# Discrete Event Simulation in R

Joseph Shaw, Matt Medina, Dylon Dickinson

March 22, 2018

**Contents**

**What is DES?:** DES is discrete event simulation. The events that are being simulated are discontinous or discrete. We model real world situations of events on a smaller scale so that we can make a blueprint of what will happen in the actual model. There are three general approaches to doing discrete event simulation, which are Activity, Event, and Process Oriented DES. All three of these approaches use a queue to store either a list of the events happening or processes. They also all need to keep track of the environment time so that each event can be simulated in a certain amount of time. This time is also needed in order to tell when certain events are going to be happening, and for when a resource is available for something to use. We will look at two of them, Event-Oriented and Process Oriented DES. While they do the same thing, the two different methods of solving how to do DES are very different.

**Event-Oriented DES:** The DES package for R uses an event-oriented DES. The events are stored in an event list component contained within a R environment named simlist. Use of an R environment variable allows for simlist components to be altered through use of functions, rather than having to reassign them. The event list in the DES package is represented as a matrix that stores 1 event per row, with the first column being the event time and the second column being the event type. The user has the capability of appending more user defined columns for other event purposes that may be useful for the discrete event simulation. One of the great things about the event list in the DES.r package is that the events don't need to be sorted in time ascending order, even though the events must be processed in time ascending order. This is currently the default for event processing in the DES package, which makes use of R's which.min() function to allow for processing the next scheduled event, without the significant performance cost of sorting. The DES package for R also has an option for pre-generating arrival events, which makes use of a time-ordered list of arrivals.

Each event type is signified by the use of an R numeric, which is crucial for the purposes of distinguishing between events so that they may be handled accordingly. This handling of event types is made available through use of a user supplied react() function in the application code. This react() function signifies how event types are processed, and just as importantly, how new events are to be created.

The global time is logged and updated through the simlist component currtime. Among other components for simlist is the timelim and dbg components. The former is used to store the maximum simulated time, whereas the latter is a nifty tool for debugging that allows for single stepping and on-screen event printing.

To setup application code that makes use of DES.r, the user must setup an instance of DES.r with any application-specific components needed in simlist. The user must also setup the react() function to handle and create new events, as mentioned above. The user must then initialize the first events to take place in the simulation. The initial events are then used in the mainloop provided in the DES package, so that they can process and trigger the next events to be created in an iterative fashion. This iterative process continues processing and creating new events until the timelim is reached.

**Process-Oriented DES:** Process-Oriented descrete event simulation differs from Event-Oriented discrete event simulation in that it uses threading in order to allow multiple processes to interact at the same time. Processes can be in different states, such as running(or active), ready, idle, or terminated. The simulation will run until each process has terminated, with each process given a certain amount of time that it needs to run for. Like for Even-Oriented DES, we use an Queue to hold the events, but the events are generally tracked as time, and the processes will run until they receive a command to hold, idle, or terminate. An example of this comes from Simpy, the python package for DES.

With Simpy, we make a class that takes in the process class of simpy in order to be able to run the process. This makes it a subclass of the process class of simpy, which needs to be initialized in order to be run. This also requires us to make a run function for the simulation class to activate. This run function is in fact a

generator, which is used to tell the OS when to allow the subprocesses to run, remain idle, or terminate, which typically returns a tuple with the amount of time. This generator allows for Simpy to be able to do multi-threaded processes during its simulation. Each instance of a class that has the run function within it will basically be a thread, while the os itself is another thread. The operations for yield are hold, release, passivate, and request. Hold will tell the simulation to stop running until the environment time his the hold time, request makes a process get in line to use a resource that is needed, release allows the next process to use a resource, and passivate will make the process wait until another process awakens it.

# The Rposim Package:
## 4.1 Explanation of the package

Our Rposim package attemps to be similar to Simpy, however without generators or threading the task is a bit difficult. We opted to use bigmemory to store our global variables into a matrix which allows us to create multiple instances of R and allows a shared data location in memory for each instance to access and write data to. Because of these issues, we require users of the package to use specific commands in their applications. They need to reference the bigmemory that is going to be used in the simulation so that the globals can be writted, otherwise we get race conditions that make the program run forever.

