simmer: Discrete-Event Simulation for R

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.

1. Introduction

A discrete-event simulation (DES) models a system as a discrete sequence of events. The execution of an event at a given time will lead to a change in the system's state. One can configure the combination of events in a system in such a way that it can *simulate* an actual process. The applications of DES-based simulation are broad. If used to simulate a process one can for example gain insights into the process' risk, efficiency and effectiveness. Also, by simulation of an alternative configuration of a process, one can proactively estimate the benefits of changes to the process. This in turn allows one to get clear insights into the benefits of process redesign strategies (e.g., extra resources).

simmer is a process-oriented and trajectory-based discrete-event simulation package for R. Designed to be a generic framework, it leverages the power of **Rcpp** to boost performance and make DES modelling in R not only effective but also efficient. As a noteworthy characteristic, **simmer** exploits the concept of trajectory: a common path in the simulation model for entities of the same type. As a modelling framework it is flexible and simple to use, and leverages the chaining/piping workflow introduced by the **magrittr** package.

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.

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 2016) focuses on spatially explicit discrete models, and the **queuecomputer** package

(Ebert 2016) 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 respectively the Python and Julia languages.

2. The simulation core design

The core of any discrete-event simulator comprises two main components: an event list, ordered by time of occurrence, and an event loop that executes the routines associated to each event. 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 (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 2016) 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 subsections 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 *simultaneity problem*, an important topic that every DES framework has to deal with.

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 Fishman (2001), 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.

[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 has been announced to be included in R version 3.4.0 (Revolutions 2017).

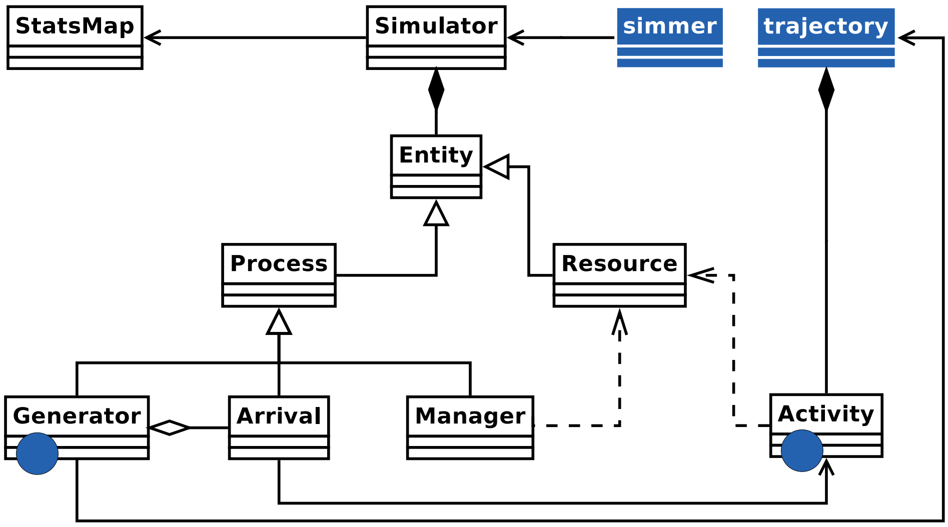


Figure 1: UML diagram of the simulation core architecture. Blue classes represent how R encapsulates the C++ core. Blue circles represent how C++ interfaces with R.

Generator 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. Architectural choices

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 the modelling capabilities, and vice versa. The following lines are devoted to discuss how this trade-off is resolved in **simmer**.

Figure 1 sketches a Unified Modelling Language (UML) 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 list, which is implemented as a priority queue ordered by 1) time of occurrence and 2) priority[2]. Apart from those two attributes, each event holds a pointer to a process. The `Simulator` provides methods for scheduling and unscheduling events. Moreover, it is responsible for managing simulation-wide entities (e.g., resources and generators) and facilities (e.g., signaling between processes, global attributes and monitoring).

Thus, monitoring counters, which are derived from the `StatsMap` 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:

- 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 subsection, there are two types of entities: passive ones (`Resource`) and active ones (processes `Generator`, `Arrival` and `Manager`). Generators 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 1, 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. All the activities that the **simmer** core provides derive from this one.

