simmer: Discrete-Event Simulation for R

Iñaki Ucar
Universidad Carlos III de Madrid

Bart Smeets
dataroots

Arturo Azcorra
Universidad Carlos III de Madrid
IMDEA Networks Institute

Abstract

The **simmer** package brings discrete-event simulation to R. It is designed as a generic yet powerful process-oriented framework. The architecture encloses a robust and fast simulation core written in C++ with automatic monitoring capabilities. It provides a rich and flexible R API that revolves around the concept of *trajectory*, a common path in the simulation model for entities of the same type.

Keywords: R, Discrete-Event Simulation.

This manuscript corresponds to **simmer** version 4.2.1 and was typeset on January 09, 2019. For citations, please use the JSS version (see `citation("simmer")`).

1. Introduction

The complexity of many real-world systems involves unaffordable analytical models, and consequently, such systems are commonly studied by means of simulation. This problem-solving technique precedes the emergence of computers, but tool and technique got entangled as a result of their development. As defined by Shannon (1975), simulation "is the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behaviour of the system or of evaluating various strategies (within the limits imposed by a criterion or a set of criteria) for the operation of the system."

Different types of simulation apply depending on the nature of the system under consideration. A common model taxonomy classifies simulation problems along three main dimensions (Law and Kelton 2000): (*i*) deterministic vs. stochastic, (*ii*) static vs. dynamic (depending on whether they require a *time* component), and (*iii*) continuous vs. discrete (depending on how the system changes). For instance, *Monte Carlo methods* are well-known examples of static stochastic simulation techniques. On the other hand, *Discrete-event simulation* (DES) is a specific technique for modelling stochastic, dynamic and discretely evolving systems. As opposed to *continuous simulation*, which typically uses smoothly-evolving equational models, DES is characterised by sudden state changes at precise points of (simulated) time.

Customers arriving at a bank, products being manipulated in a supply chain, or packets traversing a network are common examples of such systems. The discrete nature of a given system arises as soon as its behaviour can be described in terms of *events*, which is the most fundamental concept in DES. An *event* is an instantaneous occurrence that may change the state of the system, while, between events, all the state variables remain constant.

The applications of DES are vast, including, but not limited to, areas such as manufacturing

systems, construction engineering, project management, logistics, transportation systems, business processes, healthcare and telecommunications networks (Banks 2005). The simulation of such systems provides insights into the process' risk, efficiency and effectiveness. Also, by simulation of an alternative configuration, one can proactively estimate the effects of changes to the system. In turn, this allows one to get clear insights into the benefits of process redesign strategies (e.g., extra resources). A wide range of practical applications is prompted by this, such as analysing bottlenecks in customer services centres, optimising patient flows in hospitals, testing the robustness of a supply chain or predicting the performance of a new protocol or configuration of a telecommunications network.

There are several world views, or programming styles, for DES (Banks 2005). In the *activity-oriented* approach, a model consists of sequences of activities, or operations, waiting to be executed depending on some conditions. The simulation clock advances in fixed time increments. At each step, the whole list of activities is scanned, and their conditions, verified. Despite its simplicity, the simulation performance is too sensitive to the election of such a time increment. Instead, the *event-oriented* approach completely bypasses this issue by maintaining a list of scheduled events ordered by time of occurrence. Then, the simulation just consists in jumping from event to event, sequentially executing the associated routines. Finally, the *process-oriented* approach refines the latter with the addition of interacting *processes*, whose activation is triggered by events. In this case, the modeller defines a set of processes, which correspond to entities or objects of the real system, and their life cycle.

simmer (Ucar and Smeets 2017a) is a DES package for R which enables high-level process-oriented modelling, in line with other modern simulators. But in addition, it exploits the novel concept of *trajectory*: a common path in the simulation model for entities of the same type. In other words, a trajectory consist of a list of standardised actions which defines the life cycle of equivalent processes. This design pattern is flexible and simple to use, and takes advantage of the chaining/piping workflow introduced by the **magrittr** package (Bache and Wickham 2014).

Let us illustrate this with a simple example taken from Pidd (1988), Section 5.3.1:

> Consider a simple engineering job shop that consists of several identical machines. Each machine is able to process any job and there is a ready supply of jobs with no prospect of any shortages. Jobs are allocated to the first available machine. The time taken to complete a job is variable but is independent of the particular machine being used. The machine shop is staffed by operatives who have two tasks:
>
> 1. RESET machines between jobs if the cutting edges are still OK.
> 2. RETOOL those machines with cutting edges that are too worn to be reset.
>
> In addition, an operator may be AWAY while attending to personal needs.

Figure 1 shows the activity cycle diagram for the considered system. Circles (READY, STOPPED, OK, WAITING) represent states of the machines or the operatives respectively, while rectangles (RUNNING, RETOOL, RESET, AWAY) represent activities that take some (random) time to complete. Two kind of processes can be identified: shop jobs, which use machines and degrade them, and personal tasks, which take operatives AWAY for some time.

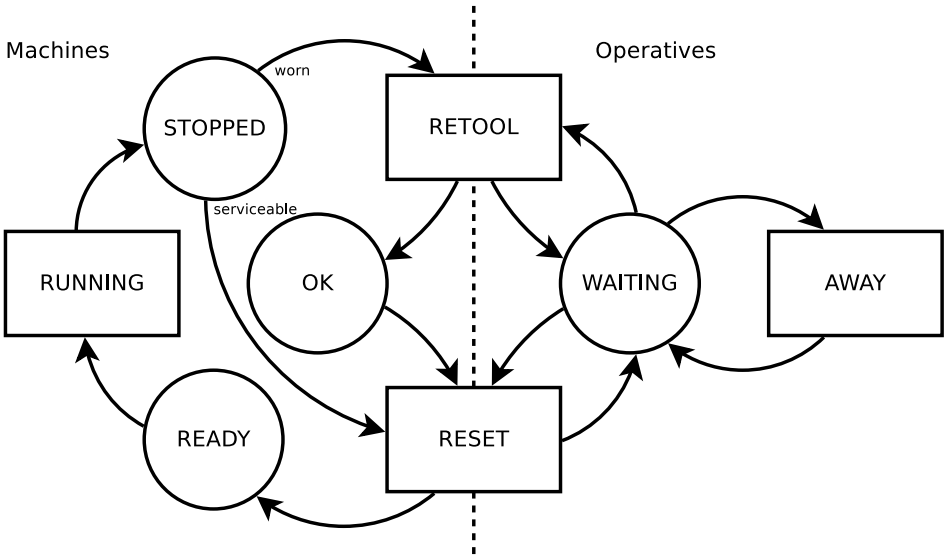


Figure 1: The job shop activity cycle diagram.

There is a natural way of simulating this system with **simmer** which consist in considering machines and operatives as resources, and describing the life cycles of shop jobs and personal tasks as trajectories.

First of all, let us instantiate a new simulation environment and define the completion time for the different activities as random draws from exponential distributions. Likewise, the interarrival times for jobs and tasks are defined (`NEW_JOB`, `NEW_TASK`), and we consider a probability of 0.2 for a machine to be worn after running a job (`CHECK_JOB`).

```
R> library("simmer")
R>
R> set.seed(1234)
R>
R> (env <- simmer("Job Shop"))

simmer environment: Job Shop | now: 0 | next:
{ Monitor: in memory }

R> RUNNING <- function() rexp(1, 1)
R> RETOOL <- function() rexp(1, 2)
R> RESET <- function() rexp(1, 3)
R> AWAY <- function() rexp(1, 1)
R> CHECK_WORN <- function() runif(1) < 0.2
R> NEW_JOB <- function() rexp(1, 5)
R> NEW_TASK <- function() rexp(1, 1)
```

The trajectory of an incoming job starts by seizing a machine in READY state. It takes some random time for RUNNING it after which the machine's serviceability is checked. An operative and some random time to RETOOL the machine may be needed, and either way an

operative must RESET it. Finally, the trajectory releases the machine, so that it is READY again. On the other hand, personal tasks just seize operatives for some time.

```
R> job <- trajectory() %>%
R+   seize("machine") %>%
R+   timeout(RUNNING) %>%
R+   branch(
R+     CHECK_WORN, continue = TRUE,
R+     trajectory() %>%
R+       seize("operative") %>%
R+       timeout(RETOOL) %>%
R+       release("operative")
R+   ) %>%
R+   seize("operative") %>%
R+   timeout(RESET) %>%
R+   release("operative") %>%
R+   release("machine")
R>
R> task <- trajectory() %>%
R+   seize("operative") %>%
R+   timeout(AWAY) %>%
R+   release("operative")
```

