Package ‘ABM’

Type Package

Title Agent Based Model Simulation Framework

Version 0.4.1

Date 2024-03-14

Description A high-performance, flexible and extensible framework to develop continuous-time agent based models. Its high performance allows it to simulate millions of agents efficiently. Agents are defined by their states (arbitrary R lists). The events are handled in chronological order. This avoids the multi-event interaction problem in a time step of discrete-time simulations, and gives precise outcomes. The states are modified by provided or user-defined events. The framework provides a flexible and customizable implementation of state transitions (either spontaneous or caused by agent interactions), making the framework suitable to apply to epidemiology and ecology, e.g., to model life history stages, competition and cooperation, and disease and information spread. The agent interactions are flexible and extensible. The framework provides random mixing and network interactions, and supports multi-level mixing patterns. It can be easily extended to other interactions such as inter- and intra-households (or workplaces and schools) by subclassing an R6 class. It can be used to study the effect of age-specific, group-specific, and contact- specific intervention strategies, and complex interactions between individual behavior and population dynamics. This modeling concept can also be used in business, economical and political models. As a generic event based framework, it can be applied to many other fields. More information about the implementation and examples can be found at <https://github.com/junlingm/ABM>.

License GPL (>= 2)

URL https://github.com/junlingm/ABM

BugReports https://github.com/junlingm/ABM/issues

Imports R6, Rcpp

LinkingTo Rcpp
Encoding UTF-8
RoxygenNote 7.2.3
NeedsCompilation yes
Author Junling Ma [aut, cre]
Maintainer Junling Ma <junlingm@uvic.ca>
Repository CRAN
Date/Publication 2024-03-14 22:40:02 UTC

R topics documented:

ABM-package .................................................. 3
addAgent ..................................................... 6
Agent .......................................................... 6
clearEvents .................................................. 8
Contact ........................................................ 9
Event ........................................................... 11
getAgent ....................................................... 12
getID ........................................................... 13
getSize ........................................................ 13
getState ....................................................... 14
getime ......................................................... 14
getWaitingTime .............................................. 15
leave ......................................................... 15
matchState .................................................. 16
newAgent ...................................................... 16
newConfigurationModel .................................... 17
newCounter .................................................. 17
newEvent ..................................................... 18
newExpWaitingTime ....................................... 19
newGammaWaitingTime .................................... 19
newPopulation ............................................. 20
newRandomMixing ......................................... 20
newStateLogger ............................................. 21
Population ................................................... 21
schedule ...................................................... 24
setDeathTime ............................................... 24
setState ..................................................... 25
setState ..................................................... 25
Simulation ................................................... 26
State .......................................................... 28
stateMatch .................................................. 29
unschedule ................................................... 29

Index 30
Description

This package provides a framework to simulate agent based models that are based on states and events.

Details

Agent:
The concept of this framework is agent, which is an object of the `Agent` class. An agent maintains its own state, which is a named R list storing any R values in it. (see State). The main task of an agent is to manage events (see Event), and handle them in chronological order.

Population:
An object of the `Population` class manages agents and their contacts. The contacts of agents are managed by Contact objects. The main functionality for a contact object is to provide contacts of a given individuals at a given time. For example, `newRandomMixing()` returns such an object that finds a random agent in the population as a contact. the effect of contacts on the states of agents are defined using a state transition rule. Please see `addTransition` method of `Simulation` for more details.

Simulation:
The `Simulation` class inherits the `Population` class. So a simulation manages agents and their contacts. Thus, the class also inherits the `Agent` class. So a simulation can have its own state, and events attached (scheduled) to it. In addition, it also manages all the transitions, using its `addTransition` method. At last, it maintains loggers, which record (or count) the state changes, and report their values at specified times.

During a simulation the earliest event in the simulation is picked out, unscheduled (detached), and handled, which potentially causes the state change of the agent (or another agent in the simulation). The state change is then logged by loggers (see `newCounter()` and `newStateLogger()` for more details) that recognize the state change.

Usage:
To use this framework, we start by creating a simulation object, populate the simulation with agents (either using the argument in the constructor, or use its `addAgent` method), and initialize the agents with their initial states using its `setState` method.

We then attach (`schedule()`) events to agents (possibly to the populations or the simulation object too), so that these events change the agents’ state. For models which agents’ states are defined by discrete states, such as the SIR epidemic model, the events are managed by the framework through state transitions, using rules defined by the `addTransition` method of the `Simulation` class.

At last, we add loggers to the simulation using the `Simulation` class’ `addLogger` method’ and either `newCounter()` or `newStateLogger()`. At last, run the simulation using its `run` method, which returns the observations of the loggers at the requested time points as a data.frame object.

For more information and examples, please see the Wiki pages on Github.
Author(s)

Maintainer: Junling Ma <junlingm@uvic.ca>

See Also

Useful links:
- [https://github.com/junlingm/ABM](https://github.com/junlingm/ABM)
- Report bugs at [https://github.com/junlingm/ABM/issues](https://github.com/junlingm/ABM/issues)