Finally, it is worth mentioning the couple of blue circles depicted in Figure 1. 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]The priority aspect is important to tackle the *simultaneity problem* which will be addressed in the next subsection.

2.3. The simultaneity problem

As noted by (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 (Rönngren and Liljenstam 1999, Jha and Bagrodia (2000)).

As an example of the implications, let us consider a simple case. At time t_{i-1}, there is one arrival seizing a resource, 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.

Priority	Event
`PRIORITY_MAX`	Modify a generator (e.g., activate or deactivate it)
`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)
`PRIORITY_GENERATOR`	Generate new arrivals
...	General activities
`PRIORITY_MIN`	Other tasks (e.g., a timer for reneging)

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

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 1 (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 (Bache and Wickham 2014).

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 2016). 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 }
{ 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()`, this is the `value` parameter.

If we want to set an attribute that depends on another attribute, or on the current value of the attribute to be set, this is possible. In fact, if a function with one parameter is supplied, the current set of attributes for that arrival is passed as a named vector to that function. For instance, the following trajectory prints `81`:

```
R> traj <- trajectory() %>%
R+   set_attribute("weight", 80) %>%
R+   set_attribute("weight", function(attr) attr["weight"] + 1) %>%
R+   log_(function(attr) paste0("My weight is ", attr["weight"]))
```

In general, whenever an activity accepts a function as a parameter, the rule above applies, and we can obtain the current set of attributes as the first argument of that function. If the supplied function has no parameters, it is evaluated in the same way, but the attribute vector is not accessible in the body.

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 generator, but they can also be dynamically changed on a per-arrival basis using the `set_prioritization()` activity.

```
R> traj <- trajectory() %>%
R+   set_prioritization(c(1, 7, TRUE)) %>%
R+   set_attribute("priority", 3) %>%
R+   set_prioritization(function(attrs) c(attrs["priority"], 7, TRUE))
```

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()`.

```
R> env <- simmer()
R>
R> increment <- function(res) {
R+   function() get_capacity(env, res) + 1
R+ }
R>
R> traj <- trajectory() %>%
R+   seize("resource_name", amount=1) %>%
R+   set_capacity("resource_name", increment("resource_name")) %>%
R+   timeout(function() rexp(1, 10)) %>%
R+   release("resource_name", amount=1)
```

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_traj <- trajectory("patient_trajectory") %>%
R+   log_("arriving...") %>%
R+   seize("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+   timeout(5) %>%
R+   release("doctor", 1)
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_traj, at(0, 1)) %>%
R+   run()

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

The value supplied to all these methods is a dynamical parameter. On the other hand, the resource name must be fixed. There is a special mechanism for dynamically selecting resources: 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 a few 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.

```
R> patient_traj <- trajectory("patient trajectory") %>%
R+   select(resources = c("doctor1", "doctor2", "doctor3"),
R+          policy = "round-robin") %>%
R+   set_capacity_selected(1) %>%
R+   seize_selected(amount = 1) %>%
R+   timeout(5) %>%
R+   release_selected(amount = 1)
```

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

Interaction with generators

There are four activities specifically intended to modify generators. An arrival may `activate()` or `deactivate()` a generator, but also modify with `set_trajectory()` the trajectory to which it attaches the arrivals created, or set a new interarrival distribution with `set_distribution()`. For dynamically selecting a generator, the parameter that specifies the generator 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(option = function() round(now(env)), continue = c(FALSE, TRUE),
R+          trajectory() %>%
R+            log_("branch 1"),
R+          trajectory() %>%
R+            log_("branch 2")
R+   ) %>%
R+   clone(n = 3,
R+         trajectory() %>%
R+           log_("clone 0"),
R+         trajectory() %>%
R+           log_("clone 1"),
R+         trajectory() %>%
R+           log_("clone 2")) %>%
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: clone 2
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> traj <- trajectory() %>%
R+   log_("Hello!") %>%
R+   timeout(1) %>%
R+   rollback(amount = 2, times = 2)
R>
R> simmer() %>%
R+   add_generator("hello_sayer", traj, 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[3]. 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> traj <- 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.