Once the processes' trajectories are defined, we append 10 identical machines and 5 operatives to the simulation environment, as well as two generators for jobs and tasks.

```
R> env %>%
R+   add_resource("machine", 10) %>%
R+   add_resource("operative", 5) %>%
R+   add_generator("job", job, NEW_JOB) %>%
R+   add_generator("task", task, NEW_TASK) %>%
R+   run(until=1000)

simmer environment: Job Shop | now: 1000 | next: 1000.11891377085
{ Monitor: in memory }
{ Resource: machine | monitored: TRUE | server status: 9(10) | queue... }
{ Resource: operative | monitored: TRUE | server status: 4(5) | queue... }
{ Source: job | monitored: 1 | n_generated: 4954 }
{ Source: task | monitored: 1 | n_generated: 1041 }
```

The simulation has been run for 1000 units of time, and the simulator has monitored all the state changes and lifetimes of all processes, which enables any kind of analysis without any additional effort from the modeller's side. For instance, we may extract a history of the resource's state to analyse the average number of machines/operatives in use as well as the average number of jobs/tasks waiting for an assignment.

```
R> aggregate(cbind(server, queue) ~ resource, get_mon_resources(env), mean)
```

```
   resource   server     queue
1   machine 7.632616 0.5598666
2 operative 3.389082 0.3436009
```

The development of the **simmer** package started in the second half of 2014. The initial need for a DES framework for R came up in projects related to process optimisation in healthcare facilities. Most of these cases involved patients following a clear trajectory through a care process. This background is not unimportant, as it lead to the adoption and implementation of a *trajectory* concept at the very core of **simmer**'s DES engine. This strong focus on clearly defined *trajectories* is somewhat innovative and, more importantly, very intuitive. Furthermore, this framework relies on a fast C++ simulation core to boost performance and make DES modelling in R not only effective, but also efficient.

Over time, the **simmer** package has seen significant improvements and has been at the forefront of DES for R. Although it is the most generic DES framework, it is however not the only R package which delivers such functionality. For example, the **SpaDES** package (Chubaty and McIntire 2017) focuses on spatially explicit discrete models, and the **queuecomputer** package (Ebert 2017) implements an efficient method for simulating queues with arbitrary arrival and service times. Going beyond the R language, the direct competitors to **simmer** are **SimPy** (Team SimPy 2017) and **SimJulia** (Lauwens 2017), built for the Python and Julia languages.

2. The simulation core design

The core of any modern discrete-event simulator comprises two main components: an event list, ordered by time of occurrence, and an event loop that extracts and executes events. In contrast to other interpreted languages such as Python, which is compiled by default to an intermediate byte-code, R code is purely parsed and evaluated at runtime[1]. This fact makes it a particularly slow language for DES, which consists of executing complex routines (pieces of code associated to the events) inside a loop while constantly allocating and deallocating objects (in the event queue).

In fact, first attempts were made in pure R by these authors, and a minimal process-based implementation with **R6** classes (Chang 2017) proved to be unfeasible in terms of performance compared to similar approaches in pure Python. For this reason, it was decided to provide a robust and fast simulation core written in C++. The R API interfaces with this C++ core by leveraging the **Rcpp** package (Eddelbuettel and François 2011; Eddelbuettel 2013), which has become one of the most popular ways of extending R packages with C or C++ code.

The following sections are devoted to describe the simulation core architecture. First, we establish the DES terminology used in the rest of the paper. Then, the architectural choices made are discussed, as well as the event queue and the *simultaneity problem*, an important topic that every DES framework has to deal with.

[1]Some effort has been made in this line with the **compiler** package, introduced in R version 2.13.0 (Luke Tierney 2016), furthermore, a JIT-compiler was included in R version 3.4.0.

2.1. Terminology

This document uses some DES-specific terminology, e.g., *event*, *state*, *entity*, *process* or *attribute*. Such standard terms can be easily found in any textbook about DES (refer to Banks (2005), for instance). There are, however, some **simmer**-specific terms, and some elements that require further explanation to understand the package architecture.

Resource A passive entity, as it is commonly understood in standard DES terminology. However, **simmer** resources are conceived with queuing systems in mind, and therefore they comprise two internal self-managed parts:

- **Server** which, conceptually, represents the resource itself. It has a specified capacity and can be seized and released.
- **Queue** A priority queue of a certain size.

Manager An active entity, i.e., a process, that has the ability to adjust properties of a resource (capacity and queue size) at run-time.

Source A process responsible for creating new *arrivals* with a given interarrival time pattern and inserting them into the simulation model.

Arrival A process capable of interacting with resources or other entities of the simulation model. It may have some attributes and prioritisation values associated and, in general, a limited lifetime. Upon creation, every arrival is attached to a given *trajectory*.

Trajectory An interlinkage of *activities* constituting a recipe for arrivals attached to it, i.e., an ordered set of actions that must be executed. The simulation model is ultimately represented by a set of trajectories.

Activity The individual unit of action that allows arrivals to interact with resources and other entities, perform custom routines while spending time in the system, move back and forth through the trajectory dynamically, and much more.

2.2. Architecture

Extending an R package (or any other piece of software written in any interpreted language) with compiled code poses an important trade-off between performance and flexibility: placing too much functionality into the compiled part produces gains in performance, but degrades modelling capabilities, and vice versa. The following lines are devoted to discuss how this trade-off is resolved in **simmer**.

Figure 2 sketches a UML (Unified Modelling Language) description of the architecture, which constitutes a process-based design, as in many modern DES frameworks. We draw the attention now to the C++ classes (depicted in white).

The first main component is the `Simulator` class. It comprises the event loop and the event queue, which will be addressed in the next section. The `Simulator` provides methods for scheduling and unscheduling events. Moreover, it is responsible for managing simulation-wide entities (e.g., resources and arrival sources) and facilities (e.g., signaling between processes and batches) through diverse C++ unordered maps:

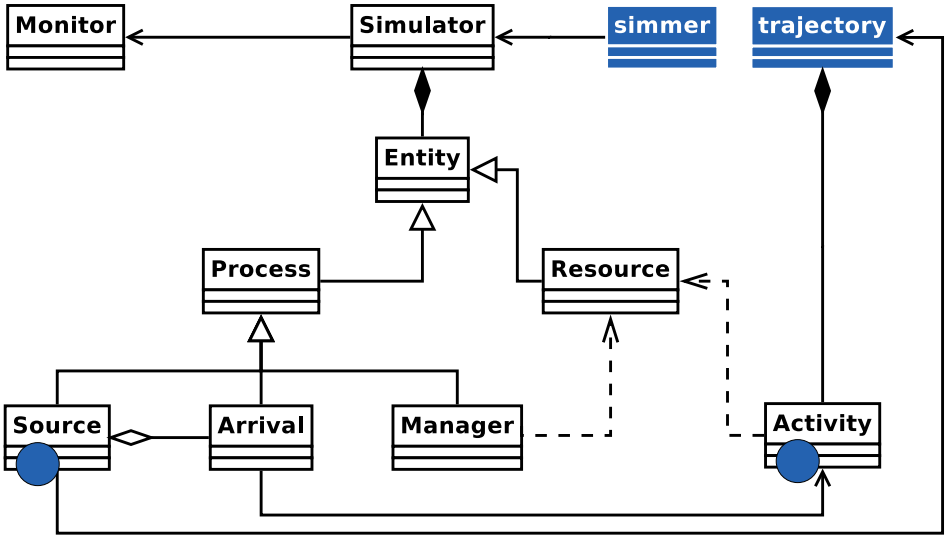


Figure 2: UML diagram of the simulation core architecture. Blue classes represent how R encapsulates the C++ core. Blue circles represent how C++ interfaces with R.

- Maps of resources and processes (sources, arrivals and managers) by name.
- A map of pending events, which allows to unschedule a given process.
- Maps of signals subscribed by arrivals and handlers defined for different signals.
- Maps for forming batches of arrivals, named and unnamed.

This class also holds global attributes and monitoring information. Thus, monitoring counters, which are derived from the `Monitor` class, are centralised, and they register every change of state produced during the simulation time. There are five types of built-in changes of state that are recorded by calling `Monitor`'s `record_*()` methods:

- An arrival is accepted into a resource (served or enqueued). The resource notifies about the new status of its internal counters.
- An arrival leaves a resource. The resource notifies the new status of its internal counters, and the arrival notifies start, end and activity times in that particular resource.
- A resource is modified during runtime (i.e., a change in the capacity or queue size). The resource notifies the new status of its internal counters.
- An arrival modifies an attribute, one of its own or a global one. The arrival notifies the new value.
- An arrival leaves its trajectory by exhausting the activities associated (considered as *finished*) or because of another reason (*non-finished*, e.g., it is rejected from a resource). The arrival notifies global start, end and activity times.