We also require users to use an S4 class for their application, so that we can retrieve the process id for the class. These process ids will be stored in bigmemory and will be used by the library in order to do certain yield the requests that the user may have. This is how we make up for the lack of generators in R. Each process id has a vector of values which tells the simulation whether it's running or not, whether theres a hold or not, and the time of the next event in the simulation. While we do have these restrictions, the rest of the code is general and up to the user to create.

As stated before though, we tried making our syntax look as similar as possible to simpy so that there would be less issues for those that wanted to port their code to R from simpy. Our functions are in the form of yield_hold, yield_request, yield_release, yield_passivate, now, cancel, reactivate, initialize, activate, and simulate. Of which initialize, activate, and simulate are required in order to run any simulation. Their arguments are very close to what simpy has, with exception to activate, which takes the class instead of the class and a function. One of the race conditions that was showing up was coming from trying to insert the function, so we decided to just take the class.

## 4.2 Differences across different DES methods and TCP/IP method

If we compare our pacakge to the DES.R package, there is a huge difference right off the bat in that the DES.R package is event-oriented DES while our package is process-oriented DES. We use processes to simulate the events, rather than having to create an event list to keep track of the events. We also don't use an R environment, we use global variables to keep track of time and all of the data needed for operations within threads. We don't initialize the first events, rather we allow the Rposim package to do the full simulation without the need to initialize the first events.

Comparing our package to Simpy, we notice that simpy can use generators while we cannot, we have to make our own methods that handle the yield commands that simpy has. As stated in the previous section, we modeled our package to be similar to Simpy. However, making a 1:1 port is impossible due to the restrictions of R. This led us to have to use bigmemory in order to solve the issue of having no generators and not being able to do parallel programming.

Now to compare the package that should be most similar to our Rposim package, Simmer. Simmer mainly uses an environment, trajectories, resources, and library made generators to run its simulation, while we rely soley on global variables and our own functions to simulate generators. Simmer uses a unique way of adding arrival events to try and simulate a generator to update the environment time. We just use Bigmemory to update our global variable of simulation time so that each simulation knows when to stop. Also as stated before, if event arrivals stop in simmer, the simulation stops, while in our package, we are automatically creating packages from the user defined function until the simulation time is done.

We could use a TCP/IP approach in order to solve the issue of R having no threads, instead of shared-memory. This would involve using a server that would hold all the data and resources that the processes need and the processes would be clients that connect to the server. We could then have the server run the simulation for each process and easily receive codes for when to hold the simulation, and when to passivate and reactivate them. One of the processes would be the server that would send the server the operations that each of the other processes would need to be done on them. When the simulation is over, the os would then pull the data that is needed and send it to the application that is calling the library.

**The Simmer Package:** The Simmer package in R is a process-oriented discrete event simulator aimed at providing everything SimPy has to offer. The current simulated time is stored in the environment and can be accessed through the use of Simmer's now() function. Trajectories are much like the Run() function in SimPy, in the sense that they allow for the environment to set or update its attributes. When setting an attribute within a trajectory, the global parameter must be set to 'TRUE' for the attribute to show up in the monitored attributes list within the simmer environment, once the simulation is complete. The trajectory will only run for as many arrival events as you have, or until it reaches the maximum simulated time.

The seize and release functions built into Simmer provide the same functionality as yield request and yield release in SimPy; they remove and add back a resource respectively. These two functions are used within a trajectory, to keep track of which resources are (or are not) available. For more tedious simulated environment setups, multiple trajectories can be used in conjunction with one another to utilize the same resources. You can do this, although the use of more trajectories can make the simulation harder to follow.

Generators in Simmer are modeled after the generators provided in SimPy, although the fact that R doesn't provide functionality for threading makes them different. To compensate for the lack of threading in R, Simmer's built in generators use an arrival event in conjunction with its timeout function. This provides the functionality of a co-routine, which is the hallmark of generators in SimPy, and hence the "secret sauce" of Simmer.

