Package 'simmer'

July 30, 2019

Type Package

Title Discrete-Event Simulation for R

Version 4.3.0

Description A process-oriented and trajectory-based Discrete-Event Simulation (DES) package for R. It is designed as a generic yet powerful 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. Documentation about 'simmer' is provided by several vignettes included in this package, via the paper by Ucar, Smeets & Azcorra (2019, <doi:10.18637/jss.v090.i02>), and the paper by Ucar, Hernández, Serrano & Azcorra (2018, <doi:10.1109/MCOM.2018.1700960>); see 'citation(``simmer'')' for details.

License GPL (>= 2)

Encoding UTF-8

URL `http://r-simmer.org`, `https://github.com/r-simmer/simmer`

BugReports `https://github.com/r-simmer/simmer/issues`

Depends R (>= 3.1.2)

Imports Rcpp, magrittr, codetools, utils

Suggests simmer.plot, parallel, testthat, knitr, rmarkdown, rticles

LinkingTo Rcpp (>= 0.12.9), BH (>= 1.62.0-1)

RoxygenNote 6.1.1

VignetteBuilder knitr

NeedsCompilation yes

Author Iñaki Ucar [aut, cph, cre] (<https://orcid.org/0000-0001-6403-5550>), Bart Smeets [aut, cph]

Maintainer Iñaki Ucar <iucar@fedoraproject.org>

Repository CRAN

Date/Publication 2019-07-30 21:30:02 UTC

R topics documented:

simmer-package **simmer***: Discrete-Event Simulation for* R

Description

A process-oriented and trajectory-based Discrete-Event Simulation (DES) package for R. Designed to be a generic framework like **SimPy** or **SimJulia**, it leverages the power of **Rcpp** to boost the performance and turning DES in R feasible. As a noteworthy characteristic, **simmer** exploits the concept of trajectory: a common path in the simulation model for entities of the same type. It is pretty flexible and simple to use, and leverages the chaining/piping workflow introduced by the **magrittr** package.

Author(s)

Iñaki Ucar and Bart Smeets

References

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[1].

Ucar I., Hernández J.A., 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[2].

See Also

simmer's homepage `http://r-simmer.org` and GitHub repository `https://github.com/r-simmer/simmer`.

Examples

```
## Not run:
# introduction to simmer
vignette("simmer-01-introduction")

# JSS paper available as vignette
vignette("simmer-02-jss")

# more vignettes
vignette(package = "simmer")

## End(Not run)
```

[1] http://doi.org/10.18637/jss.v090.i02
[2] http://doi.org/10.1109/MCOM.2018.1700960

`activate` *Activate/Deactivate Sources*

Description

Activities for activating or deactivating the generation of arrivals by name. Sources must be defined in the simulation environment (see `add_generator`, `add_dataframe`).

Usage

```
activate(.trj, source)

deactivate(.trj, source)
```

Arguments

`.trj`	the trajectory object.
`source`	the name of the source or a function returning a name.

Value

Returns the trajectory object.

See Also

`set_trajectory`, `set_source`

Examples

```
traj <- trajectory() %>%
  deactivate("dummy") %>%
  timeout(1) %>%
  activate("dummy")

simmer() %>%
  add_generator("dummy", traj, function() 1) %>%
  run(10) %>%
  get_mon_arrivals()
```

`add_dataframe` *Add a Data Frame*

Description

Attach a new source of arrivals to a trajectory from a data frame.

Usage

```
add_dataframe(.env, name_prefix, trajectory, data, mon = 1, batch = 50,
  col_time = "time", time = c("interarrival", "absolute"),
  col_attributes = NULL, col_priority = "priority",
  col_preemptible = col_priority, col_restart = "restart")
```

Arguments

`.env`	the simulation environment.
`name_prefix`	the name prefix of the generated arrivals.
`trajectory`	the trajectory that the generated arrivals will follow (see `trajectory`).
`data`	a data frame with, at least, a column of (inter)arrival times (see details).
`mon`	whether the simulator must monitor the generated arrivals or not (0 = no monitoring, 1 = simple arrival monitoring, 2 = level 1 + arrival attribute monitoring)
`batch`	number of arrivals generated at a time. Arrivals are read from the data frame and attached to the trajectory in batches depending on this value. In general, it should not be changed.
`col_time`	name of the time column in the data frame.
`time`	type of time column: *interarrival*, if the time column contains interarrival times, or *absolute*, if the time column contains absolute arrival times.
`col_attributes`	vector of names of the attributes columns (see details).
`col_priority`	name of the priority column.
`col_preemptible`	name of the preemptible column.
`col_restart`	name of the restart column.

Details

The data frame provided must have, at least, a column of (inter)arrival times. This method will look for it under the name `"time"` by default, although this can be changed with the `col_time` parameter.

If there is any column named `col_priority="priority"`, `col_preemptible=priority` or `col_restart="restart"`, they will be used to set the prioritization values for each arrival (see `add_generator`).

If there are additional columns (with `col_attributes=NULL`, by default), they will be assigned to arrival attributes named after each column name. All these columns must be numeric (or logical). Otherwise, if a vector of column names is specified, only these will be assigned as attributes and the rest of the columns will be ignored.

A value of `batch=Inf` means that the whole data frame will be attached at the beginning of the simulation. This is not desirable in general, because the performance of the event queue is degraded when it is populated with too many events. On the other hand, a low value results in an increased overhead due to many function calls. The default value has been tested to provide a good trade-off.

Value

Returns the simulation environment.

See Also

Other sources: `add_generator`.

`add_generator` *Add a Generator*

Description

Attach a new source of arrivals to a trajectory from a generator function.

Usage

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

Arguments

`.env`	the simulation environment.
`name_prefix`	the name prefix of the generated arrivals.
`trajectory`	the trajectory that the generated arrivals will follow (see `trajectory`).
`distribution`	a function modelling the interarrival times (returning a negative value stops the generator).
`mon`	whether the simulator must monitor the generated arrivals or not (0 = no monitoring, 1 = simple arrival monitoring, 2 = level 1 + arrival attribute monitoring)
`priority`	the priority of each arrival (a higher integer equals higher priority; defaults to the minimum priority, which is 0).
`preemptible`	if a seize occurs in a preemptive resource, this parameter establishes the minimum incoming priority that can preempt these arrivals (an 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 activity must be restarted after being preempted.

Value

Returns the simulation environment.

See Also

Convenience functions: `at`, `from`, `to`, `from_to`.

Other sources: `add_dataframe`.

`add_global` *Add a Global Attribute*

Description

Attach a global variable to the simulation.