As mentioned in the previous section, there are two types of entities: passive ones (`Resource`) and active ones (processes `Source`, `Arrival` and `Manager`). Sources create new arrivals, and the latter are the main actors of the simulation model. Managers can be used for dynamically changing the properties of a resource (capacity and queue size). All processes share a `run()` method that is invoked by the event loop each time a new event is extracted from the event list.

There is a fourth kind of process not shown in Figure 2, called `Task`. It is a generic process that executes a given function once, and it is used by arrivals, resources, activities and the simulator itself to trigger dynamic actions or split up events. A `Task` is for instance used under the hood to trigger reneging or to broadcast signals after some delay.

The last main component, completely isolated from the `Simulator`, is the `Activity` class. This abstract class represents a clonable object, chainable in a double-linked list to form trajectories. Most of the activities provided by **simmer** derive from it. `Fork` is another abstract class (not depicted in Figure 2) which is derived from `Activity`. Any activity supporting the definition of sub-trajectories must derive from this one instead, such as `Seize`, `Branch` or `Clone`. All the activities must implement the virtual methods `print()` and `run()`.

Finally, it is worth mentioning the couple of blue circles depicted in Figure 2. They represent the *points of presence* of R in the C++ core, i.e., where the core interfaces back with R to execute custom user-defined code.

In summary, the C++ core is responsible for all the heavy tasks, i.e., managing the event loop, the event list, generic resources and processes, collecting all the statistics, and so on. And still, it provides enough flexibility to the user for modelling the interarrival times from R and execute any custom user-defined code through the activities.

2.3. The event queue

The event queue is the most fundamental part of any DES software. It is responsible for maintaining a list of events to be executed by the event loop in an ordered fashion by time of occurrence. This last requirement establishes the need for a data structure with a low access, search, insertion and deletion complexity. A binary tree is a well-known data structure that satisfies these properties, and it is commonly used for this purpose. Unfortunately, binary trees, or equivalent structures, cannot be efficiently implemented without pointers, and this is the main reason why pure R is very inefficient for DES.

In **simmer**, the event queue is defined as a C++ multiset, a kind of associative container implemented as a balanced tree internally. Apart from the efficiency, it was selected to support event unscheduling through iterators. Each event holds a pointer to a process, which will be retrieved and run in the event loop. Events are inserted in the event queue ordered by 1) time of occurrence and 2) priority. This secondary order criterion is devoted to solve a common issue for DES software called *the simultaneity problem*.

The simultaneity problem

As noted by Rönngren and Liljenstam (1999), Jha and Bagrodia (2000), there are many circumstances from which simultaneous events (i.e., events with the same timestamp) may arise. How they are handled by a DES framework has critical implications on reproducibility and simulation correctness.

As an example of the implications, let us consider an arrival seizing a resource at time t_{i-1}, which has `capacity=1` and `queue_size=0`. At time t_i, two simultaneous events happen: 1) the resource is released, and 2) another arrival tries to seize the resource. It is indisputable what should happen in this situation: the new arrival seizes the resource while the other continues its path. But note that if 2) is executed *before* 1), the new arrival is rejected (!). Therefore, it is obvious that release events must always be executed *before* seize events.

If we consider a dynamically managed resource (i.e., its capacity changes over time) and, instead of the event 1) in the previous example, the manager increases the capacity of the resource, we are in the very same situation. Again, it is obvious that resource managers must be executed *before* seize attempts.

A further analysis reveals that, in order to preserve correctness and prevent a simulation crash, it is necessary to break down resource releases in two parts with different priorities: the release in itself and a post-release event that tries to serve another arrival from the queue. Thus, every resource manager must be executed *after* releases and *before* post-releases. This and other issues are solved with a priority system (see Table 1) embedded in the event list implementation that provides a deterministic and consistent execution of simultaneous events.

Table 1: Priority system (in decreasing order) and events associated.

Priority	Event
`PRIORITY_MAX`	Terminate arrivals
`PRIORITY_RELEASE`	Resource release
`PRIORITY_MANAGER`	Manager action (e.g., resource capacity change)
`PRIORITY_RELEASE_POST`	Resource post-release (i.e., serve from the queue)
...	General activities
`PRIORITY_MIN`	Other tasks (e.g., new arrivals, timers...)

3. The simmer API

The R API exposed by **simmer** comprises two main elements: the `simmer` environment (or *simulation environment*) and the `trajectory` object, which are depicted in Figure 2 (blue classes). As we will see throughout this section, simulating with **simmer** simply consists of building a simulation environment and one or more trajectories. For this purpose, the API is composed of verbs and actions that can be chained together. For easy-of-use, these have been made fully compatible with the pipe operator (`%>%`) from the **magrittr** package.

3.1. The trajectory object

A *trajectory* can be defined as a recipe and consists of an ordered set of *activities*. The idea behind this concept is very similar to the idea behind **dplyr** for data manipulation (Wickham and Francois 2017). To borrow the words of H. Wickham, "by constraining your options, it simplifies how you can think about" discrete-event modelling. Activities are *verbs* that correspond to common functional DES blocks.

The `trajectory()` method instantiates the object, and activities can be appended using the `%>%` operator:

```
R> traj0 <- trajectory() %>%
R+   log_("Entering the trajectory") %>%
R+   timeout(10) %>%
R+   log_("Leaving the trajectory")
```

The trajectory above illustrates the two most basic activities available: displaying a message (`log_()`) and spending some time in the system (`timeout()`). An arrival attached to this trajectory will execute the activities in the given order, i.e., it will display "Entering the trajectory", then it will spend 10 units of (simulated) time, and finally it will display "Leaving the trajectory".

The example uses *fixed parameters*: a string and a numeric value respectively. However, at least the main parameter for all activities (this is specified in the documentation) can also be what we will call a *dynamical parameter*, i.e., a function. This thus, although not quite useful yet, is also valid:

```
R> traj1 <- trajectory() %>%
R+   log_(function() "Entering the trajectory") %>%
R+   timeout(function() 10) %>%
R+   log_(function() "Leaving the trajectory")
```

Also, trajectories can be split apart, joined together and modified:

```
R> traj2 <- join(traj0[c(1, 3)], traj0[2])
R> traj2[1] <- traj2[3]
R> traj2

trajectory: anonymous, 3 activities
{ Activity: Timeout      | delay: 10 }
{ Activity: Log          | message: Leaving th..., level: 0 }
{ Activity: Timeout      | delay: 10 }
```

There are many activities available. We will briefly review them by categorising them into different topics.

Arrival properties

Arrivals are able to store attributes and modify these using `set_attribute()`. Attributes consist of pairs `(key, value)` (character and numeric respectively) which by default are set *per arrival* unless they are defined as global. As we said before, all activities support at least one dynamical parameter. In the case of `set_attribute()`, both arguments, `keys` and `values`, can be vectors or functions returning vectors.

Attributes can be retrieved in any R function by calling `get_attribute()`, whose first argument must be a `simmer` object. For instance, the following trajectory prints `81`:

```
R> env <- simmer()
R>
R> traj <- trajectory() %>%
R+   set_attribute("weight", 80) %>%
R+   set_attribute("weight", 1, mod="+") %>%
R+   log_(function() paste0("My weight is ", get_attribute(env, "weight")))
```

Arrivals also hold a set of three prioritisation values for accessing resources:

priority A higher value equals higher priority. The default value is the minimum priority, which is 0.

preemptible If a preemptive resource is seized, this parameter establishes the minimum incoming priority that can preempt this arrival (the activity is interrupted and another arrival with a `priority` greater than `preemptible` gains the resource). In any case, `preemptible` must be equal or greater than `priority`, and thus only higher priority arrivals can trigger preemption.

restart Whether the ongoing activity must be restarted after being preempted.

These three values are established for all the arrivals created by a particular source, but they can also be dynamically changed on a per-arrival basis using the `set_prioritization()` and `get_prioritization()` activities, in the same way as attributes.

Interaction with resources

The two main activities for interacting with resources are `seize()` and `release()`. In their most basic usage, they seize/release a given `amount` of a resource specified by name. It is also possible to change the properties of the resource with `set_capacity()` and `set_queue_size()`.