[3] 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`[4] 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("you shall pass",
R+        handler = trajectory() %>%
R+          log_("got a signal!")) %>%
R+   log_("waiting...") %>%
R+   wait() %>%
R+   log_("continuing!")
R>
R> t_signaler <- 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_signaler, 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 an arrival abandons the trajectory at a given moment. The most simple activity enabling

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

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_traj <- trajectory("bank") %>%
R+   log_("Here I am") %>%
R+   renege_in(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_traj, 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 generators, and controls the simulation execution. The `simmer()` method instantiates the object, after which resources and generators can be appended using the `%>%` operator:

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

```
simmer environment: anonymous | now: 0 | next: 0
{ Resource: res_name | monitored: 1 | server status: 0(1) | queue status...
{ Generator: arrival | monitored: 1 | n_generated: 0 }
```

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

```
R> env %>%
R+   run(until=40)

25: arrival0: Entering the trajectory
35: arrival0: Leaving the trajectory

simmer environment: anonymous | now: 40 | next: 50
{ Resource: res_name | monitored: 1 | server status: 0(1) | queue status...
{ Generator: 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' parameters (capacity, queue size, server count and queue count) and generators' parameters (number of arrivals generated so far)[5].

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. Resources are defined with the following signature:

```
add_resource(.env, name, capacity = 1, queue_size = Inf, mon = TRUE,
             preemptive = FALSE, preempt_order = c("fifo", "lifo"),
             queue_size_strict = FALSE)
```

That is to say, with the `name` of the resource, the `capacity` of the server and the `queue_size` (0 means no queue). Resources are monitored and are 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.

[5] All these methods can also be used at run-time inside trajectories if the simulation environment is in the scope of these trajectories.

Generators

Generators are defined with the following signature:

```
add_generator(.env, name_prefix, trajectory, distribution, mon = 1,
              priority = 0, preemptible = priority, restart = FALSE)
```

The `add_generator()` function expects a `name_prefix` for each generated arrival, a trajectory to attach them to and an interarrival `distribution`. Parameters `priority`, `preemptible` and `restart` have been described in Section 3.1.1. The monitoring flag accepts several levels in this case:

0. No monitoring enabled.
1. Arrival monitoring.
2. Level 1 + attribute monitoring.

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. They do so with zero-delay and re-schedule themselves with a delay equal to the sum of the values obtained. Whenever a negative interarrival value is obtained, the generator stops.

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

This section aims to provide some modelling examples covering the tools we have described. The topics addressed are queuing systems, replication, parallelisation and a couple of more advanced examples using a broader selection of activities.

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 generators, 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.

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

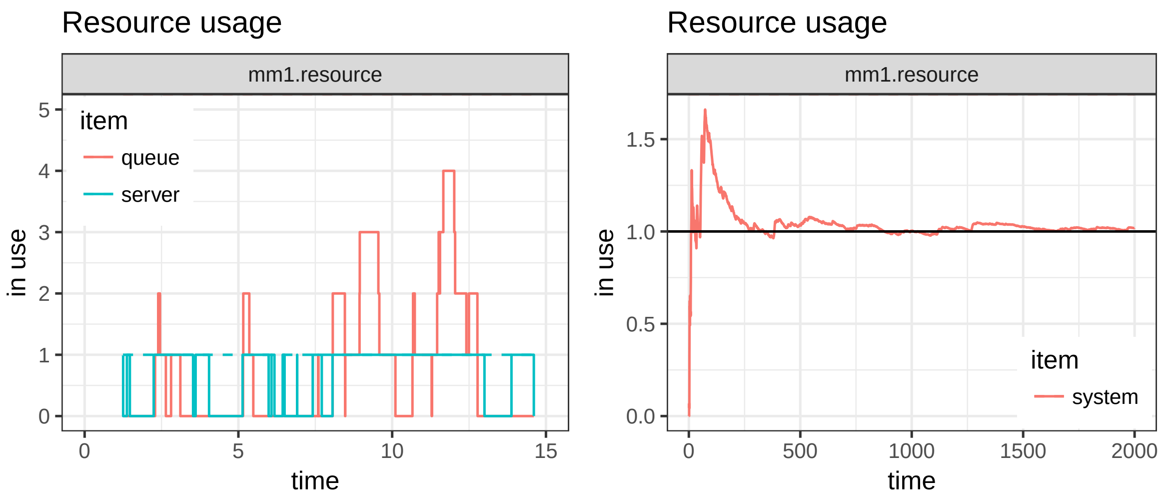


Figure 2: Detail of the resource usage (left) and convergence over time (right).

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 2017) provides convenience plotting methods to, for instance, quickly visualise the usage of a resource over time. Figure 2 gives a 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+   dplyr::group_by(replication) %>%
R+   dplyr::summarise(mean = mean(end_time - start_time))
R>
R> t.test(mm1.data[["mean"]])


	One Sample t-test

data:  mm1.data[["mean"]]
t = 94.883, df = 99, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.4925143 0.5135535
sample estimates:
mean of x
0.5030339
```