Usage

```
add_global(.env, key, value)
```

Arguments

`.env`	the simulation environment.
`key`	the attribute name.
`value`	the value to set, either a numeric or a `schedule`, so that the global may change during the simulation.

Value

Returns the simulation environment.

See Also

Convenience functions: `schedule`.

add_resource *Add a Resource*

Description

Define a new resource in a simulation environment.

Usage

```
add_resource(.env, name, capacity = 1, queue_size = Inf, mon = TRUE,
  preemptive = FALSE, preempt_order = c("fifo", "lifo"),
  queue_size_strict = FALSE, queue_priority = c(0, Inf))
```

Arguments

.env	the simulation environment.
name	the name of the resource.
capacity	the capacity of the server, either a numeric or a schedule, so that the value may change during the simulation.
queue_size	the size of the queue, either a numeric or a schedule, so that the value may change during the simulation.
mon	whether the simulator must monitor this resource or not.
preemptive	whether arrivals in the server can be preempted or not based on seize priorities.
preempt_order	if the resource is preemptive and preemption occurs with more than one arrival in the server, this parameter defines which arrival should be preempted first. It must be fifo (First In First Out: older preemptible tasks are preempted first) or lifo (Last In First Out: newer preemptible tasks are preempted first).
queue_size_strict	if the resource is preemptive and preemption occurs, this parameter controls whether the queue_size is a hard limit. By default, preempted arrivals go to a dedicated queue, so that queue_size may be exceeded. If this option is TRUE, preempted arrivals go to the standard queue, and the maximum queue_size is guaranteed (rejection may occur). Whenever an arrival is rejected (due to a server drop or a queue drop), it will set the finished flag to FALSE in the output of get_mon_arrivals. Unfinished arrivals can be handled with a drop-out trajectory that can be set using the handle_unfinished activity.
queue_priority	the priority range required to be able to access the queue if there is no room in the server (if a single value is provided, it is treated as the minimum priority). By default, all arrivals can be enqueued.

Value

Returns the simulation environment.

See Also

Convenience functions: `schedule`.

`batch` *Batch/Separate Arrivals*

Description

Activities for collecting a number of arrivals before they can continue processing and splitting a previously established batch.

Usage

```
batch(.trj, n, timeout = 0, permanent = FALSE, name = "",
  rule = NULL)

separate(.trj)
```

Arguments

`.trj`	the trajectory object.
`n`	batch size, accepts a numeric.
`timeout`	set an optional timer which triggers batches every `timeout` time units even if the batch size has not been fulfilled, accepts a numeric or a callable object (a function) which must return a numeric (0 = disabled).
`permanent`	if `TRUE`, batches cannot be split.
`name`	optional string. Unnamed batches from different `batch` activities are independent. However, if you want to feed arrivals from different trajectories into a same batch, you need to specify a common name across all your `batch` activities.
`rule`	an optional callable object (a function) which will be applied to every arrival to determine whether it should be included into the batch, thus

Value

Returns the trajectory object.

Examples

```
## unnamed batch with a timeout
traj <- trajectory() %>%
  log_("arrived") %>%
  batch(2, timeout=5) %>%
  log_("in a batch") %>%
  timeout(5) %>%
  separate() %>%
  log_("leaving")
```

```
simmer() %>%
  add_generator("dummy", traj, at(0:2)) %>%
  run() %>% invisible

## batching based on some dynamic rule
traj <- trajectory() %>%
  log_("arrived") %>%
  # always FALSE -> no batches
  batch(2, rule=function() FALSE) %>%
  log_("not in a batch") %>%
  timeout(5) %>%
  separate() %>%
  log_("leaving")

simmer() %>%
  add_generator("dummy", traj, at(0:2)) %>%
  run() %>% invisible

## named batch, shared across trajectories
traj0 <- trajectory() %>%
  log_("arrived traj0") %>%
  batch(2, name = "mybatch")

traj1 <- trajectory() %>%
  log_("arrived traj1") %>%
  timeout(1) %>%
  batch(2, name = "mybatch") %>%
  log_("in a batch") %>%
  timeout(2) %>%
  separate() %>%
  log_("leaving traj1")

simmer() %>%
  add_generator("dummy0", traj0, at(0)) %>%
  add_generator("dummy1", traj1, at(0)) %>%
  run() %>% invisible
```

`branch` *Fork the Trajectory Path*

Description

Activity for defining a fork with `N` alternative sub-trajectories.

Usage

```
branch(.trj, option, continue, ...)
```

Arguments

.trj	the trajectory object.
option	a callable object (a function) which must return an integer between 0 and N. A return value equal to 0 skips the branch and continues to the next activity. A returning value between 1 to N makes the arrival to follow the corresponding sub-trajectory.
continue	a vector of N booleans that indicate whether the arrival must continue to the main trajectory after each sub-trajectory or not (if only one value is provided, it will be recycled to match the number of sub-trajectories).
...	N trajectory objects (or a list of N trajectory objects) describing each sub-trajectory.

Value

Returns the trajectory object.

Examples

```
env <- simmer()

traj <- trajectory() %>%
  set_global("path", 1, mod="+", init=-1) %>%
  log_(function() paste("Path", get_global(env, "path"), "selected")) %>%
  branch(
    function() get_global(env, "path"), continue=c(TRUE, FALSE),
    trajectory() %>%
      log_("following path 1"),
    trajectory() %>%
      log_("following path 2")) %>%
  log_("continuing after the branch (path 0)")

env %>%
  add_generator("dummy", traj, at(0:2)) %>%
  run() %>% invisible
```

---

clone *Clone/Synchronize Arrivals*

Description

Activities for defining a parallel fork and removing the copies. clone replicates an arrival n times (the original one + n-1 copies). synchronize removes all but one clone for each set of clones.

Usage

```
clone(.trj, n, ...)

synchronize(.trj, wait = TRUE, mon_all = FALSE)
```

Arguments

.trj	the trajectory object.
n	number of clones, accepts either a numeric or a callable object (a function) which must return a numeric.
...	a number of optional parallel sub-trajectories (or a list of sub-trajectories). Each clone will follow a different sub-trajectory if available.
wait	if FALSE, all clones but the first to arrive are removed. if TRUE (default), all clones but the last to arrive are removed.
mon_all	if TRUE, get_mon_arrivals will show one line per clone.

Value

Returns the trajectory object.