The `seize()` activity is special in the sense that the outcome depends on the state of the resource. The arrival may successfully seize the resource and continue its path, but it may also be enqueued or rejected and dropped from the trajectory. To handle these special cases with total flexibility, `seize()` supports the specification of two optional sub-trajectories: `post.seize`, which is followed after a successful seize, and `reject`, followed if the arrival is rejected. As in every activity supporting the definition of sub-trajectories, there is a boolean parameter called `continue`. For each sub-trajectory, it controls whether arrivals should continue to the activity following the `seize()` in the main trajectory after executing the sub-trajectory.

```
R> patient <- trajectory() %>%
R+   log_("arriving...") %>%
R+   seize(
R+     "doctor", 1, continue = c(TRUE, FALSE),
R+     post.seize = trajectory("accepted patient") %>%
R+       log_("doctor seized"),
R+     reject = trajectory("rejected patient") %>%
R+       log_("rejected!") %>%
R+       seize("nurse", 1) %>%
R+       log_("nurse seized") %>%
R+       timeout(2) %>%
R+       release("nurse", 1) %>%
R+       log_("nurse released")
R+   ) %>%
R+   timeout(5) %>%
R+   release("doctor", 1) %>%
R+   log_("doctor released")
R>
```

```
R> env <- simmer() %>%
R+   add_resource("doctor", capacity = 1, queue_size = 0) %>%
R+   add_resource("nurse", capacity = 10, queue_size = 0) %>%
R+   add_generator("patient", patient, at(0, 1)) %>%
R+   run()

0: patient0: arriving...
0: patient0: doctor seized
1: patient1: arriving...
1: patient1: rejected!
1: patient1: nurse seized
3: patient1: nurse released
5: patient0: doctor released
```

The value supplied to all these methods may be a dynamical parameter. On the other hand, the resource name must be fixed. There is a special mechanism to select resources dynamically: the `select()` activity. It marks a resource as selected for an arrival executing this activity given a set of `resources` and a `policy`. There are several policies implemented internally that can be accessed by name:

shortest-queue The resource with the shortest queue is selected.
round-robin Resources will be selected in a cyclical nature.
first-available The first available resource is selected.
random A resource is randomly selected.

Its `resources` parameter is allowed to be dynamical, and there is also the possibility of defining custom policies. Once a resource is selected, there are special versions of the aforementioned activities for interacting with resources without specifying its name, such as `seize_selected()`, `set_capacity_selected()` and so on.

Interaction with sources

There are four activities specifically intended to modify arrival sources. An arrival may `activate()` or `deactivate()` a source, but also modify with `set_trajectory()` the trajectory to which it attaches the arrivals created, or set a new interarrival distribution with `set_source()`. For dynamically selecting a source, the parameter that specifies the source name in all these methods can be dynamical.

```
R> traj <- trajectory() %>%
R+   deactivate("dummy") %>%
R+   timeout(1) %>%
R+   activate("dummy")
R>
R> simmer() %>%
R+   add_generator("dummy", traj, function() 1) %>%
R+   run(10) %>%
R+   get_mon_arrivals()
```

```
    name start_time end_time activity_time finished replication
1 dummy0          1        2             1     TRUE           1
2 dummy1          3        4             1     TRUE           1
3 dummy2          5        6             1     TRUE           1
4 dummy3          7        8             1     TRUE           1
```

Branching

A branch is a point in a trajectory in which one or more sub-trajectories may be followed. Two types of branching are supported in **simmer**. The `branch()` activity places the arrival in one of the sub-trajectories depending on some condition evaluated in a dynamical parameter called `option`. It is the equivalent of an `if/else` in programming, i.e., if the value of `option` is i, the i-th sub-trajectory will be executed. On the other hand, the `clone()` activity is a *parallel* branch. It does not take any option, but replicates the arrival `n-1` times and places each one of them into the `n` sub-trajectories supplied.

```
R> env <- simmer()
R>
R> traj <- trajectory() %>%
R+   branch(
R+     option = function() round(now(env)), continue = c(FALSE, TRUE),
R+     trajectory() %>% log_("branch 1"),
R+     trajectory() %>% log_("branch 2")
R+   ) %>%
R+   clone(
R+     n = 2,
R+     trajectory() %>% log_("clone 0"),
R+     trajectory() %>% log_("clone 1")
R+   ) %>%
R+   synchronize(wait = TRUE) %>%
R+   log_("out")
R>
R> env %>%
R+   add_generator("dummy", traj, at(1, 2)) %>%
R+   run() %>% invisible

1: dummy0: branch 1
2: dummy1: branch 2
2: dummy1: clone 0
2: dummy1: clone 1
2: dummy1: out
```

Note that `clone()` is the only exception among all activities supporting sub-trajectories that does not accept a `continue` parameter. By default, all the clones continue in the main trajectory after this activity. To remove all of them except for one, the `synchronize()` activity may be used.

Loops

There is a mechanism, `rollback()`, for going back in a trajectory and thus executing loops over a number of activities. This activity causes the arrival to step back a given `amount` of activities (that can be dynamical) a number of `times`. If a `check` function returning a boolean is supplied, the `times` parameter is ignored and the arrival determines whether it must step back each time it hits the `rollback`.

```
R> hello <- trajectory() %>%
R+   log_("Hello!") %>%
R+   timeout(1) %>%
R+   rollback(amount = 2, times = 2)
R>
R> simmer() %>%
R+   add_generator("hello_sayer", hello, at(0)) %>%
R+   run() %>% invisible
```

```
0: hello_sayer0: Hello!
1: hello_sayer0: Hello!
2: hello_sayer0: Hello!
```

Batching

Batching consists of collecting a number of arrivals before they can continue their path in the trajectory as a unit[2]. This means that if, for instance, 10 arrivals in a batch try to seize a unit of a certain resource, only one unit may be seized, not 10. A batch may be splitted with `separate()`, unless it is marked as `permanent`.

```
R> roller <- trajectory() %>%
R+   batch(10, timeout = 5, permanent = FALSE) %>%
R+   seize("rollercoaster", 1) %>%
R+   timeout(5) %>%
R+   release("rollercoaster", 1) %>%
R+   separate()
```

By default, all the arrivals reaching a batch are joined into it, and batches wait until the specified number of arrivals are collected. Nonetheless, arrivals can avoid joining the batch under any constraint if an optional function returning a boolean, `rule`, is supplied. Also, a batch may be triggered before collecting a given amount of arrivals if some `timeout` is specified. Note that batches are shared only by arrivals directly attached to the same trajectory. Whenever a globally shared batch is needed, a common `name` must be specified.

[2] A concrete example of this is the case where a number of people (the arrivals) together take, or rather seize, an elevator (the resource).

Asynchronous programming

There are a number of methods enabling asynchronous events. The `send()` activity broadcasts one or more `signals` to all the arrivals subscribed to them. Signals can be triggered immediately or after some `delay`. In this case, both parameters, `signals` and `delay`, can be dynamical. Arrivals are able to block and `wait()` until a certain signal is received.

Arrivals can subscribe to `signals` and (optionally) assign a `handler` using the `trap()` activity. Upon a signal reception, the arrival stops the current activity and executes the `handler`[3] if provided. Then, the execution returns to the activity following the point of interruption. Nonetheless, trapped signals are ignored when the arrival is waiting in a resource's queue. The same applies inside a batch: all the signals subscribed before entering the batch are ignored. Finally, the `untrap()` activity can be used to unsubscribe from `signals`.

```
R> t_blocked <- trajectory() %>%
R+   trap(
R+     "you shall pass",
R+     handler = trajectory() %>%
R+       log_("got a signal!")
R+   ) %>%
R+   log_("waiting...") %>%
R+   wait() %>%
R+   log_("continuing!")
R>
R> t_signal <- trajectory() %>%
R+   log_("you shall pass") %>%
R+   send("you shall pass")
R>
R> simmer() %>%
R+   add_generator("blocked", t_blocked, at(0)) %>%
R+   add_generator("signaler", t_signal, at(5)) %>%
R+   run() %>% invisible

0: blocked0: waiting...
5: signaler0: you shall pass
5: blocked0: got a signal!
5: blocked0: continuing!
```

By default, signal handlers may be interrupted as well by other signals, meaning that a `handler` may keep restarting if there are frequent enough signals being broadcasted. If an uninterruptible `handler` is needed, this can be achieved by setting the flag `interruptible` to `FALSE` in `trap()`.

