Package 'simmer.bricks'

January 16, 2018

Type Package

Title Helper Methods for 'simmer' Trajectories

Version 0.1.0

Description Provides wrappers for common activity patterns in 'simmer' trajectories.

License MIT + file LICENSE

Encoding UTF-8

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

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

Depends R (>= 3.1.2), simmer (>= 3.6.5)

Suggests testthat, knitr, rmarkdown

ByteCompile yes

RoxygenNote 6.0.1

VignetteBuilder knitr

NeedsCompilation no

Author Iñaki Ucar [aut, cph, cre] (0000-0001-6403-5550)

Maintainer Iñaki Ucar <i.ucar86@gmail.com>

Repository CRAN

Date/Publication 2018-01-16 11:15:58 UTC

R topics documented:

`simmer.bricks-package` **simmer.bricks***: Helper Methods for* **simmer** *Trajectories*

Description

Provides wrappers for common activity patterns in **simmer** trajectories.

Author(s)

Iñaki Ucar

See Also

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

`delayed_release` *Delayed Release of a Resource*

Description

This brick encapsulates a delayed release: the arrival releases the resource and continues its way immediately, but the resource is busy for an additional period of time.

Usage

```
delayed_release(.trj, .env, resource, task, amount = 1, preemptive = FALSE,
  mon_all = FALSE)
```

Arguments

`.trj`	the trajectory object.
`.env`	the simulation environment.
`resource`	the name of the resource.
`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).
`amount`	the amount to seize/release, accepts either a numeric or a callable object (a function) which must return a numeric.
`preemptive`	whether arrivals in the server can be preempted or not based on seize priorities.
`mon_all`	if `TRUE`, `get_mon_arrivals` will show one line per clone.

Value

Returns the following chain of activities: clone > synchronize (see examples below).

Examples

```
env <- simmer()

## These are equivalent if the resource is non-preemptive:
trajectory() %>%
  delayed_release(env, "res1", 5, 1)

trajectory() %>%
  clone(
    2,
    trajectory() %>%
      set_capacity("res1", function()
        get_capacity(env, "res1") - 1) %>%
      release("res1", 1),
    trajectory() %>%
      timeout(5) %>%
      set_capacity("res1", function()
        get_capacity(env, "res1") + 1)
  ) %>%
  synchronize(wait=FALSE)

## These are equivalent if the resource is preemptive:
trajectory() %>%
  delayed_release(env, "res2", 5, 1, preemptive=TRUE)

trajectory() %>%
  clone(
    2,
    trajectory() %>%
      release("res2", 1),
    trajectory() %>%
      set_prioritization(function()
        get_prioritization(env) + c(rep(.Machine$integer.max, 2), 0)) %>%
      seize("res2", 1) %>%
      timeout(5) %>%
      release("res2", 1)
  ) %>%
  synchronize(wait=FALSE)
```

---

do_parallel *Perform Parallel Tasks*

Description

This brick encapsulates the activity of n workers running parallel sub-trajectories.

Usage

```
do_parallel(.trj, .env, ..., wait = TRUE, mon_all = FALSE)
```

Arguments

.trj	the trajectory object.
.env	the simulation environment.
...	parallel sub-trajectories.
wait	if TRUE, the arrival waits until all parallel sub-trajectories are finished; if FALSE, the arrival continues as soon as the first parallel task ends.
mon_all	if TRUE, get_mon_arrivals will show one line per clone.

Value

Returns the following chain of activities: clone > synchronize (> wait > untrap if wait=FALSE) (see examples below).

Examples

```
env <- simmer()
signal <- function() get_name(env)

task.1 <- trajectory("task 1") %>%
  timeout(function() rexp(1))
task.2 <- trajectory("task 2") %>%
  timeout(function() rexp(1))

## These are equivalent:
trajectory() %>%
  do_parallel(
    env, wait = TRUE,
    task.1,
    task.2
  )

trajectory() %>%
  clone(
    n = 3,
    trajectory("original") %>%
      trap(signal) %>%
      wait() %>%
      wait() %>%
      untrap(signal),
    task.1[] %>%
      send(signal),
    task.2[] %>%
      send(signal)) %>%
  synchronize(wait = TRUE)

## These are equivalent:
```

```
trajectory() %>%
  do_parallel(
    env, wait = FALSE,
    task.1,
    task.2
  )

trajectory() %>%
  clone(
    n = 3,
    trajectory("original") %>%
      trap(signal),
    task.1[] %>%
      send(signal),
    task.2[] %>%
      send(signal)) %>%
  synchronize(wait = FALSE) %>%
  wait() %>%
  untrap(signal)
```

`visit` *Visit a Resource*

Description

These bricks encapsulate a resource visit: seize, spend some time and release.

Usage

```
visit(.trj, resource, task, amount = 1)

visit_selected(.trj, task, amount = 1, id = 0)
```

Arguments

`.trj`	the trajectory object.
`resource`	the name of the resource.
`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).
`amount`	the amount to seize/release, accepts either a numeric or a callable object (a function) which must return a numeric.
`id`	selection identifier for nested usage.

Value

Returns the following chain of activities: `seize` > `timeout` > `release` (see examples below).

Examples

```
## These are equivalent:
trajectory() %>%
  visit("res", 5, 1)

trajectory() %>%
  seize("res", 1) %>%
  timeout(5) %>%
  release("res", 1)

## These are equivalent:
trajectory() %>%
  visit_selected(5, 1)

trajectory() %>%
  seize_selected(1) %>%
  timeout(5) %>%
  release_selected(1)
```

`wait_n` *Wait a Number of Signals*

Description

These bricks encapsulate `n` stops: wait for a sequence of `n` signals. `wait_until` also traps and untraps the required signals.

Usage

```
wait_n(.trj, n = 1)

wait_until(.trj, signals, n = 1)
```

Arguments

`.trj`	the trajectory object.
`n`	number of `wait` activities to chain.
`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.

Value

`wait_n` returns `n` times `wait`. `wait_until` also adds `trap` and `untrap` at the beginning and end, respectively, of the chain of `wait`s (see examples below).

Examples

```
## These are equivalent:
trajectory() %>%
  wait_n(3)

trajectory() %>%
  wait() %>%
  wait() %>%
  wait()

## These are equivalent:
trajectory() %>%
  wait_until("green")

trajectory() %>%
  trap("green") %>%
  wait() %>%
  untrap("green")

## These are equivalent:
trajectory() %>%
  wait_until(c("one", "another"), 2)

trajectory() %>%
  trap(c("one", "another")) %>%
  wait() %>%
  wait() %>%
  untrap(c("one", "another"))
```

Index