## Contributions:

Write-ups:

DES - Joseph Shaw
Event-Oriented DES - Matt Medina
Process-Oriented Des - Joseph Shaw
The Rposim Package - Dylon Dickinson, Joseph Shaw
The Simmer Package - Matt Medina

Code:

1. Rposim - Dylon Dickinson, Joseph Shaw
   1.1 RposimMachRep1.r - Dylon Dickinson, Joseph Shaw
   1.2 RposimMachRep2.r - Dylon Dickinson, Joseph Shaw
   1.3 RposimMachRep3.r - Dylon Dickinson, Joseph Shaw
2. SimmerMachRep1.r - Matt Medina, Joseph Shaw

## Appendix:

Rposim.r

```
    library(bigmemory)
    library(methods)

    #globals
    pQ <- c()
    fQ <- c()
    data <- big.matrix(1,3,init=0.0)
    numProcs <- 0

    setRefClass("Process",
    fields=list(pid="numeric"),
    methods = list(
    initialize = function()
    {
        .self$pid = 0
    }))

    setRefClass("Resource",
    fields=list(n="numeric"))

    # Initialize
    initialize <- function(g=c())
    {
    #print("top of init")
        #double the number of rows in data
        data <<- as.big.matrix(rbind(as.matrix(data),matrix(0,nrow=dim(data)[1],ncol=3)))
        dput(describe(data),file="memPtr")

        if(length(g) > 100 || length(g) == 0)
        {
        }else
        {
            globals <<- as.big.matrix(matrix(g,nrow=1,ncol=length(g)))
            dput(describe(globals),file="globals")
        }
        #print("end of init")
    }

    # Used to indicate how much time to allocate to a thread
    yield_hold <- function(p, holdTime)
    {
        #print("in yield")
        if(p$pid == 0) {p$pid <- numProcs + 1}
        if(p$pid > dim(data)[1]) { initialize() }
        data[p$pid,2] <<- 1
        data[p$pid,3] <<- data[1,2] + holdTime
        data[p$pid,1] <<- 1

        while(data[p$pid,1] == 1) {data <- attach.big.matrix(dget('memPtr'))}

        #print("end of yield")
```

```
}

# Cause a thread to join a queue for a given resource and use if no other threads in queue
yield_request <- function(resource)
{
    rQ = c(rQ,p$pid)
}

# Indicate that a thread is done using a resource allowing next thread to use the resource (Need bi
yield_release <- function(resource)
{

}

# Have a thread wait until awakened by some other thread.
yield_passivate <-function()
{

}

# Mark a thread runnable when first created
activate <- function(p)
{
    #print("in activate")
    #print(p$pid)
    numProcs <<- numProcs + 1
    if(p$pid == 0) {p$pid <- numProcs + 1}
    if(p$pid > dim(data)[1])
    {
        initialize()
    }
    pQ <- c(pQ, p)
    save.image("env.RData")
    save(p,file="saveP.RData")

    fileConn<-file("scr.R")
    writeLines(c("load('env.RData')","load('saveP.RData')","library(methods)","library(bigmemory)",'
    close(fileConn)

    system("R CMD BATCH scr.R out.txt &")

    #print("after activate")
}

# Awakens a previously-passivated thread.
reactivate <- function()
{

}

# Cancels all the events associated with a previously-passivated thread.
cancel <- function()
{
```

```
}

now <- function()
{
    #print("start of now")
    data[1,2]  #current time
}

# Do the simulation
simulate <- function(until)
{
#print("in simulate")
    while(data[1,2] < until)
    {
        m <- as.matrix(data)
        minP <- which.min(subset(m, m[,1] == 1)[,3]) #get waiting proc with min time

        if(length(minP) != 0)
        {

            minP <- minP + 2
            data[1,2] <- data[minP,3]  #update total time

            #print(data)

            #print("minp")
            #print(minP)

            #print(data[minP,])
        if(data[minP,2] == 1)
        {
            data[minP,1] <- 0
        } else if(data[minP,2] == 2)
        {
            #handle yield_request
        } else if(data[minP,2] == 3)
        {
            #handle yield_release
        } else if(data[minP,2] == 4)
        {
            #handle yield_passivate
        } else if(data[minP,2] == 5)
        {
            #handle reactivate???
        }
    }
    #print("bottom of sim loop")
}
#print("end of simulate")
}
```