Examples

```
## clone and wait for the others
traj <- trajectory() %>%
  clone(
    n = 3,
    trajectory() %>%
      log_("clone 0 (original)") %>%
      timeout(1),
    trajectory() %>%
      log_("clone 1") %>%
      timeout(2),
    trajectory() %>%
      log_("clone 2") %>%
      timeout(3)) %>%
  log_("sync reached") %>%
  synchronize(wait = TRUE) %>%
  log_("leaving")

simmer() %>%
  add_generator("arrival", traj, at(0)) %>%
  run() %>% invisible

## more clones that trajectories available
traj <- trajectory() %>%
  clone(
    n = 5,
    trajectory() %>%
      log_("clone 0 (original)") %>%
      timeout(1)) %>%
  log_("sync reached") %>%
  synchronize(wait = TRUE) %>%
  log_("leaving")

simmer() %>%
  add_generator("arrival", traj, at(0)) %>%
```

```
  run() %>% invisible

## clone and continue without waiting
traj <- trajectory() %>%
  clone(
    n = 3,
    trajectory() %>%
      log_("clone 0 (original)") %>%
      timeout(1),
    trajectory() %>%
      log_("clone 1") %>%
      timeout(2),
    trajectory() %>%
      log_("clone 2") %>%
      timeout(3)) %>%
  log_("sync reached") %>%
  synchronize(wait = FALSE) %>%
  log_("leaving")

simmer() %>%
  add_generator("arrival", traj, at(0)) %>%
  run() %>% invisible
```

`Extract.trajectory` *Extract or Replace Parts of a Trajectory*

Description

Operators acting on trajectories to extract or replace parts.

Usage

```
## S3 method for class 'trajectory'
x[i]

## S3 method for class 'trajectory'
x[[i]]

## S3 replacement method for class 'trajectory'
x[i] <- value

## S3 replacement method for class 'trajectory'
x[[i]] <- value
```

Arguments

`x`	the trajectory object.

i	indices specifying elements to extract. Indices are `numeric` or `character` or `logical` vectors or empty (missing) or `NULL`. Numeric values are coerced to integer as by `as.integer` (and hence truncated towards zero). Negative integers indicate elements/slices to leave out the selection. Character vectors will be matched to the names of the activities in the trajectory as by `%in%`. Logical vectors indicate elements/slices to select. Such vectors are recycled if necessary to match the corresponding extent. An empty index will return the whole trajectory. An index value of `NULL` is treated as if it were `integer(0)`.
value	another trajectory object.

Value

Returns a new trajectory object.

See Also

`length.trajectory`, `get_n_activities`, `join`.

Examples

```
x <- join(lapply(1:12, function(i)
  trajectory() %>% timeout(i)
))
x

x[10]                 # the tenth element of x
x[-1]                 # delete the 1st element of x
x[c(TRUE, FALSE)]     # logical indexing
x[c(1, 5, 2, 12, 4)]  # numeric indexing
x[c(FALSE, TRUE)] <- x[c(TRUE, FALSE)] # replacing
x
```

`generators` *Convenience Functions for Generators*

Description

These convenience functions facilitate the definition of generators of arrivals for some common cases.

Usage

```
at(...)

from(start_time, dist, arrive = TRUE)

to(stop_time, dist)

from_to(start_time, stop_time, dist, arrive = TRUE, every = NULL)

when_activated(n = 1)
```

Arguments

...	a vector or multiple parameters of times at which to initiate an arrival.
start_time	the time at which to launch the initial arrival.
dist	a function modelling the interarrival times. It is supposed to be an infinite source of values >= 0 (e.g., rexp and the like). If the function provided returns any negative value, the behaviour is undefined.
arrive	if set to TRUE (default) the first arrival will be generated at start_time and will follow dist from then on. If set to FALSE, will initiate dist at start_time (and the first arrival will most likely start at a time later than start_time).
stop_time	the time at which to stop the generator.
every	repeat with this time cycle.
n	number of arrivals to generate when activated.

Details

at generates arrivals at specific absolute times.

from generates inter-arrivals following a given distribution with a specified start time. union of the last two.

to generates inter-arrivals following a given distribution with a specified stop time.

from_to is the union of from and to.

when_activated sets up an initially inactive generator which generates n arrivals each time it is activated from any trajectory using the activity activate.

Value

Returns a generator function (a closure).

See Also

add_generator.

Examples

```
## common to all examples below
# some trajectory
t0 <- trajectory() %>%
  timeout(0)
# some distribution
distr <- function() runif(1, 1, 2)

# arrivals at 0, 1, 10, 30, 40 and 43
simmer() %>%
  add_generator("dummy", t0, at(0, c(1,10,30), 40, 43)) %>%
  run(100) %>%
  get_mon_arrivals()

# apply distribution starting at 5 (and no end)
simmer() %>%
  add_generator("dummy", t0, from(5, distr)) %>%
  run(10) %>%
  get_mon_arrivals()

# apply distribution until 5 (starting at 0)
simmer() %>%
  add_generator("dummy", t0, to(5, distr)) %>%
  run(10) %>%
  get_mon_arrivals()

# apply distribution from 8 to 16 h every 24 h:
simmer() %>%
  add_generator("dummy", t0, from_to(8, 16, distr, every=24)) %>%
  run(48) %>%
  get_mon_arrivals()

# triggering arrivals on demand from a trajectory
t1 <- trajectory() %>%
  activate("dummy")

simmer() %>%
  add_generator("dummy", t0, when_activated()) %>%
  add_generator("trigger", t1, at(2)) %>%
  run() %>%
  get_mon_arrivals()
```

`get_capacity` *Get Resource Parameters*

Description

Getters for resources: server capacity/count and queue size/count, seized amount and selected resources.

Usage

```
get_capacity(.env, resources)

get_capacity_selected(.env, id = 0)

get_queue_size(.env, resources)

get_queue_size_selected(.env, id = 0)

get_server_count(.env, resources)

get_server_count_selected(.env, id = 0)

get_queue_count(.env, resources)

get_queue_count_selected(.env, id = 0)

get_seized(.env, resources)

get_seized_selected(.env, id = 0)

get_selected(.env, id = 0)
```

Arguments

`.env`	the simulation environment.
`resources`	one or more resource names.
`id`	selection identifier for nested usage.

Value

Return a vector (character for `get_selected`, numeric for the rest of them).

See Also

`get_resources`, `set_capacity`, `set_queue_size`.

`get_mon` *Monitoring Statistics*

Description

Getters for obtaining monitored data (if any) about arrivals, attributes and resources.

