simmer

Discrete-Event Simulation for R

Iñaki Úcar, postdoctoral fellow @ UC3M-Santander Big Data Institute (IBiDat)
Bart Smeets, data scientist @ dataroots.io

November 15-16, 2019

Simulation

From R. 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 behavior of the system** or of **evaluating various strategies** [...] for the operation of the system.

Simulation

From R. 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 behavior of the system** or of **evaluating various strategies** [...] for the operation of the system.

Taxonomy, from Law and Kelton (2000):

1. deterministic vs. stochastic
2. (time component?) static vs. dynamic
3. (if dynamic) continuous vs. discrete

Simulation

From R. 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 behavior of the system** or of **evaluating various strategies** [...] for the operation of the system.

Taxonomy, from Law and Kelton (2000):

1. deterministic vs. stochastic
2. (time component?) static vs. dynamic
3. (if dynamic) continuous vs. discrete

Examples:

- deterministic + dynamic + continuous = *Dynamical Systems*
- stochastic + static = *Monte Carlo Simulation*
- **stochastic + dynamic + discrete = *Discrete-Event Simulation* (DES)**

Simulation

DES

What can be modelled as a Discrete-Event Simulation (DES)?

- An **event** is an instantaneous occurrence that may change the **state of the system**
- The number of events is countable
- Between events, all the state variables remain constant
- Output: snapshots of the state of the system over time

What can be modelled as a Discrete-Event Simulation (DES)?

- An **event** is an instantaneous occurrence that may change the **state of the system**
- The number of events is countable
- Between events, all the state variables remain constant
- Output: snapshots of the state of the system over time

Common examples:

- customers arriving at a bank,
- products being manipulated in a supply chain,
- packets traversing a network,
- …

and many more applications from manufacturing systems, construction engineering, project management, logistics, transportation systems, business processes, healthcare, telecommunications networks…

Simulation

DES

Programming styles (Banks 2005):

- **Activity-oriented**: fixed time increments; scan and verify conditions
- **Event-oriented**: event routines; event list ordered by time of ocurrence
- **Process-oriented**: life cycle of entities of the real system; triggered by events

Introduction

Simulation

DES

Programming styles (Banks 2005):

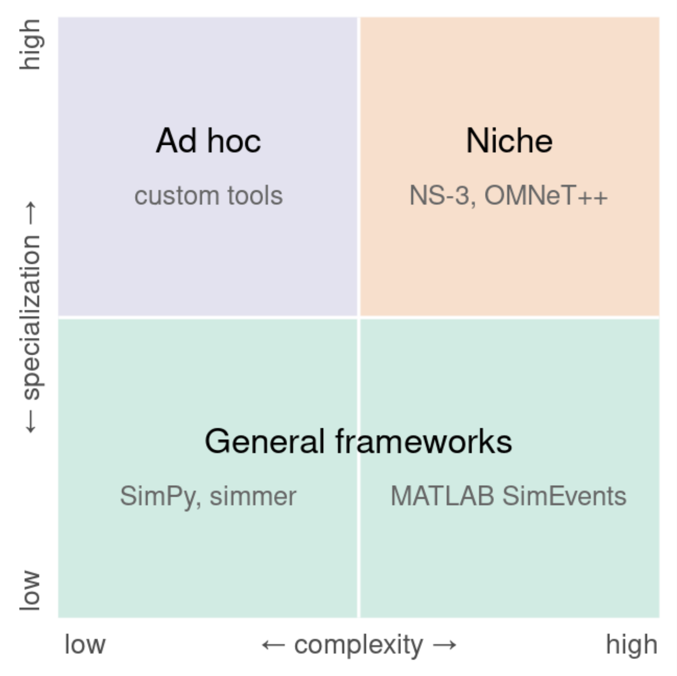
- **Activity-oriented**: fixed time increments; scan and verify conditions
- **Event-oriented**: event routines; event list ordered by time of ocurrence
- **Process-oriented**: life cycle of entities of the real system; triggered by events

Spectrum of tools:

- High **complexity** and **specialization** generally means more accuracy

but

- More specialization requires more effort
- More complexity requires more effort



Introduction

Simulation

DES

`simmer`

Main characteristics:

- General versatile framework for fast prototyping
- Rich and user-friendly R API over a fast C++ simulation core
- Process-oriented **trajectory-based** modelling
- Automatic monitoring capabilities
- Integration with R: repeatability, analysis, visualization

Introduction

Simulation

DES

`simmer`

Main characteristics:

- General versatile framework for fast prototyping
- Rich and user-friendly R API over a fast C++ simulation core
- Process-oriented **trajectory-based** modelling
- Automatic monitoring capabilities
- Integration with R: repeatability, analysis, visualization

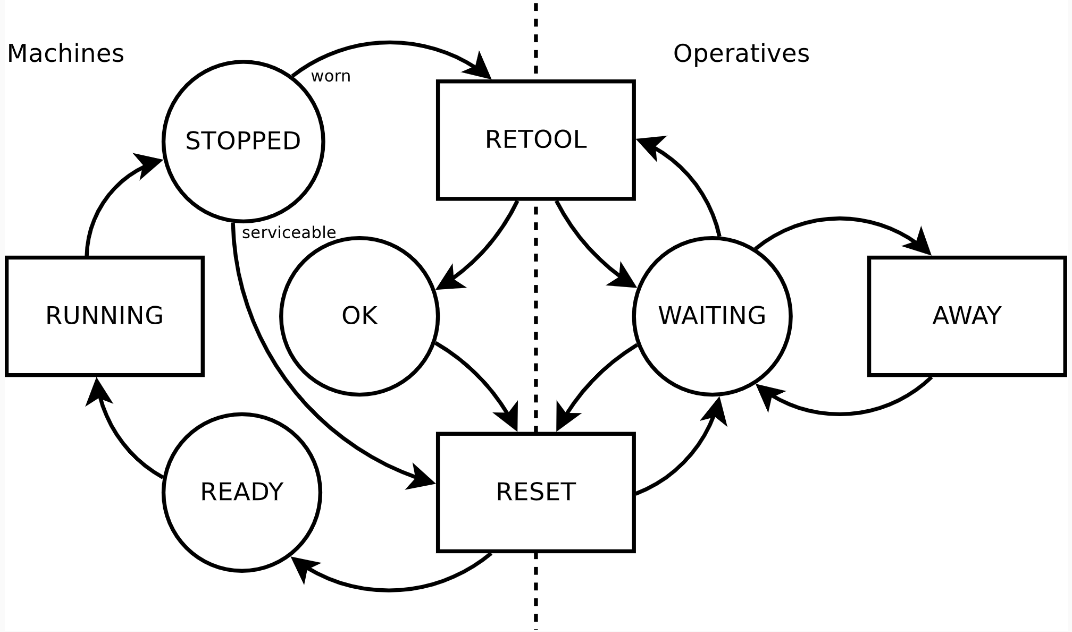
Resources:

- Online documentation (manual + 10 vignettes): https://r-simmer.org
- **Ucar I**, Smeets B, Azcorra A (2019). “simmer: Discrete-Event Simulation for R.” *Journal of Statistical Software*, 90(2), 1-30. doi: 10.18637/jss.v090.i02.
- **Ucar I**, Hernández JA, Serrano P, Azcorra A (2018). “Design and Analysis of 5G Scenarios with simmer: An R Package for Fast DES Prototyping.” *IEEE Communications Magazine*, 56(11), 145-151. doi: 10.1109/MCOM.2018.1700960.

Example: Job Shop

From M. Pidd (1988), Section 5.3.1:

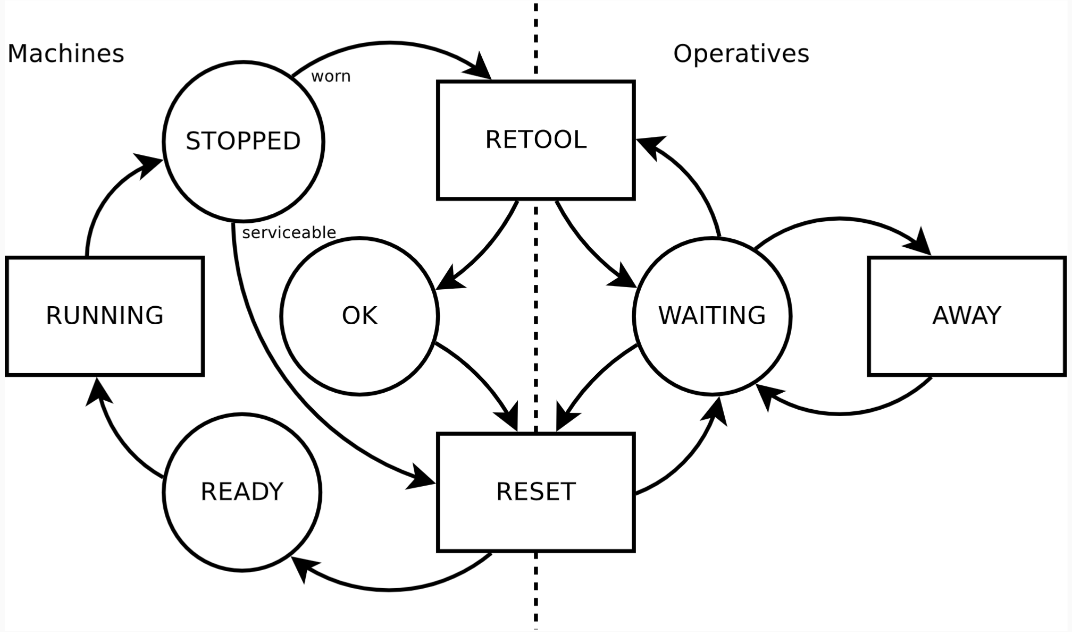
- Jobs are allocated to the first available machine.
- Machines need to be retooled (sometimes) and reset by an operative.
- Operatives may be away attending other tasks.



Example: Job Shop

From M. Pidd (1988), Section 5.3.1:

- Jobs are allocated to the first available machine.
- Machines need to be retooled (sometimes) and reset by an operative.
- Operatives may be away attending other tasks.

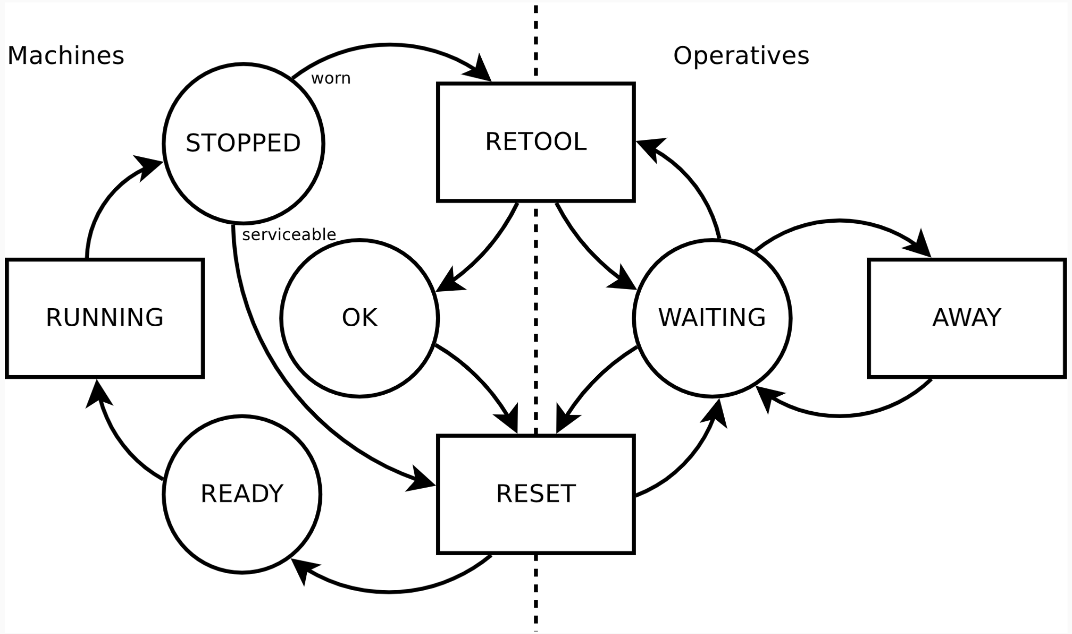



```r
job <- trajectory() %>%
  seize("machine") %>%
  timeout(RUNNING) %>%
  branch(
    CHECK_WORN, continue = TRUE,
    trajectory() %>%
      seize("operative") %>%
      timeout(RETOOL) %>%
      release("operative")
  ) %>%
  seize("operative") %>%
  timeout(RESET) %>%
  release("operative") %>%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

Example: Job Shop

From M. Pidd (1988), Section 5.3.1:

- Jobs are allocated to the first available machine.
- Machines need to be retooled (sometimes) and reset by an operative.
- Operatives may be away attending other tasks.




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

