing Attributes
    get_mon_attributes(env) %>%
        dplyr::group_by(name) %>%
        dplyr::slice(n()) %>%
        dplyr::arrange(name)

# Line 19: Seize Repairman Resource
    seize("repairman",1) %>%

# Line 21: Release Repairman Resource
    release("repairman",1) %>%
```

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 6: MachRep Simmer Implementation

```r
library(simmer)

MTTF <- 300.0              # Mean time to failure in minutes
BREAK_MEAN <- 1 / MTTF     # Param. for exponential distribution
REPAIR_TIME <- 30.0        # Time it takes to repair a machine in minutes
JOB_DURATION <- 30.0       # Duration of other jobs in minutes
NUM_MACHINES <- 10         # Number of machines in the machine shop
SIM_TIME <- 50000          # Simulation time in stuff

# setup
set.seed(42)
env <- simmer()

runMach <- function(machine)
    trajectory() %>%
        set_attribute("upTime", 0) %>%
        set_attribute("repairs",0) %>%
        timeout(function() rexp(1, BREAK_MEAN)) %>%
        seize("repairman",1) %>%
        timeout(REPAIR_TIME) %>%
        release("repairman",1) %>%
        set_attribute("repairs", 1, mod="+") %>%
        set_attribute("upTime", (function() now(env)-get_attribute(env, "upTime")-get_attribute(env,"repairs")*REPAIR_TIME), mod="+") %>%
        rollback(6, Inf) # go to 'timeout' over and over

machines <- paste0("machine", 1:NUM_MACHINES-1)

for (i in machines) env %>%
    add_generator(i, runMach(i), at(0), mon = 2)

env %>%
    add_resource("repairman", 1, Inf, preemptive = TRUE) %>% invisible

env %>%
    run(SIM_TIME) %>% invisible

get_mon_attributes(env) %>%
    dplyr::group_by(name) %>%
    dplyr::slice(n()) %>%
    dplyr::arrange(name)
```

6 Who did What?

The three authors combined to work on the `Rposim` implementation. Considerable efforts were made in the beginning to understand the challenges of its implementation, as well as achieving a high level understanding of how this could be achieved. It is regretable that the authors failed to complete a working implementation of `Rposim`, as they were greatly satisfied having figured out how to implement it on a high-level.

The research for the report, as well as writing implementations for the `Simmer` and `Rposim` packages were divided up fairly between the authors. All contributed to writing and editing the final report. Thus the entirety of the project was a collective group effort.