Usage

```
get_mon_arrivals(.envs, per_resource = FALSE, ongoing = FALSE)

get_mon_attributes(.envs)

get_mon_resources(.envs)
```

Arguments

.envs	the simulation environment (or a list of environments).
per_resource	if TRUE, statistics will be reported on a per-resource basis.
ongoing	if TRUE, ongoing arrivals will be reported. The columns end_time and finished of these arrivals are reported as NAs.

Value

Returns a data frame.

---

get_n_generated *Get Process Parameters*

Description

Getters for processes (sources and arrivals) number of arrivals generated by a source, the name of the active arrival, an attribute from the active arrival or a global one, and prioritization values.

Usage

```
get_n_generated(.env, sources)

get_trajectory(.env, sources)

get_name(.env)

get_attribute(.env, keys)

get_global(.env, keys)

get_prioritization(.env)
```

Arguments

.env	the simulation environment.
sources	one or more resource names.
keys	the attribute name(s).

Details

get_n_generated returns the number of arrivals generated by the given sources. get_trajectory returns the trajectory to which they are attached (as a list).

get_name returns the number of the running arrival. get_attribute returns a running arrival's attributes. If a provided key was not previously set, it returns a missing value. get_global returns a global attribute. get_prioritization returns a running arrival's prioritization values. get_name, get_attribute and get_prioritization are meant to be used inside a trajectory; otherwise, there will be no arrival running and these functions will throw an error.

See Also

get_sources, set_trajectory, set_attribute, set_global, set_prioritization.

---

get_sources *Get Sources and Resources Defined*

Description

Get a list of names of sources or resources defined in a simulation environment.

Usage

```
get_sources(.env)

get_resources(.env)
```

Arguments

.env	the simulation environment.

Value

A character vector.

---

handle_unfinished *Handle Unfinished Arrivals*

Description

Activity for setting a drop-out trajectory for unfinished arrivals, i.e., those dropped from a resource (due to preemption, resource shrinkage or a rejected seize) or those that leave a trajectory.

Usage

```
handle_unfinished(.trj, handler)
```

Arguments

`.trj`	the trajectory object.
`handler`	trajectory object to handle unfinished arrivals. A `NULL` value will unset the drop-out trajectory.

Value

Returns the trajectory object.

See Also

`leave`, `set_capacity`

Examples

```
traj <- trajectory() %>%
  log_("arrived") %>%
  handle_unfinished(
    trajectory() %>%
      log_("preempted!")) %>%
  seize("res") %>%
  log_("resource seized") %>%
  timeout(10) %>%
  release("res") %>%
  log_("leaving")

simmer() %>%
  add_resource("res", 1, 0, preemptive=TRUE, queue_size_strict=TRUE) %>%
  add_generator("dummy", traj, at(0)) %>%
  add_generator("priority_dummy", traj, at(5), priority=1) %>%
  run() %>% invisible
```

---

join *Join Trajectories*

Description

Concatenate any number of trajectories in the specified order.

Usage

```
join(...)
```

Arguments

`...`	trajectory objects.

Value

Returns a new trajectory object.

See Also

`Extract.trajectory`, `length.trajectory`, `get_n_activities`.

Examples

```
t1 <- trajectory() %>% seize("dummy", 1)
t2 <- trajectory() %>% timeout(1)
t3 <- trajectory() %>% release("dummy", 1)

## join can be used alone
join(t1, t2, t3)

## or can be chained in a trajectory definition
trajectory() %>%
  join(t1) %>%
  timeout(1) %>%
  join(t3)
```

`leave` *Leave the Trajectory*

Description

Activity for leaving the trajectory with some probability.

Usage

```
leave(.trj, prob)
```

Arguments

`.trj`	the trajectory object.
`prob`	a probability or a function returning a probability.

Details

Arrivals that leave the trajectory will set the `finished` flag to `FALSE` in the output of `get_mon_arrivals`. Unfinished arrivals can be handled with a drop-out trajectory that can be set using the `handle_unfinished` activity.

Value

Returns the trajectory object.

See Also

handle_unfinished, renege_in

Examples

```
set.seed(1234)

traj <- trajectory() %>%
  log_("leave with some probability") %>%
  leave(function() runif(1) < 0.5) %>%
  log_("didn't leave")

simmer() %>%
  add_generator("dummy", traj, at(0, 1)) %>%
  run() %>% invisible
```

---

length.trajectory *Number of Activities in a Trajectory*

Description

Get the number of activities in a trajectory. length returns the number of first-level activities (sub-trajectories not included). get_n_activities returns the total number of activities (sub-trajectories included).

Usage

```
## S3 method for class 'trajectory'
length(x)

get_n_activities(x)
```

Arguments

x	the trajectory object.

Value

Returns a non-negative integer of length 1.

See Also

Extract.trajectory, join.

Examples

```
x <- trajectory() %>%
  timeout(1)

x <- x %>%
  clone(2, x, x)
x

## length does not account for subtrajectories
length(x)
get_n_activities(x)
```

`log_` *Debugging*

Description

Activities for displaying messages preceded by the simulation time and the name of the arrival, and for setting conditional breakpoints.

Usage

```
log_(.trj, message, level = 0)

stop_if(.trj, condition)
```

Arguments

`.trj`	the trajectory object.
`message`	the message to display, accepts either a string or a callable object (a function) which must return a string.
`level`	debugging level. The `message` will be printed if, and only if, the `level` provided is less or equal to the `log_level` defined in the simulation environment (see `simmer`).
`condition`	a boolean or a function returning a boolean.

Value

Returns the trajectory object.

Examples

```
## log levels
traj <- trajectory() %>%
  log_("this is always printed") %>% # level = 0 by default
  log_("this is printed if `log_level>=1`", level = 1) %>%
  log_("this is printed if `log_level>=2`", level = 2)

simmer() %>%
  add_generator("dummy", traj, at(0)) %>%
  run() %>% invisible

simmer(log_level = 1) %>%
  add_generator("dummy", traj, at(0)) %>%
  run() %>% invisible

simmer(log_level = Inf) %>%
  add_generator("dummy", traj, at(0)) %>%
  run() %>% invisible
```

`monitor` *Create a Monitor*

Description

Methods for creating `monitor` objects for simulation environments.

Usage

```
monitor(name, xptr, get_arrivals, get_attributes, get_resources,
  handlers = NULL, finalize = function() { })

monitor_mem()

monitor_delim(path = tempdir(), keep = FALSE, sep = " ",
  ext = ".txt", reader = read.delim, args = list(stringsAsFactors =
  FALSE))

monitor_csv(path = tempdir(), keep = FALSE, reader = read.csv,
  args = list(stringsAsFactors = FALSE))
```

Arguments

`name`	an identifier to show when printed.
`xptr`	an external pointer pointing to a C++ object derived from the abstract class simmer::Monitor. See C++ API for further details and, in particular, the `simmer/monitor.h` header.