```
task <- trajectory() %>%
  seize("operative") %>%
  timeout(AWAY) %>%
  release("operative")
```

Example: Job Shop

```r
library(simmer); set.seed(1234)

RUNNING ← function() rexp(1, 1)
CHECK_WORN ← function() runif(1) < 0.2
RETOOL ← function() rexp(1, 2)
RESET ← function() rexp(1, 3)
AWAY ← function() rexp(1, 1)
```

```r
job ← trajectory() %>%
  seize("machine") %>%
  timeout(RUNNING) %>%
  branch(
    CHECK_WORN, continue = TRUE,
    trajectory() %>%
      seize("operative") %>%
      timeout(RETOOL) %>%
      release("operative")
  ) %>%
  seize("operative") %>%
  timeout(RESET) %>%
  release("operative") %>%
  release("machine")
```

```r
task ← trajectory() %>%
  seize("operative") %>%
  timeout(AWAY) %>%
  release("operative")
```

Example: Job Shop

```r
library(simmer); set.seed(1234)

RUNNING <- function() rexp(1, 1)
CHECK_WORN <- function() runif(1) < 0.2
RETOOL <- function() rexp(1, 2)
RESET <- function() rexp(1, 3)
AWAY <- function() rexp(1, 1)

job <- trajectory() %>%
  ...
task <- trajectory() %>%
  ...

NEW_JOB <- function() rexp(1, 5)
NEW_TASK <- function() rexp(1, 1)
```

Example: Job Shop

```r
library(simmer); set.seed(1234)

RUNNING ← function() rexp(1, 1)
CHECK_WORN ← function() runif(1) < 0.2
RETOOL ← function() rexp(1, 2)
RESET ← function() rexp(1, 3)
AWAY ← function() rexp(1, 1)

job ← trajectory() %>%
  ...
task ← trajectory() %>%
  ...

NEW_JOB ← function() rexp(1, 5)
NEW_TASK ← function() rexp(1, 1)
```

```r
env ← simmer("Job Shop") %>%
  add_resource("machine", 10) %>%
  add_resource("operative", 5) %>%
  add_generator("job", job, NEW_JOB) %>%
  add_generator("task", task, NEW_TASK) %>%
  run(until=1000)
env
```

```
## simmer environment: Job Shop | now: 1000 | next: 1000.118
## { Monitor: in memory }
## { Resource: machine | monitored: TRUE | server status: 9(
## { Resource: operative | monitored: TRUE | server status:
## { Source: job | monitored: 1 | n_generated: 4954 }
## { Source: task | monitored: 1 | n_generated: 1041 }
```

Example: Job Shop

```r
library(simmer); set.seed(1234)

RUNNING ← function() rexp(1, 1)
CHECK_WORN ← function() runif(1) < 0.2
RETOOL ← function() rexp(1, 2)
RESET ← function() rexp(1, 3)
AWAY ← function() rexp(1, 1)

job ← trajectory() %>%
  ...
task ← trajectory() %>%
  ...

NEW_JOB ← function() rexp(1, 5)
NEW_TASK ← function() rexp(1, 1)
```

```r
env ← simmer("Job Shop") %>%
  add_resource("machine", 10) %>%
  add_resource("operative", 5) %>%
  add_generator("job", job, NEW_JOB) %>%
  add_generator("task", task, NEW_TASK) %>%
  run(until=1000)
env
```