4.3. Advanced examples

The following examples are taken from the **SimPy** documentation[6], and they illustrate a broader selection of modelling techniques. The first one, *Machine Shop*, covers the use of attributes, loops, preemptive resources and resource selection. The second one, *Movie Renege*, additionally covers the use of branching, shared events, reneging and advanced uses of resources (namely, the execution of an alternative sub-trajectory if resource seizing is rejected).

Machine shop

> This example comprises a workshop with `n` identical machines. A stream of jobs (enough to keep the machines busy) arrives. Each machine breaks down periodically. Repairs are carried out by one repairwoman. The repairwoman has other, less important, tasks to perform too. Broken machines preempt theses tasks. The repairwoman continues them when she is done with the machine repairment. The workshop works continuously.

[6] https://simpy.readthedocs.io/en/latest/examples

First of all, we set up the environment:

```
R> library(simmer)
R>
R> PT_MEAN <- 10.0
R> PT_SIGMA <- 2.0
R> MTTF <- 300.0
R> BREAK_MEAN <- 1 / MTTF
R> REPAIR_TIME <- 30.0
R> JOB_DURATION <- 30.0
R> NUM_MACHINES <- 10
R> WEEKS <- 4
R> SIM_TIME <- WEEKS * 7 * 24 * 60
R>
R> set.seed(42)
R> env <- simmer()
```

where `PT_MEAN`, `PT_SIGMA` are the average processing time and its standard deviation in minutes, `MTFF` is the mean time to failure in minutes, `BREAK_MEAN` is the failure rate, `REPAIR_TIME` is the time it takes to repair a machine in minutes, `JOB_DURATION` is the duration of other jobs in minutes, `NUM_MACHINES` is the number of machines in the machine shop, `WEEKS` is the simulation time in weeks, and `SIM_TIME` is the simulation time in minutes.

The `make_parts` trajectory defines a machine's operating loop. A worker seizes the machine and starts manufacturing and counting parts in an infinite loop. Provided that we want more than one machine, we parametrise the trajectory as a function of the machine's name:

```
R> make_parts <- function(machine)
R+   trajectory() %>%
R+     set_attribute("parts", 0) %>%
R+     seize(machine, 1) %>%
R+     timeout(function() rnorm(1, PT_MEAN, PT_SIGMA)) %>%
R+     set_attribute("parts", function(attr) attr["parts"] + 1) %>%
R+     rollback(2, Inf)
```

The final `rollback()` performs a indefinite loop over the previous two activities. The repairwoman's *unimportant* jobs may be modelled in the same way (without the accounting part):

```
R> other_jobs <- trajectory() %>%
R+   seize("repairwoman", 1) %>%
R+   timeout(JOB_DURATION) %>%
R+   rollback(1, Inf)
```

Failures are modelled as high-priority arrivals, both for the machines and for the repairwoman. Each random generated failure will randomly select and seize (break) a machine, and will seize (call) the repairwoman. After the machine is repaired, both resources are released, and the corresponding workers continue where they were stopped.

```
R> machines <- paste0("machine", 1:NUM_MACHINES-1)
R>
R> failure <- trajectory() %>%
R+   select(machines, policy = "random") %>%
R+   seize_selected(1) %>%
R+   seize("repairwoman", 1) %>%
R+   timeout(REPAIR_TIME) %>%
R+   release("repairwoman", 1) %>%
R+   release_selected(1)
```