`get_arrivals` a function to retrieve the arrivals tables. It must accept the `xptr` as a first argument, even if it is not needed, and a boolean `per_resource` as a second argument (see `get_mon_arrivals`).

`get_attributes` a function to retrieve the attributes table. It must accept the `xptr` as a first argument, even if it is not needed.

`get_resources` a function to retrieve the resources table. It must accept the `xptr` as a first argument, even if it is not needed.

`handlers` an optional list of handlers that will be stored in a slot of the same name. For example, `monitor_mem` does not use this slot, but `monitor_delim` and `monitor_csv` store the path to the created files.

`finalize` an optional function to be called when the object is destroyed. For example, `monitor_mem` does not require any finalizer, but `monitor_delim` and `monitor_csv` use this to remove the created files when the monitor is destroyed.

`path` directory where files will be created (must exist).

`keep` whether to keep files on exit. By default, files are removed.

`sep` separator character.

`ext` file extension to use.

`reader` function that will be used to read the files.

`args` a list of further arguments for `reader`.

Details

The `monitor` method is a generic function to instantiate a `monitor` object. It should not be used in general unless you want to extend `simmer` with a custom monitor.

The in-memory monitor is enabled by default (`memory_mem`), and it should the fastest.

For large simulations, or if the RAM footprint is an issue, you may consider monitoring to disk. To that end, `monitor_delim` stores the values in flat delimited files. The usual `get_mon_*` methods retrieve data frames from such files using the `reader` provided. By default, `read.delim` is used, but you may consider using faster alternatives from other packages. It is also possible to `keep` the files in a custom directory to read and post-process them in a separate workflow.

`monitor_csv` is a special case of `monitor_delim` with `sep=","` and `ext=".csv"`.

Value

A `monitor` object.

Examples

```
mon <- monitor_csv()
mon

env <- simmer(mon=mon) %>%
  add_generator("dummy", trajectory() %>% timeout(1), function() 1) %>%
```

```
  run(10)
env

read.csv(mon$handlers$arrivals) # direct access
get_mon_arrivals(env)           # adds the "replication" column
```

`now` *Simulation Time*

Description

Get the current simulation time.

Usage

```
now(.env)
```

Arguments

`.env`	the simulation environment.

Value

Returns a numeric value.

See Also

`peek`.

`peek` *Peek Next Events*

Description

Look for future events in the event queue and (optionally) obtain info about them.

Usage

```
peek(.env, steps = 1, verbose = FALSE)
```

Arguments

`.env`	the simulation environment.
`steps`	number of steps to peek.
`verbose`	show additional information (i.e., the name of the process) about future events.

Value

Returns numeric values if `verbose=F` and a data frame otherwise.

See Also

`now`.

`renege_in` *Renege on some Condition*

Description

Activities for setting or unsetting a timer or a signal after which the arrival will abandon.

Usage

```
renege_in(.trj, t, out = NULL, keep_seized = FALSE)

renege_if(.trj, signal, out = NULL, keep_seized = FALSE)

renege_abort(.trj)
```

Arguments

`.trj`	the trajectory object.
`t`	timeout to trigger reneging, accepts either a numeric or a callable object (a function) which must return a numeric.
`out`	optional sub-trajectory in case of reneging.
`keep_seized`	whether to keep already seized resources. By default, all resources are released.
`signal`	signal to trigger reneging, accepts either a string or a callable object (a function) which must return a string.

Details

Note that `renege_if` works similarly to `trap`, but in contrast to that, reneging is triggered even if the arrival is waiting in a queue or is part of a non-permanent `batch`.

Value

Returns the trajectory object.

See Also

`send`, `leave`

Examples

```
bank <- trajectory() %>%
  log_("here I am") %>%
  # renege in 5 minutes
  renege_in(
    5,
    out = trajectory() %>%
      log_("lost my patience. Reneging...")) %>%
  seize("clerk") %>%
  # stay if I'm being attended within 5 minutes
  renege_abort() %>%
  log_("I'm being attended") %>%
  timeout(10) %>%
  release("clerk") %>%
  log_("finished")

simmer() %>%
  add_resource("clerk", 1) %>%
  add_generator("customer", bank, at(0, 1)) %>%
  run() %>% invisible
```

---

reset *Reset a Simulator*

Description

Reset the following components of a simulation environment: time, event queue, resources, sources and statistics.

Usage

```
reset(.env)
```

Arguments

.env	the simulation environment.

Value

Returns the simulation environment.

See Also

stepn, run.

`rollback` *Rollback a Number of Activities*

Description

Activity for going backwards to a previous point in the trajectory. Useful to implement loops.

Usage

```
rollback(.trj, amount, times = Inf, check = NULL)
```

Arguments

`.trj`	the trajectory object.
`amount`	the amount of activities (of the same or parent trajectories) to roll back.
`times`	the number of repetitions until an arrival may continue.
`check`	a callable object (a function) which must return a boolean. If present, the `times` parameter is ignored, and the activity uses this function to check whether the rollback must be done or not.

Value

Returns the trajectory object.

Examples

```
## rollback a specific number of times
traj <- trajectory() %>%
  log_("hello!") %>%
  timeout(1) %>%
  rollback(2, 3)

simmer() %>%
  add_generator("hello_sayer", traj, at(0)) %>%
  run() %>% invisible

## custom check
env <- simmer()

traj <- trajectory() %>%
  set_attribute("var", 0) %>%
  log_(function()
    paste("attribute level is at:", get_attribute(env, "var"))) %>%
  set_attribute("var", 25, mod="+") %>%
  rollback(2, check=function() get_attribute(env, "var") < 100) %>%
  log_("done")

env %>%
```

```
add_generator("dummy", traj, at(0)) %>%
run() %>% invisible
```

`run` *Run a Simulation*

Description

Execute steps until a given criterion.

Usage

```
run(.env, until = Inf, progress = NULL, steps = 10)

stepn(.env, n = 1)
```

Arguments

`.env`	the simulation environment.
`until`	stop time.
`progress`	optional callback to show the progress of the simulation. The completed ratio is periodically passed as argument to the callback.
`steps`	number of steps to show as progress (it takes effect only if `progress` is provided).
`n`	number of events to simulate.

Value

Returns the simulation environment.

See Also

`reset`.

schedule *Generate a Scheduling Object*

Description

Resource convenience function to generate a scheduling object from a timetable specification.