Reneging

Besides being rejected while trying to seize a resource, arrivals are also able to leave the trajectory at any moment, synchronously or asynchronously. Namely, reneging means that

[3] The `handler` parameter accepts a trajectory object. Once the handler gets called, it will route the arrival to this sub-trajectory.

an arrival abandons the trajectory at a given moment. The most simple activity enabling this is `leave`, which immediately triggers the action given some probability. Furthermore, `renege_in()` and `renege_if()` trigger reneging asynchronously after some timeout `t` or if a `signal` is received respectively, unless the action is aborted with `renege_abort()`. Both `renege_in()` and `renege_if()` accept an optional sub-trajectory, `out`, that is executed right before leaving.

```
R> bank <- trajectory() %>%
R+   log_("Here I am") %>%
R+   renege_in(
R+     5,
R+     out = trajectory() %>%
R+       log_("Lost my patience. Reneging...")
R+   ) %>%
R+   seize("clerk", 1) %>%
R+   renege_abort() %>%
R+   log_("I'm being attended") %>%
R+   timeout(10) %>%
R+   release("clerk", 1) %>%
R+   log_("Finished")
R>
R> simmer() %>%
R+   add_resource("clerk", 1) %>%
R+   add_generator("customer", bank, at(0, 1)) %>%
R+   run() %>% invisible

0: customer0: Here I am
0: customer0: I'm being attended
1: customer1: Here I am
6: customer1: Lost my patience. Reneging...
10: customer0: Finished
```

3.2. The simulation environment

The simulation environment manages resources and sources, and controls the simulation execution. The `simmer()` method instantiates the object, after which resources and sources can be appended using the `%>%` operator:

```
R> env <- simmer() %>%
R+   add_resource("res_name", 1) %>%
R+   add_generator("arrival", traj0, function() 25) %>%
R+   print()

simmer environment: anonymous | now: 0 | next: 0
{ Monitor: in memory }
{ Resource: res_name | monitored: TRUE | server status: 0(1) | queue... }
{ Source: arrival | monitored: 1 | n_generated: 0 }
```

Then, the simulation can be executed, or `run()`, `until` a stop time:

```
R> env %>%
R+   run(until=30)

25: arrival0: Entering the trajectory

simmer environment: anonymous | now: 30 | next: 35
{ Monitor: in memory }
{ Resource: res_name | monitored: TRUE | server status: 0(1) | queue... }
{ Source: arrival | monitored: 1 | n_generated: 2 }
```

There are a number of methods for extracting information, such as the simulation time (`now()`), future scheduled events (`peek()`), and *getters* for obtaining resources' and sources' parameters (capacity, queue size, server count and queue count; number of arrivals generated so far). There are also several *setters* available for resources and sources (capacity, queue size; trajectory, distribution).

A `simmer` object can be `reset()` and re-run. However, there is a special method, `wrap()`, intended to extract all the information from the C++ object encapsulated into a `simmer` environment and to deallocate that object. Thus, most of the *getters* work also when applied to wrapped environments, but such an object cannot be reset or re-run anymore.

Resources

A **simmer** resource, as stated in Section 2.1, comprises two internal self-managed parts: a server and a priority queue. Three main parameters define a resource: `name` of the resource, `capacity` of the server and `queue_size` (0 means no queue). Resources are monitored and non-preemptive by default. Preemption means that if a high priority arrival becomes eligible for processing, the resource will temporarily stop the processing of one (or more) of the lower priority arrivals being served. For preemptive resources, the `preempt_order` defines which arrival should be stopped first if there are many lower priority arrivals, and it assumes a first-in-first-out (FIFO) policy by default. Any preempted arrival is enqueued in a dedicated queue that has a higher priority over the main one (i.e., it is served first). The `queue_size_strict` parameter controls whether this dedicated queue must be taken into account for the queue size limit, if any. If this parameter enforces the limit, then rejection may occur in the main queue.

Sources

Three main parameters define a source: a `name_prefix` for each generated arrival, a trajectory to attach them to and a source of interarrival times. A monitoring flag accepts several levels in this case to support arrival monitoring (default) and, additionally, attribute monitoring.

There are two kinds of source: *generators* and *data sources*. A *generator* (`add_generator` method, as depicted in many examples throughout this article) is a dynamic source that draws interarrival times from a user-provided function. The `add_dataframe` method allows the user to set up a *data source* which draws arrivals from a provided data frame. The former is the

most versatile source, but the latter facilitates the attachment of pre-existent or pre-computed data with attributes that are automatically initialised on arrival creation.

In the case of generators, the interarrival `distribution` must return one or more interarrival times for each call. Internally, generators create as many arrivals as values returned by this function. Whenever a negative interarrival value is obtained, the generator stops. In the case of data sources, they draw arrivals from the `data` provided in batches. Both generators and data sources re-schedule themselves with a delay equal to the sum of the values obtained to produce the next batch.

3.3. Monitoring and data retrieval

There are three methods for obtaining monitored data (if any) about arrivals, resources and attributes. They can be applied to a single simulation environment or to a list of environments, and the returning object is always a data frame, even if no data was found. Each processed simulation environment is treated as a different replication, and a numeric column named `replication` is added to every returned data frame with environment indexes as values.

get_mon_arrivals() Returns timing information per arrival: `name` of the arrival, `start_time`, `end_time`, `activity_time` (time not spent in resource queues) and a flag, `finished`, that indicates whether the arrival exhausted its activities (or was rejected). By default, this information is referred to the arrivals' entire lifetime, but it may be obtained on a per-resource basis by specifying `per_resource=TRUE`.

get_mon_resources() Returns state changes in resources: `resource` name, `time` instant of the event that triggered the state change, `server` count, `queue` count, `capacity`, `queue_size`, `system` count (`server` + `queue`) and system `limit` (`capacity` + `queue_size`).

get_mon_attributes() Returns state changes in attributes: `name` of the attribute, `time` instant of the event that triggered the state change, name of `key` that identifies the attribute and `value`.

4. Modelling with simmer

The following sections aim to provide some basic modelling examples. The topics addressed are queuing systems, replication, parallelisation and some best practices. We invite the reader to learn about a broader selection of activities and modelling techniques available in the package vignettes, which cover the use of attributes, loops, batching, branching, shared events, reneging and advanced uses of resources among others.

4.1. Queuing systems

The concept of *trajectory* developed in **simmer** emerges as a natural way to simulate a wide range of problems related to Continuous-Time Markov Chains (CTMC), and more specifically to the so-called birth-death processes and queuing systems. Indeed, **simmer** not only provides very flexible resources (with or without queue), branches, delays and arrival sources, but they are bundled in a very comprehensive framework of verbs that can be chained with the pipe operator. Let us explore the expressiveness of a **simmer** trajectory using a *traditional* queuing

example: the M/M/1. The package vignettes include further examples on M/M/c/k systems, queueing networks and CTMC models.

In Kendall's notation (Kendall 1953), an M/M/1 system has exponential arrivals (**M**/M/1), a single server (M/M/**1**) with exponential service time (M/**M**/1) and an infinite queue (implicit M/M/1/∞). For instance, people arriving at an ATM at rate λ, waiting their turn in the street and withdrawing money at rate μ. These are the basic parameters of the system, whenever $\rho < 1$:

$$\rho = \frac{\lambda}{\mu} \qquad \equiv \text{Server utilisation} \tag{1}$$

$$N = \frac{\rho}{1-\rho} \qquad \equiv \text{Average number of customers in the system (queue + server)} \tag{2}$$

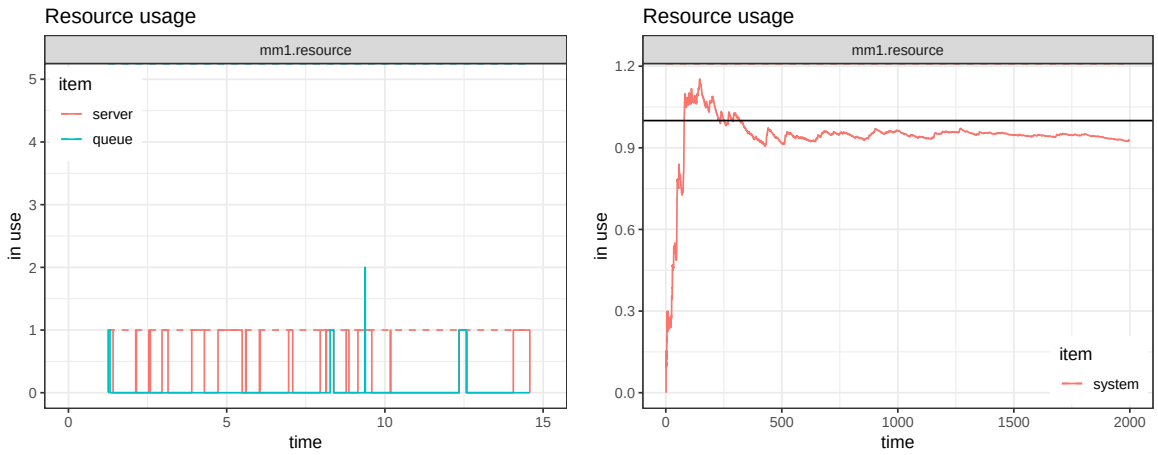
$$T = \frac{N}{\lambda} \qquad \equiv \text{Average time in the system (queue + server) [Little's law]} \tag{3}$$