RposimMachRep1.r

```
    source("Rposim.R")


    #global variables
    #UpRate = 1/1.0 # reciprocal of mean up time
    #RepairRate = 1/0.5 # reciprocal of mean repair time
    #NextID = 0 # next available ID number for MachineClass objects
    #TotalUpTime = 0.0 # total up time for all machines

    globals <- NULL

    MachineClass <- setRefClass("MachineClass",
    fields = list(StartUpTime="numeric",ID="numeric"),
    contains = "Process",
    methods = list(
    initialize = function()
    {
        .self$StartUpTime <<- 0.0
        .self$ID <<- globals[1,3]
        globals[1,3] <<- globals[1,3] + 1
        callSuper()
    }
    Run = function()
    {
            while(1)
        {
        print("top of machine loop")
            # record current time, now(), so can see how long machine is up
            .self$StartUpTime <- now()
            # hold for exponentially distributed up time
            UpTime <- rexp(1,globals[1,1])
            yield_hold(.self, UpTime) # simulate UpTime
            globals[1,4] <<- globals[1,4] + now() - .self$StartUpTime
            RepairTime <- rexp(1,globals[1,2])
            # hold for exponentially distributed repair time
            yield_hold(.self, RepairTime)
            #print MachineClass.TotalUpTime
            print("bottom of machine loop")
        }
    }))
    main <- function()
    {
    print("top of main")
        initialize(c(1/1.0, 1/0.5, 0, 0.0))
        # set up the two machine threads
        for(i in 1:2)
        {
            # create a MachineClass object
            M <- MachineClass()
            activate(M) # required
        }
        # run until simulated time 10000
```

```
    MaxSimtime = 10000.0
    simulate(MaxSimtime) # required

    print(paste("the percentage of up time was ", TotalUpTime/(2*MaxSimtime)))
}

main()
```