Down below, the machines (resources) and their workers (generators) are appended to the simulation environment. Note that each machine, which is defined as preemptive, has space for one worker (or a failure) and no space in its queue. The same applies for the repairwoman, but this time the queue is infinite, given that there could be any number of machines at any time waiting for repairments. Finally, the failure generator is defined with `priority=1` (default: 0), and the simulation begins:

```
R> for (i in machines){
R+   env %>%
R+   add_resource(i, 1, 0, preemptive = TRUE) %>%
R+   add_generator(paste0(i, "_worker"), make_parts(i), at(0), mon = 2)
R+ }
R>
R> env %>%
R+   add_resource("repairwoman", 1, Inf, preemptive = TRUE) %>%
R+   add_generator("repairwoman_worker", other_jobs, at(0)) %>%
R+   add_generator("failure", failure,
R+                 function() rexp(1, BREAK_MEAN * NUM_MACHINES),
R+                 priority = 1) %>%
R+   run(SIM_TIME) %>% invisible
```

The last value per worker, from the table of attributes, reports the number of parts made for each machine. The result is shown in Table 2.

```
R> last_values <-
R+   get_mon_attributes(env) %>%
R+   dplyr::group_by(name) %>%
R+   dplyr::slice(n()) %>%
R+   dplyr::arrange(name)
```

Time	Name	Key	Value	Replication
40312.79	machine0_worker0	parts	3234	1
40313.02	machine1_worker0	parts	3369	1
40317.91	machine2_worker0	parts	3345	1
40315.71	machine3_worker0	parts	3269	1

Time	Name	Key	Value	Replication
40311.65	machine4_worker0	parts	3310	1
40314.01	machine5_worker0	parts	3266	1
40316.61	machine6_worker0	parts	3371	1
40314.26	machine7_worker0	parts	3386	1
40315.19	machine8_worker0	parts	3216	1
40318.83	machine9_worker0	parts	3401	1

Table 2: Number of parts made for each machine.

Movie renege

> This example models a movie theater with one ticket counter selling tickets for three movies (next show only). People arrive at random times and try to buy a random number (1-6) of tickets for a random movie. When a movie is sold out, all people waiting to buy a ticket for that movie renege (leave the queue).

First of all, we set up the environment:

```
R> library(simmer)
R>
R> TICKETS <- 50
R> SIM_TIME <- 120
R> movies <- c("R Unchained", "Kill Process", "Pulp Implementation")
R>
R> set.seed(42)
R> env <- simmer()
```

where `TICKETS` is the number of tickets per movie and `SIM_TIME` is the simulation time.

The main actor of this simulation is the *moviegoer*, a process that will try to buy a number of tickets for a certain movie. The logic behind a *moviegoer* trajectory is as follows:

1. **Select** a movie and go to the theater.
2. At any moment, the *moviegoer* **reneges if** the movie becomes sold out (the "sold out" event is received).
3. In particular, **leave** immediately if the movie already became sold out.
4. Reach the ticket counter and stay in the queue. When the turn comes (counter is **seized**),
    1. Try to buy (**seize** the resource) tickets (randomly chosen between 1 and 6).
        - If there are not enough tickets, the *moviegoer* is **rejected** after some discussion.
    2. Provided the *moviegoer* has tickets, the reneging condition is **aborted**.
    3. Check the tickets available and, if at most one ticket is left…
        - **Set** instant rejection for future customers (at point 3.).

  - **Send** the "sold out" event (subscribed at point 2.) for current customers (waiting at point 4.).

5. **Release** the ticket counter after the duration of the purchase.
6. Enjoy the movie!

This recipe is directly translated into the following trajectory:

```
R> moviegoer <- trajectory() %>%
R+   set_attribute("movie", function() sample(3, 1)) %>%
R+   select(function(attr) movies[attr["movie"]]) %>%
R+   renege_if(function(attr) paste0(movies[attr["movie"]], " sold out")) %>%
R+   leave(function(attr) get_capacity(env, movies[attr["movie"]]) == 0) %>%
R+   seize("counter", 1) %>%
R+   seize_selected(
R+     function() sample(6, 1), continue = FALSE,
R+     reject = trajectory() %>%
R+       timeout(0.5) %>%
R+       release("counter", 1)
R+   ) %>%
R+   renege_abort() %>%
R+   branch(
R+     function(attr)
R+       get_server_count(env, movies[attr["movie"]]) > (TICKETS - 2),
R+     continue = TRUE,
R+     trajectory() %>%
R+       set_capacity_selected(0) %>%
R+       send(function(attr) paste0(movies[attr["movie"]], " sold out"))
R+   ) %>%
R+   timeout(1) %>%
R+   release("counter", 1) %>%
R+   wait()
```