If $\rho \geq 1$, it means that the system is unstable: there are more arrivals than the server is capable of handling and the queue will grow indefinitely. The simulation of an M/M/1 system is quite simple using **simmer**:

```
R> library("simmer")
R>
R> set.seed(1234)
R>
R> lambda <- 2
R> mu <- 4
R> rho <- lambda/mu
R>
R> mm1.traj <- trajectory() %>%
R+   seize("mm1.resource", amount=1) %>%
R+   timeout(function() rexp(1, mu)) %>%
R+   release("mm1.resource", amount=1)
R>
R> mm1.env <- simmer() %>%
R+   add_resource("mm1.resource", capacity=1, queue_size=Inf) %>%
R+   add_generator("arrival", mm1.traj, function() rexp(1, lambda)) %>%
R+   run(until=2000)
```

After the parameter setup, the first code block defines the trajectory: each arrival will seize the resource, wait some exponential random time (service time) and release the resource. The second code block instantiates the simulation environment, creates the resource, attaches an exponential generator to the trajectory and runs the simulation for 2000 units of time. Note that trajectories can be defined independently of the simulation environment, but it is recommended to instantiate the latter in the first place, so that trajectories are able to extract information from it (e.g., the simulation time).

As a next step, we could extract the monitoring information and perform some analyses. The extension package **simmer.plot** (Ucar and Smeets 2017b) provides convenience plotting methods to, for instance, quickly visualise the usage of a resource over time. Figure 3 gives a



(a) Instantaneous usage.

(b) Convergence over time.

Figure 3: Detail of the resource usage.

glimpse of this simulation using this package. In particular, it shows that the average number of customers in the system converges to the theoretical value given by Equation 2.

4.2. Replication and parallelisation

Typically, running a certain simulation only once is useless. In general, we will be interested in replicating the model execution many times, maybe with different initial conditions, and then perform some statistical analysis over the output. This can be easily achieved using standard R tools, e.g., `lapply()` or similar functions.

Additionally, we can leverage the parallelised version of `lapply()`, `mclapply()`, provided by the **parallel** package, to speed up this process. Unfortunately, parallelisation has the shortcoming that we lose the underlying C++ objects when each thread finishes. To avoid losing the monitored data, the `wrap()` method can be used to extract and wrap these data into a pure R object before the C++ object is garbage-collected.

The following example uses `mclapply()` and `wrap()` to perform 100 replicas of the M/M/1 simulation from the previous section (note that the trajectory is not redefined):

```
R> library("simmer")
R> library("parallel")
R>
R> set.seed(1234)
R>
R> mm1.envs <- mclapply(1:100, function(i) {
R+   simmer() %>%
R+     add_resource("mm1.resource", capacity=1, queue_size=Inf) %>%
R+     add_generator("arrival", mm1.traj, function() rexp(100, lambda)) %>%
R+     run(until=1000/lambda) %>%
R+     wrap()
R+ }, mc.set.seed=FALSE)
```

With all these replicas, we could, for instance, perform a t-test over N, the average number of customers in the system:

```
R> mm1.data <-
R+   get_mon_arrivals(mm1.envs) %>%
R+   aggregate(end_time - start_time ~ replication, data=., mean)
R>
R> t.test(mm1.data[[2]])


    One Sample t-test

data:  mm1.data[[2]]
t = 92.538, df = 99, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.4868269 0.5081617
sample estimates:
mean of x
0.4974943
```

4.3. Best practices

DES modelling can be done in an event-by-event basis, but this approach is fairly tedious and mostly unpractical. Instead, modern process-oriented approaches commonly relate to the identification of resources and processes in a given problem, and the interactions between them. The **simmer** package internally follows this paradigm and exposes generic resources and processes (*arrivals*, in **simmer** terminology), so that the user can implement all the interactions as action sequences (*trajectories*).

There are usually multiple valid ways of mapping the identified resources and processes into the elements exposed by the **simmer** API. For example, let us suppose that we would like to model an alarm clock beeping every second. In this case, the beep may be identified as a process, so that we have *different* beeps (multiple arrivals) entering a `beep` trajectory once per second:

```
R> beep <- trajectory() %>%
R+   log_("beeeep!")
R>
R> env <- simmer() %>%
R+   add_generator("beep", beep, function() 1) %>%
R+   run(2.5)

1: beep0: beeeep!
2: beep1: beeeep!
```

But instead, identifying the alarm clock as the process is equally valid, and then we have *a single alarm* (single arrival) producing all the beeps in a loop:

```
R> alarm <- trajectory() %>%
R+   timeout(1) %>%
R+   log_("beeeep!") %>%
R+   rollback(2)
R>
R> env <- simmer() %>%
R+   add_generator("alarm", alarm, at(0)) %>%
R+   run(2.5)

1: alarm0: beeeep!
2: alarm0: beeeep!
```

These are two common design patterns in **simmer** for which the outcome is the same, although there are subtle differences that depend on the problem being considered and the monitoring requirements. Furthermore, as a model becomes more complex and detailed, the resulting mapping and syntax may become more artificious. These issues are shared in different ways by other frameworks as well, such as **SimPy**, and arise due to their generic nature.

Furthermore, the piping mechanism used in the **simmer** API may invite the user to produce large monolithic trajectories. However, it should be noted that it is usually better to break them down into small manageable pieces. For instance, the following example parametrises the access to a `resource`, where `G` refers to arbitrary service times, and n servers are seized. Then, it is used to instantiate the trajectory shown in the former M/M/1 example:

```
R> xgn <- function(resource, G, n)
R+   trajectory() %>%
R+     seize(resource, n) %>%
R+     timeout(G) %>%
R+     release(resource, n)
R>
R> (mm1.traj <- xgn("mm1.resource", function() rexp(1, mu), 1))

trajectory: anonymous, 3 activities
{ Activity: Seize        | resource: mm1.resource, amount: 1 }
{ Activity: Timeout      | delay: function() }
{ Activity: Release      | resource: mm1.resource, amount: 1 }
```

Standard R tools (`lapply()` and the like) may also be used to generate large lists of trajectories with some variations. These small pieces can be concatenated together into longer trajectories using `join()`, but at the same time, they allow for multiple points of attachment of arrivals.

During a simulation, trajectories can interact with the simulation environment in order to extract or modify parameters of interest such as the current simulation time, attributes, status of resources (get the number of arrivals in a resource, get or set resources' capacity or queue size), or status of sources (get the number of generated arrivals, set sources' attached trajectory or distribution). The only requirement is that the simulation object must be defined in the same R environment (or a parent one) *before* the simulation is started. Effectively, it is enough to detach the `run()` method from the instantiation (`simmer()`), namely, they should

not be called in the same pipe. But, for the sake of consistency, it is a good coding practice to instantiate the simulation object always in the first place as follows:

```
R> set.seed(1234)
R> env <- simmer()
R>
R> traj <- trajectory() %>%
R+   log_(function() paste0("Current simulation time: ", now(env)))
R>
R> env <- env %>%
R+   add_generator("dummy", traj, at(rexp(1, 1))) %>%
R+   run()

2.50176: dummy0: Current simulation time: 2.50175860496223
```

5. Performance evaluation

This section investigates the performance of **simmer** with the aim of assessing its usability as a general-purpose DES framework. A first section is devoted to measuring the simulation time of a simple model relative to **SimPy** and **SimJulia**. The reader may find interesting to compare the expressiveness of each framework. Last but not least, the final section explores the cost of calling R from C++, revealing the existent trade-off, inherent to the design of this package, between performance and model complexity.

All the subsequent tests were performed under Fedora Linux 25 running on an Intel Core2 Quad CPU Q8400, with R 3.3.3, Python 2.7.13, **SimPy** 3.0.9, Julia 0.5.1 and **SimJulia** 0.3.14 installed from the default repositories. Absolute execution times presented here are specific to this platform and configuration, and thus they should not be taken as representative for any other system. Instead, the relative performance should be approximately constant across different systems.