RposimMachRep2.r

```
    source("Rposim.R")

    #global variables
    UpRate = 1/1.0 # reciprocal of mean up time
    RepairRate = 1/0.5 # reciprocal of mean repair time
    NextID = 0 # next available ID number for MachineClass objects
    TotalUpTime = 0.0 # total up time for all machines
    # RepairPerson = resource(1)
    # NRep <<- 0
    # NImmedRep <<- 0

    MachineClass <- setRefClass("MachineClass",
    fields = list(StartUpTime="numeric",ID="numeric"),
    contains = "Process",
    methods = list(
    initialize = function()
    {
        .self$StartUpTime <<- 0.0
        .self$ID <<- NextID
        NextID <<- NextID + 1
        NUp <<- NUp + 1

    }
    Run = function()
    {
            while(1)
        {
            # record current time, now(), so can see how long machine is up
            .self$StartUpTime <- now()
            # hold for exponentially distributed up time
            UpTime <- rexp(1,UpRate)
            yield_hold(.self, UpTime) # simulate UpTime
            TotalUpTime <<- TotalUpTime + now() - .self$StartUpTime
            # NRep = NRep + 1
            # if (RepairPerson$n == 1){
            # NImmedRep = NImmedRep + 1
            #}
            #yield_request(RepairPerson)
            RepairTime <- rexp(1,RepairRate)
            # hold for exponentially distributed repair time
            yield_hold(.self, RepairTime)
        }
    }))
    main <- function()
    {
        initialize()
        # set up the two machine threads
        for(i in 1:2)
        {
            # create a MachineClass object
            M <- MachineClass()
            activate(M,M$Run()) # required
```

```
    }
    # run until simulated time 10000
    MaxSimtime = 10000.0
    simulate(MaxSimtime) # required

    paste("the percentage of up time was ", TotalUpTime/(2*MaxSimtime))
    # paste("the percentage of times repair was immediate: ", NImmedRep/NRep)
}

main()
```

```
RposimMachRep3.r
    source("Rposim.R")

    #global variables
    UpRate = 1/1.0 # reciprocal of mean up time
    RepairRate = 1/0.5 # reciprocal of mean repair time
    NextID = 0 # next available ID number for MachineClass objects
    TotalUpTime = 0.0 # total up time for all machines
    # RepairPerson = resource(1)
    # NUp = 0
    # MachineList = c(0)

    MachineClass <- setRefClass("MachineClass",
    fields = list(StartUpTime="numeric",ID="numeric"),
    contains = "Process",
    methods = list(
    initialize = function()
    {
        .self$StartUpTime <<- 0.0
        .self$ID <<- NextID
        NextID <<- NextID + 1
        NUp <<- NUp + 1
    }
    Run = function()
    {
            while(1)
        {
            # record current time, now(), so can see how long machine is up
            .self$StartUpTime <- now()
            # hold for exponentially distributed up time
            UpTime <- rexp(1,UpRate)
            yield_hold(.self, UpTime) # simulate UpTime
            TotalUpTime <<- TotalUpTime + now() - .self$StartUpTime
            # NUp = NUp - 1
            # if(NUp == 1){
            # yield_passivate(self)
            # }else if(RepairPerson$n == 1){
            # reactivate(MachineList[1-self.ID])
            #}
            # yield_request(RepairPerson)
            #
            RepairTime <- rexp(1,RepairRate)
            # hold for exponentially distributed repair time
            yield_hold(.self, RepairTime)
            # NUp = NUp + 1
            # yield_release(RepairPerson)
        }
    }))
    main <- function()
    {
        initialize()
        # set up the two machine threads
        for(i in 1:2)
        {
```

```
        # create a MachineClass object
        M <- MachineClass()
        activate(M,M$Run()) # required
    }
    # run until simulated time 10000
    MaxSimtime = 10000.0
    simulate(MaxSimtime) # required

    paste("the percentage of up time was ", TotalUpTime/(2*MaxSimtime))
    # paste("the percentage of times repair was immediate: ", NImmedRep/NRep)
}

main()
```

SimmerMachRep1.r

```
    library(simmer)

    NUM_MACHINES <- 2       # Number of machines
    NUM_REPAIRPERSONS <- 2  # Number of repair people
    REPAIRRATE <- 1/0.5     # Average amount of time to repair a machine
    UPRATE <- 1/1.0         # Average amount of time the machine is working
    SIM_TIME <- 10000       # Total simulation time

    set.seed(12345)  # Set seed
    env1 <- simmer() # Instantiate the simulated environment

        # Setup trajectory to simulate a repair event
    repair_person <- trajectory() %>%
            # Removes a resource from of the current number of resources
        seize("repair", 1) %>%
            # Create new attribute with random uptime
        set_attribute("newUptime",function(){rexp(1,1/1.0)},global=TRUE) %>%
            # Simulate time up for machine
        timeout(function() {ifelse(is.na(get_attribute(env1,"newUptime")),0,
            get_attribute(env1,"newUptime"))}) %>%
            # Simulate time to be repaired
        timeout(function(){rexp(1,REPAIRRATE)}) %>%
            # Add the resource back to the current number of resources
        release("repair", 1)

        # Setup the simulated environment
    env1 %>%
            # Adds repair resource for trajectory to pull from
        add_resource("repair", NUM_MACHINES) %>%
            # Add initial events for number of repair people
        add_generator("repair_person", repair_person, at(rep(0,NUM_REPAIRPERSONS))) %>%
            # Makes arrival events at a randomly distributed rate
        add_generator("create_new_uptime", repair_person, function() sample(rexp(1,1/1.0), 1)) %>%

            # Start the simulation
        run(until = SIM_TIME)


        # Store all values of all attribues in the environment
    x = env1 %>% get_mon_attributes
        # Store values of newUpTime
    y = x$value
        # Sum all values of newUpTime
    z = sum(y)
        # Calculate the percentage of upTime
    PercentUp = z / (NUM_MACHINES*SIM_TIME)
        # Print the percentage of upTime
    print(paste0("Percent machines are up is: ", PercentUp))
```