Usage

```
schedule(timetable, values, period = Inf)
```

Arguments

timetable	absolute points in time in which the desired value changes.
values	one value for each point in time.
period	period of repetition.

Value

Returns a `schedule` object.

See Also

`add_resource`.

Examples

```
# Schedule 3 units from 8 to 16 h
#          2 units from 16 to 24 h
#          1 units from 24 to 8 h
capacity_schedule <- schedule(c(8, 16, 24), c(3, 2, 1), period=24)

env <- simmer() %>%
  add_resource("dummy", capacity_schedule)
```

seize *Seize/Release Resources*

Description

Activities for seizing/releasing a resource, by name or a previously selected one. Resources must be defined in the simulation environment (see `add_resource`).

Usage

```
seize(.trj, resource, amount = 1, continue = NULL, post.seize = NULL,
  reject = NULL)

seize_selected(.trj, amount = 1, id = 0, continue = NULL,
  post.seize = NULL, reject = NULL)

release(.trj, resource, amount = 1)

release_selected(.trj, amount = 1, id = 0)

release_all(.trj, resource)

release_selected_all(.trj, id = 0)
```

Arguments

`.trj`	the trajectory object.
`resource`	the name of the resource.
`amount`	the amount to seize/release, accepts either a numeric or a callable object (a function) which must return a numeric.
`continue`	a boolean (if `post.seize` OR `reject` is defined) or a pair of booleans (if `post.seize` AND `reject` are defined; if only one value is provided, it will be recycled) to indicate whether these subtrajectories should continue to the next activity in the main trajectory.
`post.seize`	an optional trajectory object which will be followed after a successful seize.
`reject`	an optional trajectory object which will be followed if the arrival is rejected. Note that if the arrival is accepted (either in the queue or in the server) and then it is dropped afterwards due to preemption or resource shrinkage, then this trajectory won't be executed. Instead, see `handle_unfinished` for another, more general, method for handling all kinds of unfinished arrivals.
`id`	selection identifier for nested usage.

Value

Returns the trajectory object.

See Also

`select`, `set_capacity`, `set_queue_size`, `set_capacity_selected`, `set_queue_size_selected`

Examples

```
## simple seize, delay, then release
traj <- trajectory() %>%
  seize("doctor", 1) %>%
  timeout(3) %>%
```

```
  release("doctor", 1)

simmer() %>%
  add_resource("doctor", capacity=1) %>%
  add_generator("patient", traj, at(0, 1)) %>%
  run() %>%
  get_mon_resources()

## arrival rejection (no space left in the queue)
traj <- trajectory() %>%
  log_("arriving...") %>%
  seize("doctor", 1) %>%
  # the second patient won't reach this point
  log_("doctor seized") %>%
  timeout(5) %>%
  release("doctor", 1)

simmer() %>%
  add_resource("doctor", capacity=1, queue_size=0) %>%
  add_generator("patient", traj, at(0, 1)) %>%
  run() %>% invisible

## capturing rejection to retry
traj <- trajectory() %>%
  log_("arriving...") %>%
  seize(
    "doctor", 1, continue = FALSE,
    reject = trajectory() %>%
      log_("rejected!") %>%
      # go for a walk and try again
      timeout(2) %>%
      log_("retrying...") %>%
      rollback(amount = 4, times = Inf)) %>%
  # the second patient will reach this point after a couple of walks
  log_("doctor seized") %>%
  timeout(5) %>%
  release("doctor", 1) %>%
  log_("leaving")

simmer() %>%
  add_resource("doctor", capacity=1, queue_size=0) %>%
  add_generator("patient", traj, at(0, 1)) %>%
  run() %>% invisible

## combining post.seize and reject
traj <- trajectory() %>%
  log_("arriving...") %>%
  seize(
    "doctor", 1, continue = c(TRUE, TRUE),
    post.seize = trajectory("admitted patient") %>%
      log_("admitted") %>%
      timeout(5) %>%
      release("doctor", 1),
```

```
    reject = trajectory("rejected patient") %>%
      log_("rejected!") %>%
      seize("nurse", 1) %>%
      timeout(2) %>%
      release("nurse", 1)) %>%
  # both patients will reach this point, as continue = c(TRUE, TRUE)
  timeout(10) %>%
  log_("leaving...")

simmer() %>%
  add_resource("doctor", capacity=1, queue_size=0) %>%
  add_resource("nurse", capacity=10, queue_size=0) %>%
  add_generator("patient", traj, at(0, 1)) %>%
  run() %>% invisible
```

`select` *Select Resources*

Description

Activity for selecting a resource for a subsequent seize/release or setting its parameters (capacity or queue size). Resources must be defined in the simulation environment (see `add_resource`).

Usage

```
select(.trj, resources, policy = c("shortest-queue",
  "shortest-queue-available", "round-robin", "round-robin-available",
  "first-available", "random", "random-available"), id = 0)
```

Arguments

`.trj`	the trajectory object.
`resources`	one or more resource names, or a callable object (a function) which must return one or more resource names.
`policy`	if `resources` is a character vector, this parameter determines the criteria for selecting a resource among the set of policies available (see details).
`id`	selection identifier for nested usage.

Details

The 'shortest-queue' policy selects the least busy resource; 'round-robin' selects resources in cyclical order; 'first-available' selects the first resource available, and 'random' selects a resource randomly.

All the 'available'-ending policies ('first-available', but also 'shortest-queue-available', 'round-robin-available' and 'random-available') check for resource availability (i.e., whether the capacity is non-zero), and exclude from the selection procedure those resources with capacity set to zero. This means that, for these policies, an error will be raised if all resources are unavailable.

Value

Returns the trajectory object.

See Also

`seize_selected`, `release_selected`, `set_capacity_selected`, `set_queue_size_selected`

Examples

```
## predefined policy
traj <- trajectory() %>%
  select(paste0("doctor", 1:3), "round-robin") %>%
  seize_selected(1) %>%
  timeout(5) %>%
  release_selected(1)

simmer() %>%
  add_resource("doctor1") %>%
  add_resource("doctor2") %>%
  add_resource("doctor3") %>%
  add_generator("patient", traj, at(0, 1, 2)) %>%
  run() %>%
  get_mon_resources()

## custom policy
env <- simmer()
res <- paste0("doctor", 1:3)

traj <- trajectory() %>%
  select(function() {
    occ <- get_server_count(env, res) + get_queue_count(env, res)
    res[which.min(occ)[1]]
  }) %>%
  seize_selected(1) %>%
  timeout(5) %>%
  release_selected(1)

for (i in res) env %>%
  add_resource(i)
env %>%
  add_generator("patient", traj, at(0, 1, 2)) %>%
  run() %>%
  get_mon_resources()
```

`send` *Inter-arrival Communication*

Description

These activities enable asynchronous programming. `send()` broadcasts a signal or a list of signals. Arrivals can subscribe to signals and (optionally) assign a handler with `trap()`. Note that, while inside a batch, all the signals subscribed before entering the batch are ignored. Upon a signal reception, the arrival stops the current activity and executes the handler (if provided). Then, the execution returns to the activity following the point of the interruption. `untrap()` can be used to unsubscribe from signals. `wait()` blocks until a signal is received.