5.1. Comparison with similar frameworks

A significant effort has been put into the design of **simmer** in order to make it performant enough to run general and relatively large simulation models in a reasonable amount of time. In this regard, a relevant comparison can be made against other general-purpose DES frameworks such as **SimPy** and **SimJulia**. To this effect, we retake the M/M/1 example from Section 4.1, which can be bundled into the following test:

```
R> library("simmer")
R>
R> test_mm1_simmer <- function(n, m, mon=FALSE) {
R>   mm1 <- trajectory() %>%
R>     seize("server", 1) %>%
R>     timeout(function() rexp(1, 1.1)) %>%
R>     release("server", 1)
```

```
R>
R>   env <- simmer() %>%
R>     add_resource("server", 1, mon=mon) %>%
R>     add_generator("customer", mm1, function() rexp(m, 1), mon=mon) %>%
R>     run(until=n)
R> }
```

With the selected arrival rate, $\lambda = 1$, this test simulates an average of `n` arrivals entering a nearly saturated system ($\rho = 1/1.1$). Given that **simmer** generators are able to create arrivals in batches (i.e., more than one arrival for each function call) for improved performance, the parameter `m` controls the size of the batch. Finally, the `mon` flag enables or disables monitoring.

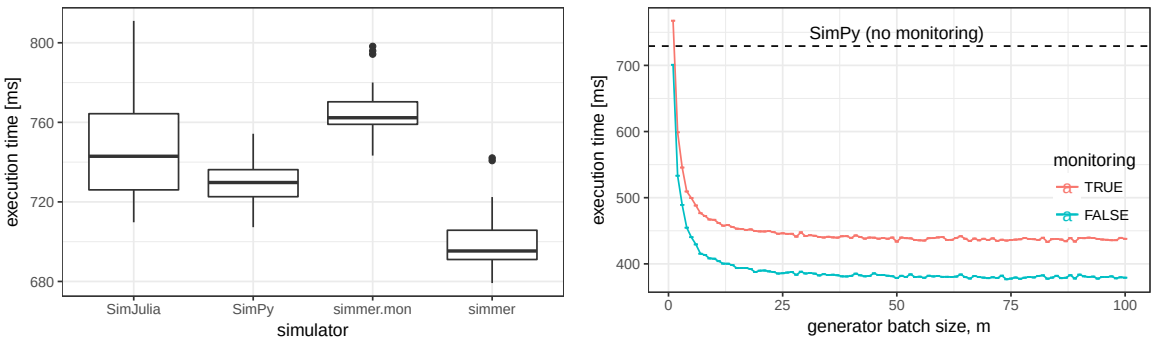
Let us build now the equivalent model using **SimPy**, with base Python for random number generation. We prepare the Python benchmark from R using the **rPython** package (Bellosta 2015) as follows:

```
R> rPython::python.exec("
R> import simpy, random, time
R>
R> def test_mm1(n):
R>   def exp_source(env, lambd, server, mu):
R>       while True:
R>           dt = random.expovariate(lambd)
R>           yield env.timeout(dt)
R>           env.process(customer(env, server, mu))
R>
R>   def customer(env, server, mu):
R>       with server.request() as req:
R>           yield req
R>           dt = random.expovariate(mu)
R>           yield env.timeout(dt)
R>
R>   env = simpy.Environment()
R>   server = simpy.Resource(env, capacity=1)
R>   env.process(exp_source(env, 1, server, 1.1))
R>   env.run(until=n)
R>
R> def benchmark(n, times):
R>   results = []
R>   for i in range(0, times):
R>     start = time.time()
R>     test_mm1(n)
R>     results.append(time.time() - start)
R>   return results
R> ")
```

Equivalently, this can be done for Julia and **SimJulia** using the **rjulia** package (Gong, Keys, and Maechler 2017). Once more, `n` controls the number of arrivals simulated on average:

```
R> rjulia::julia_init()
R> rjulia::julia_void_eval("
R> using SimJulia, Distributions
R>
R> function test_mm1(n::Float64)
R>   function exp_source(env::Environment, lambd::Float64,
R>                       server::Resource, mu::Float64)
R>     while true
R>       dt = rand(Exponential(1/lambd))
R>       yield(Timeout(env, dt))
R>       Process(env, customer, server, mu)
R>     end
R>   end
R>
R>   function customer(env::Environment, server::Resource, mu::Float64)
R>     yield(Request(server))
R>     dt = rand(Exponential(1/mu))
R>     yield(Timeout(env, dt))
R>     yield(Release(server))
R>   end
R>
R>   env = Environment()
R>   server = Resource(env, 1)
R>   Process(env, exp_source, 1.0, server, 1.1)
R>   run(env, n)
R> end
R>
R> function benchmark(n::Float64, times::Int)
R>   results = Float64[]
R>   test_mm1(n)
R>   for i = 1:times
R>     push!(results, @elapsed test_mm1(n))
R>   end
R>   return(results)
R> end
R> ")
```

It can be noted that in both cases there is no monitoring involved, because either **SimPy** nor **SimJulia** provide automatic monitoring as **simmer** does. Furthermore, the resulting code for **simmer** is more concise and expressive than the equivalent ones for **SimPy** and **SimJulia**, which are very similar.



(a) Boxplots for 20 runs of the M/M/1 test with `n=1e4`. (b) Performance evolution with the batch size m.

Figure 4: Performance comparison.

We obtain the reference benchmark with `n=1e4` and 20 replicas for both packages as follows:

```
R> n <- 1e4L
R> times <- 20
R>
R> ref <- data.frame(
R>   SimPy = rPython::python.call("benchmark", n, times),
R>   SimJulia = rjulia::j2r(paste0("benchmark(", n, ".0, ", times, ")"))
R> )
```

As a matter of fact, we also tested a small DES skeleton in pure R provided in (Matloff 2011, *7.8.3 Extended Example: Discrete-Event Simulation in R*). This code was formalised into an R package called **DES**, available on GitHub[4] since 2014. The original code implemented the event queue as an ordered vector which was updated by performing a binary search. Thus, the execution time of this version was two orders of magnitude slower than the other frameworks. The most recent version on GitHub (as of 2017) takes another clever approach though: it supposes that the event vector will be short and approximately ordered; therefore, the event vector is not sorted anymore, and the next event is found using a simple linear search. These assumptions hold for many cases, and particularly for this M/M/1 scenario. As a result, the performance of this model is only ~ 2.2 times slower than **SimPy**. Still, it is clear that pure R cannot compete with other languages in discrete-event simulation, and **DES** is not considered in our comparisons hereafter.

Finally, we set a benchmark for **simmer** using **microbenchmark**, again with `n=1e4` and 20 replicas for each test. Figure 4a shows the output of this benchmark. **simmer** is tested both in monitored and in non-monitored mode. The results show that the performance of **simmer** is equivalent to **SimPy** and **SimJulia**. The non-monitored **simmer** shows a slightly better performance than these frameworks, while the monitored **simmer** shows a slightly worse performance.

At this point, it is worth highlighting **simmer**'s ability to generate arrivals in batches (hence parameter m). To better understand the impact of batched arrival generation, the benchmark was repeated over a range of m values $(1, \ldots, 100)$. The results of the batched arrival

[4] https://github.com/matloff/des

generation runs are shown in Figure 4b. This plot depicts the average execution time of the **simmer** model with (red) and without (blue) monitoring as a function of the generator batch size m. The black dashed line sets the average execution time of the **SimPy** model to serve as a reference.

The performance with `m=1` corresponds to what has been shown in Figure 4a. But as m increases, **simmer** performance quickly improves and becomes ~ 1.6 to 1.9 times faster than **SimPy**. Surprisingly, there is no additional gain with batches greater than 40-50 arrivals at a time, but there is no penalty either with bigger batches. Therefore, it is always recommended to generate arrivals in big batches whenever possible.

5.2. The cost of calling R from C++

The C++ simulation core provided by **simmer** is quite fast, as we have demonstrated, but performance is adversely affected by numerous calls to R. The practice of calling R from C++ is generally strongly discouraged due to the overhead involved. However, in the case of **simmer**, it not only makes sense, but is even fundamental in order to provide the user with enough flexibility to build all kinds of simulation models. Nevertheless, this cost must be known, and taken into account whenever a higher performance is needed.

To explore the cost of calling R from C++, let us define the following test:

```
R> library("simmer")
R>
R> test_simmer <- function(n, delay) {
R+   test <- trajectory() %>%
R+     timeout(delay)
R+
R+   env <- simmer() %>%
R+     add_generator("test", test, at(1:n)) %>%
R+     run(Inf)
R+
R+   arrivals <- get_mon_arrivals(env)
R+ }
```

This toy example performs a very simple simulation in which `n` arrivals are attached (in one shot, thanks to the convenience function `at()`) to a `test` trajectory at $t = 1, 2, ..., n$. The trajectory consists of a single activity: a timeout with some configurable `delay` that may be a fixed value or a function call. Finally, after the simulation, the monitored data is extracted from the simulation core to R. Effectively, this is equivalent to generating a data frame of `n` rows (see the example output in Table 2).

Table 2: Output from the `test_simmer()` function.