```
## simmer environment: Job Shop | now: 1000 | next: 1000.11
## { Monitor: in memory }
## { Resource: machine | monitored: TRUE | server status: 9
## { Resource: operative | monitored: TRUE | server status:
## { Source: job | monitored: 1 | n_generated: 4954 }
## { Source: task | monitored: 1 | n_generated: 1041 }
```

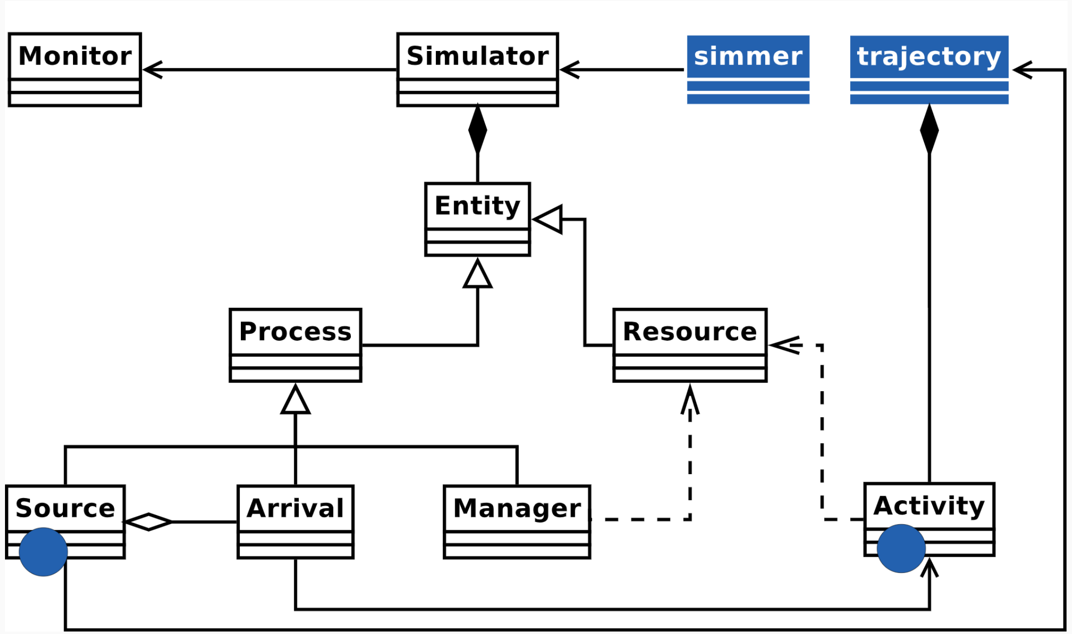
```r
aggregate(cbind(server, queue) ~ resource, get_mon_resources(env), mean)
```

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

Simulation core design

Terminology

Architecture



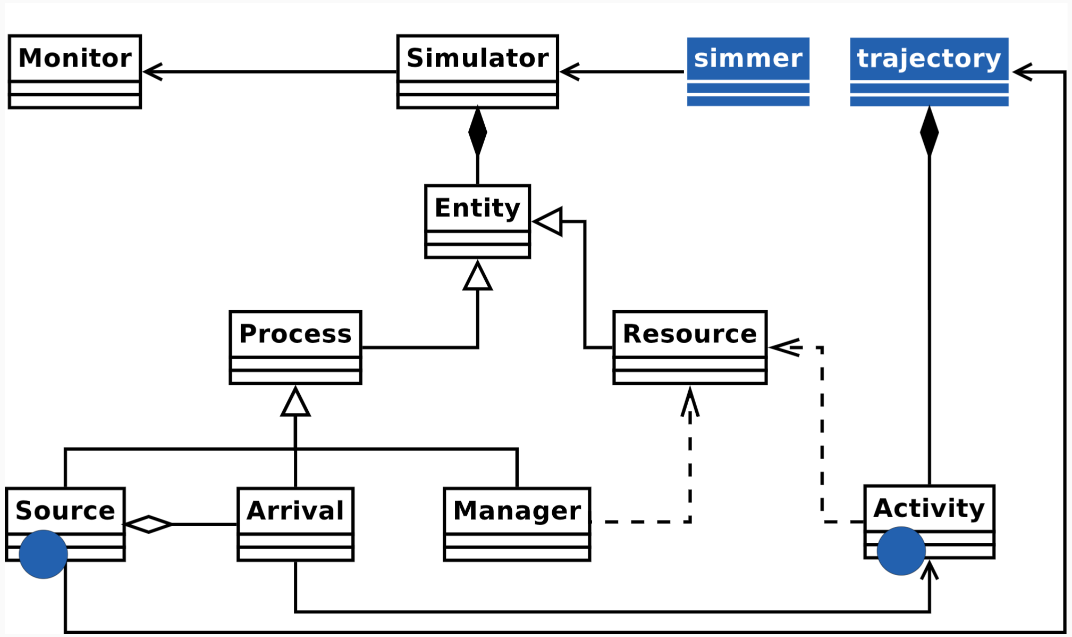
Overview of the C++ core (white) and R API (blue)

Simulation core design

Terminology

- **Resource**: server (configurable capacity) + priority queue (configurable size), supports preemption
- **Manager**: modifies resources at run time (schedule)

Architecture



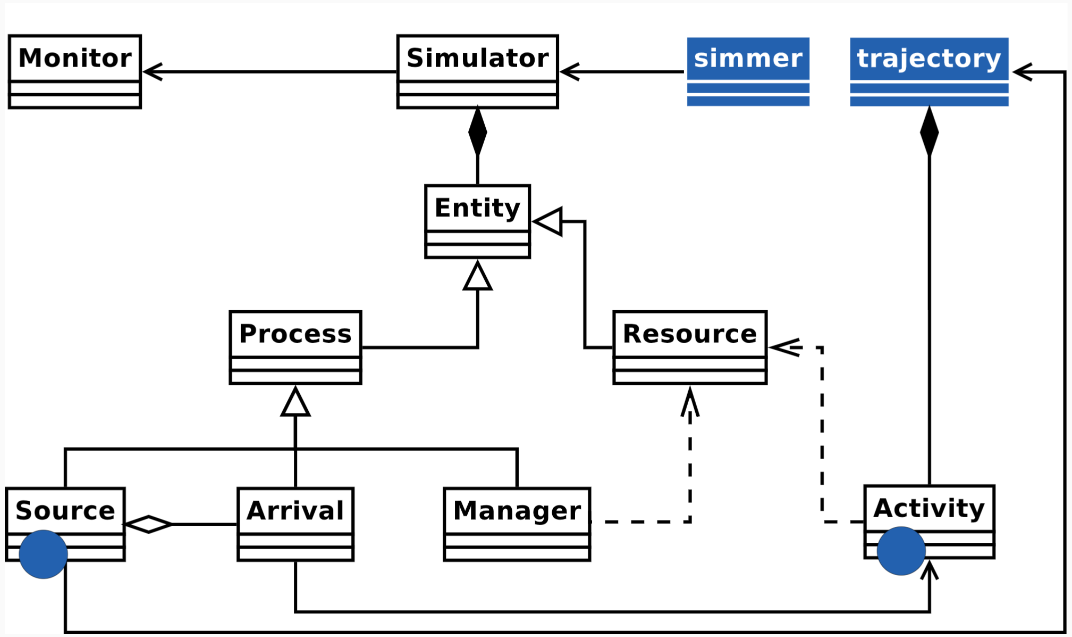
Overview of the C++ core (white) and R API (blue)

Simulation core design

Terminology

- **Resource**: server (configurable capacity) + priority queue (configurable size), supports preemption
- **Manager**: modifies resources at run time (schedule)
- **Source**: creates new arrivals following some distribution of inter-arrival times
- **Arrival**: interacting processes, with attributes and prioritization values

Architecture



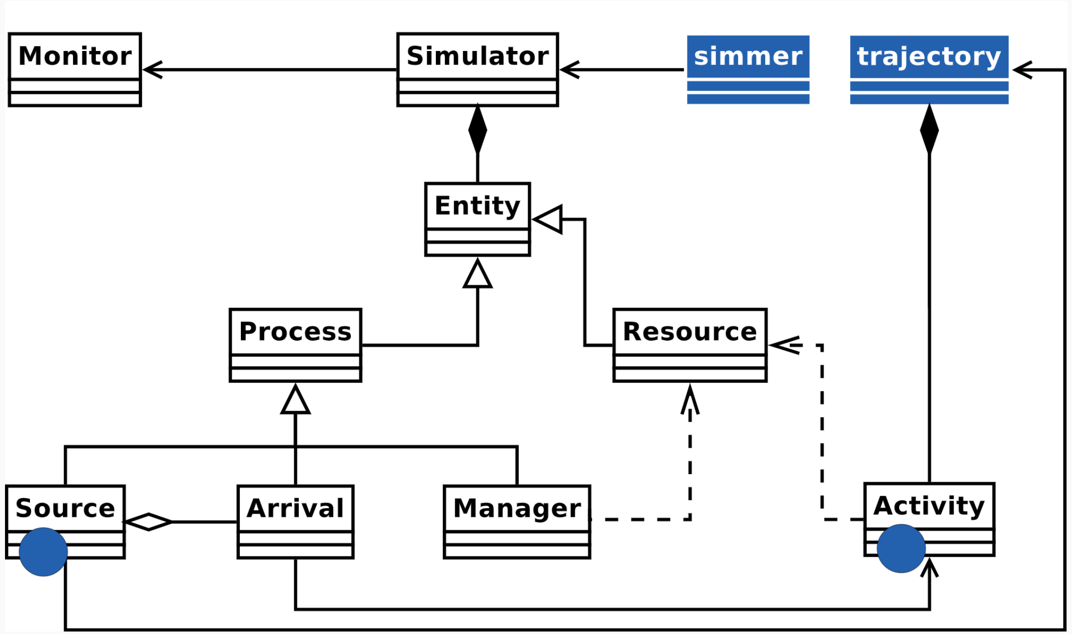
Overview of the C++ core (white) and R API (blue)

Simulation core design

Terminology

- **Resource**: server (configurable capacity) + priority queue (configurable size), supports preemption
- **Manager**: modifies resources at run time (schedule)
- **Source**: creates new arrivals following some distribution of inter-arrival times
- **Arrival**: interacting processes, with attributes and prioritization values
- **Trajectory**: interlinkage of activities, a common path for arrivals of the same type
- **Activity**: unit of action, main building block

Architecture



Overview of the C++ core (white) and R API (blue)

The simmer API

Trajectory

Similar to `dplyr` for data manipulation. In the words of H. Wickham,

> by constraining your options, it simplifies how you can think about [something]

- Trajectories are recipes, lists of activities defining the life time of arrivals
- Activities are common functional DES blocks

The simmer API

Trajectory

Similar to `dplyr` for data manipulation. In the words of H. Wickham,

> by constraining your options, it simplifies how you can think about [something]

- Trajectories are recipes, lists of activities defining the life time of arrivals
- Activities are common functional DES blocks

Fixed vs. dynamic parameters:

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

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

The simmer API

Trajectory

Activities

- Spend time in the system
  - `timeout`, `timeout_from_attribute`, `timeout_from_global`
- Modify arrival properties
  - `set_attribute`, `set_global`
  - `set_prioritization`
- Interaction with resources
  - `seize`, `release`
  - `set_capacity`, `set_queue_size`
  - `select`, `seize_selected`…
- Interaction with sources
  - `activate`, `deactivate`
  - `set_trajectory`, `set_source`
- Loops
  - `rollback`
- Branching
  - `branch`
  - `clone`, `synchronize`