Usage

```
send(.trj, signals, delay = 0)

trap(.trj, signals, handler = NULL, interruptible = TRUE)

untrap(.trj, signals)

wait(.trj)
```

Arguments

`.trj`	the trajectory object.
`signals`	signal or list of signals, accepts either a string, a list of strings or a callable object (a function) which must return a string or a list of strings.
`delay`	optional timeout to trigger the signals, accepts either a numeric or a callable object (a function) which must return a numeric.
`handler`	optional trajectory object to handle a signal received.
`interruptible`	whether the handler can be interrupted by signals.

Value

Returns the trajectory object.

See Also

`renege_if`

Examples

```
## block, signal and continue with a handler
signal <- "you shall pass"

t_blocked <- trajectory() %>%
  trap(
    signal,
    trajectory() %>%
      log_("executing the handler")) %>%
  log_("waiting...") %>%
  wait() %>%
```

```
  log_("continuing!")

t_signaler <- trajectory() %>%
  log_(signal) %>%
  send(signal)

simmer() %>%
  add_generator("blocked", t_blocked, at(0)) %>%
  add_generator("signaler", t_signaler, at(5)) %>%
  run() %>% invisible

## handlers can be interrupted, unless interruptible=FALSE
t_worker <- trajectory() %>%
  trap(
  signal,
  handler = trajectory() %>%
    log_("ok, I'm packing...") %>%
    timeout(1)) %>%
  log_("performing a looong task...") %>%
  timeout(100) %>%
  log_("and I'm leaving!")

simmer() %>%
  add_generator("worker", t_worker, at(0)) %>%
  add_generator("signaler", t_signaler, at(5, 5.5)) %>%
  run() %>% invisible
```

`set_attribute` *Set Attributes*

Description

Activity for modifying attributes. Attributes defined with `set_attribute` are *per arrival*, meaning that each arrival has its own set of attributes, not visible by any other one. On the other hand, attributes defined with `set_global` are shared by all the arrivals in the simulation.

Usage

```
set_attribute(.trj, keys, values, mod = c(NA, "+", "*"), init = 0)

set_global(.trj, keys, values, mod = c(NA, "+", "*"), init = 0)
```

Arguments

`.trj`	the trajectory object.
`keys`	the attribute name(s), or a callable object (a function) which must return attribute name(s).

values	numeric value(s) to set, or a callable object (a function) which must return numeric value(s).
mod	if set, values modify the attributes rather than substituting them.
init	initial value, applied if mod is set and the attribute was not previously initialised. Useful for counters or indexes.

Details

Attribute monitoring is disabled by default. To enable it, set mon=2 in the corresponding source (see, e.g., add_generator). Then, the evolution of the attributes during the simulation can be retrieved with get_mon_attributes. Global attributes are reported as unnamed key/value pairs.

Value

Returns the trajectory object.

See Also

get_attribute, get_global, timeout_from_attribute, timeout_from_global

Examples

```
env <- simmer()

traj <- trajectory() %>%

  # simple assignment
  set_attribute("my_key", 123) %>%
  set_global("global_key", 321) %>%

  # more than one assignment at once
  set_attribute(c("my_key", "other_key"), c(5, 64)) %>%

  # increment
  set_attribute("my_key", 1, mod="+") %>%

  # assignment using a function
  set_attribute("independent_key", function() runif(1)) %>%

  # assignment dependent on another attribute
  set_attribute("dependent_key", function()
    ifelse(get_attribute(env, "my_key") <= 0.5, 1, 0))

env %>%
  add_generator("dummy", traj, at(3), mon=2) %>%
  run() %>%
  get_mon_attributes()
```

set_capacity *Set Resource Parameters*

Description

Activities for dynamically modifying a resource's server capacity or queue size, by name or a previously selected one. Resources must be defined in the simulation environment (see `add_resource`).

Usage

```
set_capacity(.trj, resource, value, mod = c(NA, "+", "*"))

set_capacity_selected(.trj, value, id = 0, mod = c(NA, "+", "*"))

set_queue_size(.trj, resource, value, mod = c(NA, "+", "*"))

set_queue_size_selected(.trj, value, id = 0, mod = c(NA, "+", "*"))
```

Arguments

`.trj`	the trajectory object.
`resource`	the name of the resource.
`value`	new value to set.
`mod`	if set, `values` modify the attributes rather than substituting them.
`id`	selection identifier for nested usage.

Value

Returns the trajectory object.

See Also

`select`, `seize`, `release`, `seize_selected`, `release_selected`, `get_capacity`, `get_queue_size`

Examples

```
## a resource with a queue size equal to the number of arrivals waiting
traj <- trajectory() %>%
  set_queue_size("res", 1, mod="+") %>%
  seize("res") %>%
  set_queue_size("res", -1, mod="+") %>%
  timeout(10) %>%
  release("res")

simmer() %>%
  add_resource("res", 1, 0) %>%
```

```
  add_generator("dummy", traj, at(0:2)) %>%
  run() %>%
  get_mon_resources()
```

`set_prioritization` *Set Prioritization Values*

Description

Activity for dynamically modifying an arrival's prioritization values. Default prioritization values are defined by the source (see `add_generator`, `add_dataframe`).

Usage

```
set_prioritization(.trj, values, mod = c(NA, "+", "*"))
```

Arguments

`.trj`	the trajectory object.
`values`	expects either a vector/list or a callable object (a function) returning a vector/list of three values `c(priority,preemptible,restart)`. A negative value leaves the corresponding parameter unchanged. See `add_generator` for more information about these parameters.
`mod`	if set, `values` modify the attributes rather than substituting them.

Value

Returns the trajectory object.