Name	Start time	End time	Activity time	Finished	Replication
test0	1	2	1	TRUE	1
test1	2	3	1	TRUE	1
test2	3	4	1	TRUE	1

As a matter of comparison, the following `test_R_for()` function produces the very same data using base R:

```
R> test_R_for <- function(n) {
R>   name <- character(n)
R>   start_time <- numeric(n)
R>   end_time <- numeric(n)
R>   activity_time <- logical(n)
R>   finished <- numeric(n)
R>
R>   for (i in 1:n) {
R>     name[i] <- paste0("test", i-1)
R>     start_time[i] <- i
R>     end_time[i] <- i+1
R>     activity_time[i] <- 1
R>     finished[i] <- TRUE
R>   }
R>
R>   arrivals <- data.frame(
R>     name=name,
R>     start_time=start_time,
R>     end_time=end_time,
R>     activity_time=activity_time,
R>     finished=finished,
R>     replication = 1
R>   )
R> }
```

Note that we are using a `for` loop to mimic the behaviour of **simmer**'s internals, of how monitoring is made, but we concede the advantage of pre-allocated vectors to R. A second base R implementation, which builts upon the `lapply()` function, is implemented as the `test_R_lapply()` function:

```
R> test_R_lapply <- function(n) {
R>   as.data.frame(do.call(rbind, lapply(1:n, function(i) {
R>     list(
R>       name = paste0("test", i - 1),
R>       start_time = i,
R>       end_time = i + 1,
R>       activity_time = 1,
R>       finished = TRUE,
R>       replication = 1
R>     )
R>   })))
R> }
```

The `test_simmer()`, `test_R_for()` and `test_R_lapply()` functions all produce exactly the same data in a similar manner (cfr. Table 2). Now, we want to compare how a delay consisting

of a function call instead of a fixed value impacts the performance of **simmer**, and we use `test_R_for()` and `test_R_lapply()` as yardsticks.

To this end, the **microbenchmark** package (Mersmann 2015) is used. The benchmark was executed with `n=1e5` and 20 replicas for each test. Table 3 shows a summary of the resulting timings. As we can see, **simmer** is ~ 4.4 times faster than `for`-based base R and ~ 3.6 times faster than `lapply`-based base R on average when we set a fixed delay. On the ther hand, if we replace it for a function call, the execution becomes ~ 6.5 times slower, or ~ 1.5 times slower than `for`-based base R. It is indeed a quite good result if we take into account the fact that base R pre-allocates memory, and that **simmer** is doing a lot more internally. But still, these results highlight the overheads involved and encourage the use of fixed values instead of function calls whenever possible.

Table 3: Execution time (milliseconds).

Expr	Min	Mean	Median	Max
test_simmer(n, 1)	429.8663	492.365	480.5408	599.3547
test_simmer(n, function() 1)	3067.9957	3176.963	3165.6859	3434.7979
test_R_for(n)	2053.0840	2176.164	2102.5848	2438.6836
test_R_lapply(n)	1525.6682	1754.028	1757.7566	2002.6634

6. Summary

The **simmer** package presented in this paper brings a generic yet powerful process-oriented Discrete-Event Simulation framework to R. **simmer** combines a robust and fast simulation core written in C++ with a rich and flexible R API. The main modelling component is the *activity*. Activities are chained together with the pipe operator into *trajectories*, which are common paths for processes of the same type. **simmer** provides a broad set of activities, and allows the user to extend their capabilities with custom R functions.

Monitoring is automatically performed by the underlying simulation core, thereby enabling the user to focus on problem modelling. **simmer** enables simple replication and parallelisation with standard R tools. Data can be extracted into R data frames from a single simulation environment or a list of environments, each of which is marked as a different replication for further analysis.

Despite the drawbacks of combining R calls into C++ code, **simmer** shows a good performance combined with high flexibility. It is currently one of the most extensive DES frameworks for R and provides a mature workflow for truly integrating DES into R processes.

Acknowledgements

We thank the editors and the anonymous referee for their thorough reviews and valuable comments, which have been of great help in improving this paper. Likewise, we thank Norman Matloff for his advice and support. Last but not least, we are very grateful for vignette contributions by Duncan Garmonsway, and for all the fruitful ideas for new or extended features by several users via the `simmer-devel` mailing list and GitHub.

References

Bache SM, Wickham H (2014). ***magrittr***: *A Forward-Pipe Operator for R*. R package version 1.5, URL https://CRAN.R-project.org/package=magrittr.

Banks J (2005). *Discrete-event System Simulation*. Prentice-Hall International Series in Industrial and Systems Engineering. Pearson Prentice Hall. ISBN 9780131446793.

Bellosta CJG (2015). ***rPython***: *Package Allowing R to Call Python*. R package version 0.0-6, URL https://CRAN.R-project.org/package=rPython.

Chang W (2017). ***R6***: *Classes with Reference Semantics*. R package version 2.2.2, URL https://CRAN.R-project.org/package=R6.

Chubaty AM, McIntire EJB (2017). ***SpaDES***: *Develop and Run Spatially Explicit Discrete Event Simulation Models*. R package version 2.0.0, URL https://CRAN.R-project.org/package=SpaDES.

Ebert A (2017). ***queuecomputer***: *Computationally Efficient Queue Simulation*. R package version 0.8.1, URL https://CRAN.R-project.org/package=queuecomputer.

Eddelbuettel D (2013). *Seamless R and C++ Integration with* ***Rcpp***. Springer-Verlag, New York, NY, USA. ISBN 978-1-4614-6867-7.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Gong Y, Keys O, Maechler M (2017). ***rjulia***: *Integrating R and Julia – Calling Julia from R*. R package version 0.9-3, URL https://github.com/armgong/rjulia.

Jha V, Bagrodia R (2000). "Simultaneous Events and Lookahead in Simulation Protocols." *ACM Trans. Model. Comput. Simul.*, **10**(3), 241–267. ISSN 1049-3301. doi:10.1145/361026.361032.

Kendall DG (1953). "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain." *The Annals of Mathematical Statistics*, **24**(3), 338–354. doi:10.1214/aoms/1177728975.

Lauwens B (2017). ***SimJulia.jl***: *Combined Continuous-Time / Discrete-Event Process Oriented Simulation Framework Written in Julia*. Julia package version 0.3.14, URL https://github.com/BenLauwens/SimJulia.jl.

Law A, Kelton W (2000). *Simulation Modeling and Analysis*. McGraw-Hill Series in Industrial Engineering and Management Science. McGraw-Hill. ISBN 9780070592926.

Luke Tierney (2016). *A Byte-code Compiler for R*. Department of Statistics and Actuarial Science, University of Iowa. URL https://www.stat.uiowa.edu/~luke/R/compiler/compiler.pdf.

Matloff N (2011). *The Art of R Programming: A Tour of Statistical Software Design*. 1st edition. No Starch Press, San Francisco, CA, USA. ISBN 1593273843, 9781593273842.

Mersmann O (2015). ***microbenchmark**: Accurate Timing Functions*. R package version 1.4-2.1, URL https://CRAN.R-project.org/package=microbenchmark.

Pidd M (1988). *Computer Simulation in Management Science*. John Wiley & Sons. ISBN 9780471919315.

Rönngren R, Liljenstam M (1999). "On Event Ordering in Parallel Discrete Event Simulation." In *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, PADS '99, pp. 38–45. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-0155-9.

Shannon RE (1975). *Systems Simulation: The Art and Science*. Prentice-Hall. ISBN 9780138818395.

Team SimPy (2017). ***SimPy**: Discrete-Event Simulation for Python*. Python package version 3.0.9, URL https://simpy.readthedocs.io/en/latest/.

Ucar I, Smeets B (2017a). ***simmer**: Discrete-Event Simulation for R*. R package version 3.6.4, URL https://CRAN.R-project.org/package=simmer.

Ucar I, Smeets B (2017b). ***simmer.plot**: Plotting Methods for **simmer***. R package version 0.1.11, URL https://CRAN.R-project.org/package=simmer.plot.

Wickham H, Francois R (2017). ***dplyr**: A Grammar of Data Manipulation*. R package version 0.7.4, URL https://CRAN.R-project.org/package=dplyr.

Affiliation:

Iñaki Ucar
Universidad Carlos III de Madrid
Avda. de la Universidad, 30 28911 Leganés (Madrid) Spain
E-mail: iucar@fedoraproject.org

Bart Smeets
dataroots
Witte Patersstraat, 4 1040 Brussels Belgium
E-mail: bart@dataroots.io

Arturo Azcorra
Universidad Carlos III de Madrid
IMDEA Networks Institute
Avda. de la Universidad, 30 28911 Leganés (Madrid) Spain
E-mail: azcorra@it.uc3m.es