- Batching
  - `batch`, `separate`
- Inter-arrival communication
  - `send`, `trap`, `untrap`, `wait`
- Reneging
  - `leave`
  - `renege_in`, `renege_if`, `renege_abort`
  - `handle_unfinished`
- Debugging
  - `log_`
  - `stop_if`

The simmer API

Trajectory

Activities

- Spend time in the system
  - `timeout`, `timeout_from_attribute`, `timeout_from_global`
- Modify arrival properties
  - `set_attribute`, `set_global`
  - `set_prioritization`
- Interaction with resources
  - `seize`, `release`
  - `set_capacity`, `set_queue_size`
  - `select`, `seize_selected`...
- Interaction with sources
  - `activate`, `deactivate`
  - `set_trajectory`, `set_source`
- Loops
  - `rollback`
- Branching
  - `branch`
  - `clone`, `synchronize`
- Batching
  - `batch`, `separate`
- Inter-arrival communication
  - `send`, `trap`, `untrap`, `wait`
- Reneging
  - `leave`
  - `renege_in`, `renege_if`, `renege_abort`
  - `handle_unfinished`
- Debugging
  - `log_`
  - `stop_if`

(Plus many getters to retrieve parameters at run time)

The simmer API

Trajectory

Activities

Simulation environment

- Create a simulator and attach a monitor
  - `simmer`
  - `monitor_mem` (default), `monitor_csv`... (extensible, see `simmer.mon` on GitHub)

The simmer API

Trajectory

Activities

Simulation environment

- Create a simulator and attach a monitor
  - `simmer`
  - `monitor_mem` (default), `monitor_csv`... (extensible, see `simmer.mon` on GitHub)

- Add sources of arrivals
  - `add_generator`: based on a distribution function
  - `add_dataframe`: based on a data frame (additional columns as attributes)

The simmer API

Trajectory

Activities

Simulation environment

- Create a simulator and attach a monitor
  - `simmer`
  - `monitor_mem` (default), `monitor_csv`... (extensible, see `simmer.mon` on GitHub)

- Add sources of arrivals
  - `add_generator`: based on a distribution function
  - `add_dataframe`: based on a data frame (additional columns as attributes)

- Add resources
  - `add_resource`: priority resource, with capacity and queue size; optional preemption

The simmer API

Trajectory

Activities

Simulation environment

- Create a simulator and attach a monitor
  - `simmer`
  - `monitor_mem` (default), `monitor_csv`... (extensible, see `simmer.mon` on GitHub)
- Add sources of arrivals
  - `add_generator`: based on a distribution function
  - `add_dataframe`: based on a data frame (additional columns as attributes)
- Add resources
  - `add_resource`: priority resource, with capacity and queue size; optional preemption
- Add global variables
  - `add_global`

The simmer API

Trajectory

Activities

Simulation environment

- Create a simulator and attach a monitor
  - `simmer`
  - `monitor_mem` (default), `monitor_csv`... (extensible, see `simmer.mon` on GitHub)
- Add sources of arrivals
  - `add_generator`: based on a distribution function
  - `add_dataframe`: based on a data frame (additional columns as attributes)
- Add resources
  - `add_resource`: priority resource, with capacity and queue size; optional preemption
- Add global variables
  - `add_global`
- Run the simulation
  - `run`, `stepn`

`simmer` automatically records every change in the state of the system. All these statistics can be retrieved after the simulation:

```
names( get_mon_arrivals(simmer(), per_resource=FALSE) )
```

```
## [1] "name"          "start_time"    "end_time"      "activity_time"
## [5] "finished"
```

```
names( get_mon_arrivals(simmer(), per_resource=TRUE) )
```

```
## [1] "name"          "start_time"    "end_time"      "activity_time"
## [5] "resource"
```

```
names( get_mon_attributes(simmer()) )
```

```
## [1] "time"  "name"  "key"   "value"
```

```
names( get_mon_resources(simmer()) )
```

```
## [1] "resource"   "time"       "server"     "queue"      "capacity"
## [6] "queue_size" "system"     "limit"
```

Queueing systems

Natural way to simulate CTMC and birth-death processes:

```
set.seed(1234)
lambda <- 2
mu <- 4
rho <- lambda/mu
```

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

```
mm1.env <- simmer() %>%
  add_resource("mm1.resource", 1, Inf) %>%
  add_generator("arrival", mm1.traj,
                function() rexp(1, lambda)) %>%
  run(2000)
```

Modelling

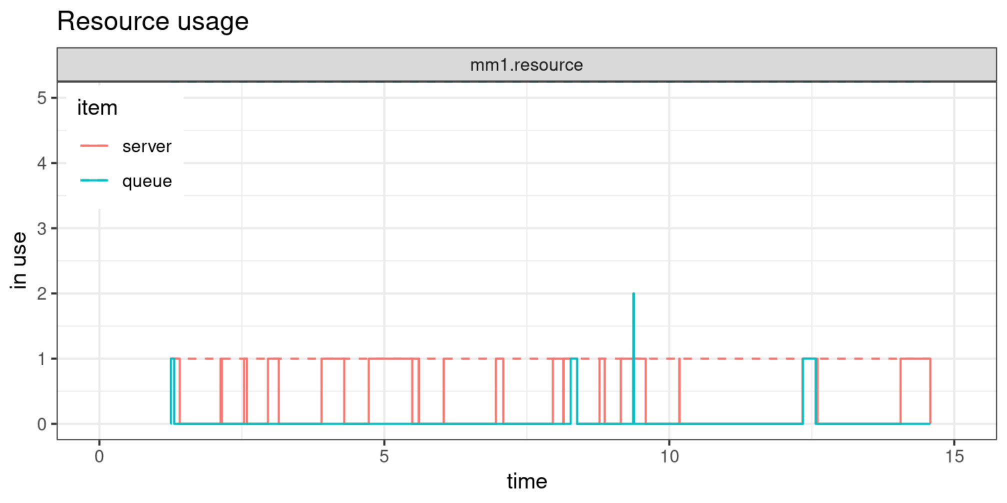
Queueing systems

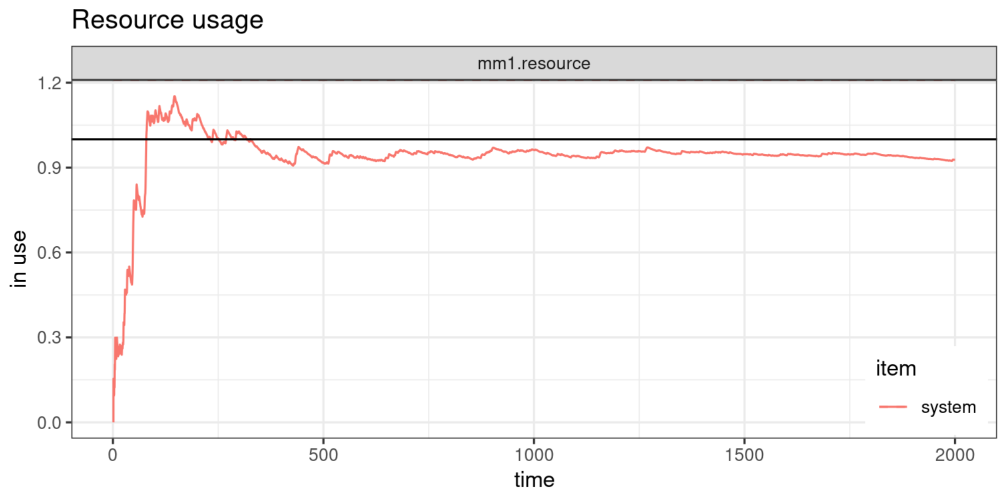
Natural way to simulate CTMC and birth-death processes:

```r
set.seed(1234)
lambda <- 2
mu <- 4
rho <- lambda/mu
```

```r
mm1.traj <- trajectory() %>%
  seize("mm1.resource", 1) %>%
  timeout(function() rexp(1, mu)) %>%
  release("mm1.resource", 1)
```

```r
mm1.env <- simmer() %>%
  add_resource("mm1.resource", 1, Inf) %>%
  add_generator("arrival", mm1.traj,
                function() rexp(1, lambda)) %>%
  run(2000)
```





Modelling

Replication

Easy replication with standard R functions:

```
mm1.envs ← lapply(1:20, function(i) {
  simmer() %>%
    add_resource("mm1.resource", 1, Inf) %>%
    add_generator("arrival", mm1.traj,
                  function() rexp(100, lambda)) %>%
    run(1000/lambda)
})
```

Modelling

Replication

Easy replication with standard R functions:

```
mm1.envs <- lapply(1:20, function(i) {
  simmer() %>%
    add_resource("mm1.resource", 1, Inf) %>%
    add_generator("arrival", mm1.traj,
                  function() rexp(100, lambda)) %>%
    run(1000/lambda)
})
```

Parallelization

Even easier parallelization of replicas:

```
mm1.envs <- parallel::mclapply(1:20, function(i) {
  simmer() %>%
    add_resource("mm1.resource", 1, Inf) %>%
    add_generator("arrival", mm1.traj,
                  function() rexp(100, lambda)) %>%
    run(1000/lambda) %>%
    wrap()
}, mc.cores=4)
```

Modelling

Replication

Easy replication with standard R functions:

```
mm1.envs <- lapply(1:20, function(i) {
  simmer() %>%
    add_resource("mm1.resource", 1, Inf) %>%
    add_generator("arrival", mm1.traj,
                  function() rexp(100, lambda)) %>%
    run(1000/lambda)
})
```

Parallelization

Even easier parallelization of replicas:

```
mm1.envs <- parallel::mclapply(1:20, function(i) {
  simmer() %>%
    add_resource("mm1.resource", 1, Inf) %>%
    add_generator("arrival", mm1.traj,
                  function() rexp(100, lambda)) %>%
    run(1000/lambda) %>%
    wrap()
}, mc.cores=4)
```

```
head(get_mon_arrivals(mm1.envs), 3)
```

```
##       name start_time  end_time activity_time finished replication
## 1 arrival0  0.1455818 0.4810091     0.3354273     TRUE           1
## 2 arrival1  0.2380988 0.6834863     0.2024772     TRUE           1
## 3 arrival2  0.4526185 0.8724942     0.1890079     TRUE           1
```

Best practices

There are usually multiple valid ways of mapping the identified resources and processes into the `simmer` API

Modelling

Best practices

There are usually multiple valid ways of mapping the identified resources and processes into the `simmer` API

Design pattern 1

```r
beep <- trajectory() %>%
  log_("beeeep!")
```

```r
env <- simmer() %>%
  add_generator("beep", beep, function() 1) %>%
  run(2.5)
```

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

Modelling

Best practices

There are usually multiple valid ways of mapping the identified resources and processes into the `simmer` API

Design pattern 1

```r
beep <- trajectory() %>%
  log_("beeeep!")
```

```r
env <- simmer() %>%
  add_generator("beep", beep, function() 1) %>%
  run(2.5)
```

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

Design pattern 2

```r
alarm <- trajectory() %>%
  timeout(1) %>%
  log_("beeeep!") %>%
  rollback(2)
```

```r
env <- simmer() %>%
  add_generator("alarm", alarm, at(0)) %>%
  run(2.5)
```

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

Performance

Comparison with similar frameworks (out-of-date!):

- SimPy 3.0.9, Python 2.7
- SimJulia 0.3.14, Julia 0.5.1

Heavy M/M/1, $\rho \approx 0.9$:

```r
test_mm1_simmer ← function(n, m, mon=FALSE) {
  mm1 ← trajectory() %>%
    seize("server", 1) %>%
    timeout(function() rexp(1, 1.1)) %>%
    release("server", 1)

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

Performance

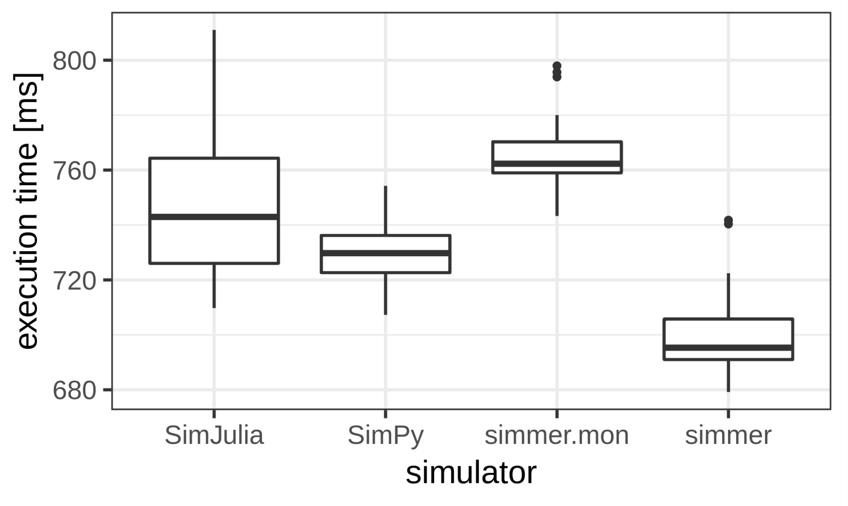
Comparison with similar frameworks (out-of-date!):

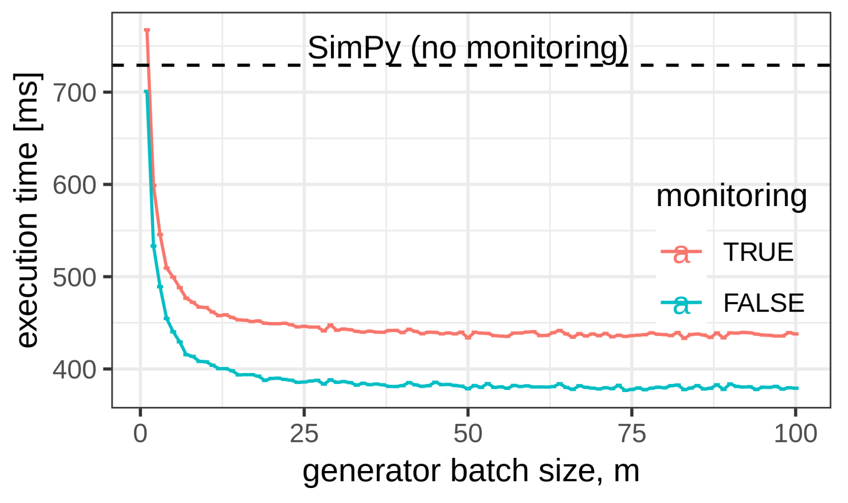
- SimPy 3.0.9, Python 2.7
- SimJulia 0.3.14, Julia 0.5.1