See Also

`get_prioritization`

Examples

```
traj <- trajectory() %>%

  # static values
  set_prioritization(c(3, 7, TRUE)) %>%

  # increment
  set_prioritization(c(2, 1, 0), mod="+") %>%

  # dynamic, custom
  set_attribute("priority", 3) %>%
  set_prioritization(function() {
    prio <- get_prioritization(env)
    attr <- get_attribute(env, "priority")
```

```
    c(attr, prio[[2]]+1, FALSE)
  })
```

---

set_trajectory *Set Source Parameters*

Description

Activities for modifying a source's trajectory or source object by name. Sources must be defined in the simulation environment (see add_generator, add_dataframe).

Usage

```
set_trajectory(.trj, source, trajectory)

set_source(.trj, source, object)
```

Arguments

.trj	the trajectory object.
source	the name of the source or a function returning a name.
trajectory	the trajectory that the generated arrivals will follow.
object	a function modelling the interarrival times (if the source type is a generator; returning a negative value stops the generator) or a data frame (if the source type is a data source).

Value

Returns the trajectory object.

See Also

activate, deactivate

Examples

```
traj1 <- trajectory() %>%
  timeout(1)

traj2 <- trajectory() %>%
  set_source("dummy", function() 1) %>%
  set_trajectory("dummy", traj1) %>%
  timeout(2)

simmer() %>%
  add_generator("dummy", traj2, function() 2) %>%
  run(6) %>%
  get_mon_arrivals()
```

simmer *Create a Simulator*

Description

This method initialises a simulation environment.

Usage

```
simmer(name = "anonymous", verbose = FALSE, mon = monitor_mem(),
  log_level = 0)
```

Arguments

name	the name of the simulator.
verbose	enable showing activity information.
mon	monitor (in memory by default); see monitor for other options.
log_level	debugging level (see log_).

Value

Returns a simulation environment.

See Also

Available methods by category:

- Simulation control: stepn, run, now, peek, reset
- Resources: add_resource, get_resources, get_capacity, get_queue_size, get_server_count, get_queue_count, get_capacity_selected, get_queue_size_selected, get_server_count_selected, get_queue_count_selected, get_seized, get_seized_selecte get_selected
- Sources: add_generator, add_dataframe, get_sources, get_n_generated, get_trajectory
- Globals: add_global, get_global
- Data retrieval: get_mon_arrivals, get_mon_attributes, get_mon_resources

Examples

```
## a simple trajectory that prints a message
t0 <- trajectory("my trajectory") %>%
  log_("arrival generated")

## create an empty simulation environment
env <- simmer("SuperDuperSim")
env
```

```
## add a generator and attach it to the trajectory above
env %>% add_generator("dummy", t0, function() 1)

## run for some time
env %>% run(until=4.5)
env %>% now()           # current simulation time
env %>% peek()          # time for the next event
env %>% stepn()         # execute next event
```

`timeout` *Delay*

Description

Activity for inserting delays and execute user-defined tasks.

Usage

```
timeout(.trj, task)

timeout_from_attribute(.trj, key)

timeout_from_global(.trj, key)
```

Arguments

`.trj`	the trajectory object.
`task`	the timeout duration supplied by either passing a numeric or a callable object (a function) which must return a numeric (negative values are automatically coerced to positive).
`key`	the attribute name, or a callable object (a function) which must return the attribute name.

Value

Returns the trajectory object.

See Also

`set_attribute`, `set_global`

`set_attribute`, `set_global`

Examples

```
env <- simmer()

traj <- trajectory() %>%

  # static delay
  timeout(3) %>%

  # dynamic, exponential delay
  timeout(function() rexp(1, 10)) %>%

  # dependent on an attribute
  set_attribute("delay", 2) %>%
  set_global("other", function() rexp(1, 2)) %>%
  timeout_from_attribute("delay") %>%
  timeout_from_global("other")

env %>%
  add_generator("dummy", traj, at(0)) %>%
  run() %>%
  get_mon_arrivals()
```

---

trajectory *Create a Trajectory*

Description

This method initialises a trajectory object, which comprises a chain of activities that can be attached to a generator. See below for a complete list of available activities by category.

Usage

```
trajectory(name = "anonymous", verbose = FALSE)
```

Arguments

name	the name of the trajectory.
verbose	enable showing additional information.

Value

Returns an environment that represents the trajectory.

See Also

Available activities by category:

- Debugging: `log_`, `stop_if`
- Delays: `timeout`, `timeout_from_attribute`, `timeout_from_global`
- Arrival properties: `set_attribute`, `set_global`, `set_prioritization`
- Interaction with resources: `select`, `seize`, `release`, `release_all`, `seize_selected`, `release_selected`, `release_selected_all`, `set_capacity`, `set_queue_size`, `set_capacity_selected`, `set_queue_size_selected`
- Interaction with generators: `activate`, `deactivate`, `set_trajectory`, `set_source`
- Branching: `branch`, `clone`, `synchronize`
- Loops: `rollback`
- Batching: `batch`, `separate`
- Asynchronous programming: `send`, `trap`, `untrap`, `wait`
- Reneging: `leave`, `handle_unfinished`, `renege_in`, `renege_if`, `renege_abort`

Manage trajectories:

- Extract or Replace Parts of a Trajectory: `Extract.trajectory`
- Join Trajectories: `join`
- Number of Activities in a Trajectory: `length.trajectory`, `get_n_activities`

Examples

```
## create an empty trajectory
x <- trajectory("my trajectory")
x

## add some activities by chaining them
x <- x %>%
  log_("here I am!") %>%
  timeout(5) %>%
  log_("leaving!")
x

## join trajectories
x <- join(x, x)

## extract and replace
x[c(3, 4)] <- x[2]
x
```

`wrap` *Wrap a Simulation Environment*

Description

This function extracts the monitored data from a simulation environment making it accessible through the same methods. Only useful if you want to parallelize heavy replicas (see the example below), because the C++ simulation backend is destroyed when the threads exit.

Usage

```
wrap(.env)
```

Arguments

`.env`	the simulation environment.

Value

Returns a simulation wrapper.

See Also

Methods for dealing with a simulation wrapper: `get_mon_arrivals`, `get_mon_attributes`, `get_mon_resources`, `get_n_generated`, `get_capacity`, `get_queue_size`, `get_server_count`, `get_queue_count`.

Examples

```
## Not run:
library(parallel)

mm1 <- trajectory() %>%
  seize("server", 1) %>%
  timeout(function() rexp(1, 2)) %>%
  release("server", 1)

envs <- mclapply(1:4, function(i) {
  simmer("M/M/1 example") %>%
    add_resource("server", 1) %>%
    add_generator("customer", mm1, function() rexp(1, 1)) %>%
    run(100) %>%
    wrap()
})

## End(Not run)
```