The next step is to add the required resources to the simulation environment: the three movies (as resources, with `capacity=TICKETS` and no queue) and the ticket counter (another resource with `capacity=1` and an infinite queue). Finally, we attach an exponential *moviegoer* generator to the corresponding trajectory and the simulation starts:

```
R> for (i in movies){
R+   env %>%
R+   add_resource(i, TICKETS, 0)
R+ }
R>
R> env %>%
R+   add_resource("counter", 1, Inf) %>%
R+   add_generator("moviegoer", moviegoer,
R+                 function() rexp(1, 1 / 0.5), mon=2) %>%
R+   run(SIM_TIME)
```

We are interested in the sold out time for each movie after the ticket counter opening, and in how many people left the queue when this event happened. The analysis can be performed by cleverly merging the information provided by **simmer**. First, we get the rows with the sold-out instants, then we get the arrivals that left at these instants and count the number of arrivals per movie. Finally, this information is merged and printed:

```
R> sold_time <- get_mon_resources(env) %>%
R+   dplyr::filter(resource != "counter" & capacity == 0)
R>
R> n_reneges <- dplyr::left_join(
R+   get_mon_arrivals(env) %>%
R+     dplyr::filter(finished == FALSE & end_time %in% sold_time$time),
R+   get_mon_attributes(env)
R+ ) %>%
R+   dplyr::mutate(resource = movies[value]) %>%
R+   dplyr::group_by(resource) %>%
R+   dplyr::count()
R>
R> invisible(apply(dplyr::left_join(sold_time, n_reneges), 1, function(i) {
R+   cat("Movie '", i["resource"], "' was sold out in ", i["time"],
R+       " minutes.\n", "  Number of people that left the queue: ",
R+       i["n"], "\n", sep="")
R+ }))
```

```
Movie 'Pulp Implementation' was sold out in 31.09917 minutes.
  Number of people that left the queue: 18
Movie 'R Unchained' was sold out in 40.59917 minutes.
  Number of people that left the queue: 14
Movie 'Kill Process' was sold out in 41.59917 minutes.
  Number of people that left the queue: 10
```

5. Performance evaluation

This section investigates the performance of **simmer** relative to R itself, **SimPy** and **SimJulia**. All the subsequent tests were performed under Fedora Linux 25 running on an Intel Core2 Quad CPU Q8400, with R 3.3.2, Python 2.7.13, **SimPy** 3.0.9, Julia 0.5.0 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. The cost of calling R from C++

The C++ simulation core provided by **simmer** is quite fast, as we will demonstrate, but the 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.

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 3).

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
test3	4	5	1	TRUE	1
test4	5	6	1	TRUE	1

Table 3: Output from the `test_simmer()` function.

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(name=name,
R+                          start_time=start_time,
R+                          end_time=end_time,
R+                          activity_time=activity_time,
R+                          finished=finished,
R+                          replication = 1)
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 3). 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 4 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.

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

Table 4: Execution time (milliseconds).

5.2. Comparison with other frameworks

A relevant comparison can also 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 2015):

```
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> ")
```

Once more, `n` controls the number of arrivals simulated on average. Note that in both cases there is no monitoring involved, because **SimPy** or **SimJulia** do not provide automatic monitoring as **simmer** does. We obtain the reference benchmark with `n=1e4` and 20 replicas for both packages as follows:

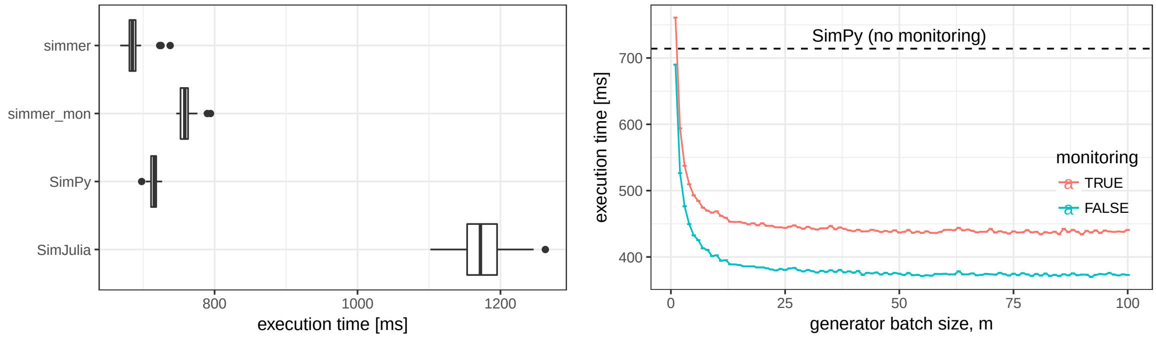
```
R> n <- 1e4L
R> times <- 20
R>
R> ref <- data.frame(
R>   SimPy = rPython::python.call("benchmark", n, times),
R>   SimJulia = rjulia::j2r(paste0("benchmark(", n, ".0, ", times, ")"))
R> )
```

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*), and the execution time for the same simulation model was two orders of magnitude slower than **SimPy** or **SimJulia**, so it is not considered hereafter. Finally, we set a benchmark for **simmer** using **microbenchmark**, again with `n=1e4` and 20 replicas for each test.

Figure 3a shows the output of the benchmark. **simmer** is tested both in monitored and in non-monitored mode. The results show that the performance of **simmer** is equivalent to **SimPy**, which is ~ 1.6 times faster than **SimJulia**. The non-monitored **simmer** shows a slightly better performance than **SimPy**, 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 generation runs are shown in Figure 3b. 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 3a. But as m increases, **simmer** performance quickly improves and becomes ~ 1.6 to 1.9 times faster than **SimPy** and ~ 2.7 to 3.1 times faster than **SimJulia**. Surprisingly, there is no additional gain



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

Figure 3: Performance comparison.

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.

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.

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`.

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 (2016). ***R6**: Classes with Reference Semantics*. R package version 2.2.0, URL `https://CRAN.R-project.org/package=R6`.

Chubaty AM, McIntire EJB (2016). ***SpaDES**: Develop and Run Spatially Explicit Discrete Event Simulation Models*. R package version 1.3.1, URL https://CRAN.R-project.org/package=SpaDES.

Ebert A (2016). ***queuecomputer**: Computationally Efficient Queue Simulation*. R package version 0.6.1, URL https://CRAN.R-project.org/package=queuecomputer.

Eddelbuettel D (2013). *Seamless R and C++ Integration with **Rcpp***. Springer-Verlag, New York. 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. URL http://www.jstatsoft.org/v40/i08/.

Fishman G (2001). *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer-Verlag New York, New York, NY. ISBN 1441928928.

Gong Y, Keys O, Maechler M (2015). ***rjulia**: Integrating R and Julia – Calling Julia from R*. R package version 0.9-3.

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. URL http://doi.acm.org/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." *Ann. Math. Statist.*, **24**(3), 338–354. doi:10.1214/aoms/1177728975. URL http://dx.doi.org/10.1214/aoms/1177728975.

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

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.

Revolutions M (2017). "Performance Improvements Coming to R 3.4.0." URL http://blog.revolutionanalytics.com/2017/02/preview-r-340.html.

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. URL http://dl.acm.org/citation.cfm?id=301429.301456.

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

Ucar I, Smeets B (2017). ***simmer.plot**: Plotting Methods for **simmer**.* R package version 0.1.5, URL http://r-simmer.org/extensions/plot.

Wickham H, Francois R (2016). ***dplyr**: A Grammar of Data Manipulation.* R package version 0.5.0, 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: inaki.ucar@uc3m.es

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

Arturo Azcorra
Universidad Carlos III de Madrid
Avda. de la Universidad, 30 28911 Leganés (Madrid) Spain
and
IMDEA Networks Institute
Avda. del Mar Mediterráneo, 22 28918 Leganés (Madrid) Spain
E-mail: azcorra@it.uc3m.es