Heavy M/M/1, $\rho \approx 0.9$:

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

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





The cost of calling R from C++ revisited

Very simple deterministic test to study the impact:

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

test_simmer(5, 1)[,1:5]
```

```
##    name start_time end_time activity_time finished
## 1 test0          1        2             1     TRUE
## 2 test1          2        3             1     TRUE
## 3 test2          3        4             1     TRUE
## 4 test3          4        5             1     TRUE
## 5 test4          5        6             1     TRUE
```

The cost of calling R from C++ revisited

Very simple deterministic test to study the impact:

```r
test_simmer <- function(n, delay) {
  test <- trajectory() %>%
    timeout(delay)
  simmer() %>%
    add_generator("test", test, at(1:n)) %>%
    run() %>%
    get_mon_arrivals()
}

test_simmer(5, 1)[,1:5]
```

```
##    name start_time end_time activity_time finished
## 1 test0          1        2             1     TRUE
## 2 test1          2        3             1     TRUE
## 3 test2          3        4             1     TRUE
## 4 test3          4        5             1     TRUE
## 5 test4          5        6             1     TRUE
```

Original benchmark in the JSS paper:

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

The cost of calling R from C++ revisited

Very simple deterministic test to study the impact:

```r
test_simmer <- function(n, delay) {
  test <- trajectory() %>%
    timeout(delay)
  simmer() %>%
    add_generator("test", test, at(1:n)) %>%
    run() %>%
    get_mon_arrivals()
}

test_simmer(5, 1)[,1:5]
```

```
##    name start_time end_time activity_time finished
## 1 test0          1        2             1     TRUE
## 2 test1          2        3             1     TRUE
## 3 test2          3        4             1     TRUE
## 4 test3          4        5             1     TRUE
## 5 test4          5        6             1     TRUE
```

Original benchmark in the JSS paper:

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

Update with `-DRCPP_USE_UNWIND_PROTECT`:

Expr	Min	Mean	Median	Max
test_simmer(n, 1)	467.8971	481.213	476.1667	521.4916
test_simmer(n, function() 1)	498.2631	583.777	561.6798	816.1343
test_R_for(n)	1158.9348	1201.460	1196.7223	1244.4041

Summary

- Generic yet powerful process-oriented Discrete-Event Simulation framework for R [1, 2]
- Combines a robust and **fast** C++ simulation core with a **rich and flexible** R API

[1] **Ucar I**, Smeets B, Azcorra A (2019). "simmer: Discrete-Event Simulation for R." *Journal of Statistical Software*, 90(2), 1-30. doi: 10.18637/jss.v090.i02.
[2] **Ucar I**, Hernández JA, Serrano P, Azcorra A (2018). "Design and Analysis of 5G Scenarios with simmer: An R Package for Fast DES Prototyping." *IEEE Communications Magazine*, 56(11), 145-151. doi: 10.1109/MCOM.2018.1700960.

Summary

- Generic yet powerful process-oriented Discrete-Event Simulation framework for R [1, 2]
- Combines a robust and **fast** C++ simulation core with a **rich and flexible** R API
- Broad set of **activities**, the basic building block; extensible via custom routines
- Activities are chained into a **trajectory**, a common path for processes of the same type

[1] **Ucar I**, Smeets B, Azcorra A (2019). "simmer: Discrete-Event Simulation for R." *Journal of Statistical Software*, 90(2), 1-30. doi: 10.18637/jss.v090.i02.
[2] **Ucar I**, Hernández JA, Serrano P, Azcorra A (2018). "Design and Analysis of 5G Scenarios with simmer: An R Package for Fast DES Prototyping." *IEEE Communications Magazine*, 56(11), 145-151. doi: 10.1109/MCOM.2018.1700960.

Summary

- Generic yet powerful process-oriented Discrete-Event Simulation framework for R [1, 2]
- Combines a robust and **fast** C++ simulation core with a **rich and flexible** R API
- Broad set of **activities**, the basic building block; extensible via custom routines
- Activities are chained into a **trajectory**, a common path for processes of the same type
- **Automatic monitoring**: focus on modelling

[1] **Ucar I**, Smeets B, Azcorra A (2019). "simmer: Discrete-Event Simulation for R." *Journal of Statistical Software*, 90(2), 1-30. doi: 10.18637/jss.v090.i02.
[2] **Ucar I**, Hernández JA, Serrano P, Azcorra A (2018). "Design and Analysis of 5G Scenarios with simmer: An R Package for Fast DES Prototyping." *IEEE Communications Magazine*, 56(11), 145-151. doi: 10.1109/MCOM.2018.1700960.

Summary

- Generic yet powerful process-oriented Discrete-Event Simulation framework for R [1, 2]
- Combines a robust and **fast** C++ simulation core with a **rich and flexible** R API
- Broad set of **activities**, the basic building block; extensible via custom routines
- Activities are chained into a **trajectory**, a common path for processes of the same type
- **Automatic monitoring**: focus on modelling
- **Integration**: easy replication, parallelization, analysis...

[1] **Ucar I**, Smeets B, Azcorra A (2019). "simmer: Discrete-Event Simulation for R." *Journal of Statistical Software*, 90(2), 1-30. doi: 10.18637/jss.v090.i02.
[2] **Ucar I**, Hernández JA, Serrano P, Azcorra A (2018). "Design and Analysis of 5G Scenarios with simmer: An R Package for Fast DES Prototyping." *IEEE Communications Magazine*, 56(11), 145-151. doi: 10.1109/MCOM.2018.1700960.

Thanks, and happy **simmer**ing!

https://r-simmer.org

simmer
DES for R