Examples

```r
# simulate an SIR model using the Gillespie method
# the population size
N = 10000
# the initial number of infectious agents
I0 = 10
# the transmission rate
beta = 0.4
# the recovery rate
gamma = 0.2
# an waiting time egenerator that handles 0 rate properly
wait.exp = function(rate) {
  if (rate == 0) Inf else rexp(1, rate)
}
# this is a function that rescheduled all the events. When the
# state changed, the old events are invalid because they are
# calculated from the old state. This is possible because the
# waiting times are exponentially distributed
reschedule = function(time, agent, state) {
  clearEvents(agent)
  t.inf = time + wait.exp(beta*state$I*state$S/N)
  schedule(agent, newEvent(t.inf, handler.infect))
  t.rec = time + wait.exp(gamma*state$I)
  schedule(agent, newEvent(t.rec, handler.recover))
}
# The infection event handler
# an event handler take 3 arguments
# time is the current simulation time
# sim is an external pointer to the Simulation object.
# agent is the agent that the event is scheduled to
handler.infect = function(time, sim, agent) {
  x = getState(agent)
  x$S = x$S - 1
  x$I = x$I + 1
  setState(agent, x)
  reschedule(time, agent, x)
}
# The recovery event handler
handler.recover = function(time, sim, agent) {
  x = getState(agent)
  x$R = x$R + 1
```
x$I = x$I - 1
setState(agent, x)
reschedule(time, agent, x)
}
# create a new simulation with no agent in it.
# note that the simulation object itself is an agent
sim = Simulation$new()
# the initial state
x = list(S=N-I0, I=I0, R=0)
sim$state = x
# schedule an infection event and a recovery event
reschedule(0, sim$get, sim$state)
# add state loggers that saves the S, I, and R states
sim$addLogger(newStateLogger("S", NULL, "S"))
sim$addLogger(newStateLogger("I", NULL, "I"))
sim$addLogger(newStateLogger("R", sim$get, "R"))
# now the simulation is setup, and is ready to run
result = sim$run(0:100)
# the result is a data.frame object
print(result)

# simulate an agent based SEIR model
# specify an exponential waiting time for recovery
gamma = newExpWaitingTime(0.2)
# specify a transmission rate
beta = 0.4
# specify a exponentially distributed latent period
sigma = newExpWaitingTime(0.5)
# the population size
N = 10000
# create a simulation with N agents, initialize the first 5 with a state "I"
# and the remaining with "S".
sim = Simulation$new(N, function(i) if (i <= 5) "I" else "S")
# add event loggers that counts the individuals in each state.
# the first variable is the name of the counter, the second is
# the state for counting. States should be lists. However, for
# simplicity, if the state has a single value, then we
# can specify the list as the value, e.g., "S", and the state
# is equivalent to list("")
sim$addLogger(newCounter("S", "S"))
sim$addLogger(newCounter("E", "E"))
sim$addLogger(newCounter("I", "I"))
sim$addLogger(newCounter("R", "R"))
# create a random mixing contact pattern and attach it to sim
m = newRandomMixing()
sim$addContact(m)
# the transition for leaving latent state anbd becoming infectious
sim$addTransition("E"=>"I", sigma)
# the transition for recovery
sim$addTransition("I"=>"R", gamma)
# also note that the waiting time can be a number, which is the same
# as newExpWaitingTime(beta)
```r
sim$addTransition("I" + "S" -> "I" + "E" ~ m, beta) # run the simulation, and get a data.frame object
result = sim$run(0:100)
print(result)
```

---

**addAgent**  
*add an agent to a population*

---

**Description**  
add an agent to a population

**Arguments**

- `population`: an external pointer to a population, for example, one returned by `newPopulation()`  
- `agent`: an external pointer to an agent, returned by `newAgent()` or `getAgent()`

**Details**

if the agent is an R6 class, we should use `agent$get` to get the external pointer. Similarly, if population is an R6 object, then we should either use `population$addAgent()` or `population$get`.

---

**Agent**  
*R6 class that represent an agent*

---

**Description**

The key task of an agent is to maintain events, and handle them in the chronological order. Agents also maintain their states, which is a list of values. The events, when handled, operate on the state of the agent (or other agents).

**Details**

During the simulation the agent with the earliest event in the simulation is picked out, unscheduled, then its earliest event is handled, which potentially causes the state change of the agent (or another agent in the simulation). The state change is then logged by loggers that recognize the state change.

An agent itself cannot handle the event. Instead, it has to be added to a simulation (or a population that itself is added to a simulation).

Note that specifying death.time is equivalent to call the `$setDeathTime` method. Check if the state of the agent matches a given state.

At the time of death, the agent is removed from the simulation. Calling it multiple times causes the agent to die at the earliest time.
Agent

Active bindings
  state  Get/set the state of the agent
  id     Get the agent ID
  get    Get the external pointer for the agent

Methods

Public methods:
  • Agent$new()
  • Agent$match()
  • Agent$schedule()
  • Agent$unschedule()
  • Agent$leave()
  • Agent$setDeathTime()
  • Agent$clone()

Method new():
  Usage:
  Agent$new(agent = NULL, death.time = NA)
  Arguments:
  agent can be either an external pointer to an agent such as one returned by newAgent, or a list
  representing the initial state for creating a new agent, or NULL (an empty state)
  death.time the time of death for the agent, a numeric value

Method match():
  Usage:
  Agent$match(rule)
  Arguments:
  rule  the state to match, a list
  Returns: a logical value Schedule an event

Method schedule():
  Usage:
  Agent$schedule(event)
  Arguments:
  event  an object of the R6 class Event, or an external pointer returned by newEvent
  Returns: the agent itself (invisible) Unschedule an event

Method unschedule():
  Usage:
  Agent$unschedule(event)
  Arguments:
**clearEvents**

Unschedule all event from an agent

<table>
<thead>
<tr>
<th>clearEvents</th>
<th>Unschedule all event from an agent</th>
</tr>
</thead>
</table>

**Description**

Unschedule all event from an agent

**Arguments**

| agent | an external pointer returned by newAgent |

**Details**

If agent is an R6 object, then we should use either agent$clearEvents() or clearEvents(agent$get)
Contact

An R6 class that implements a contact pattern in R

Description

An R6 class that implements a contact pattern in R

Details

The main task of the class is to return the contacts of a given agent. Each object of this class is associated to a population. A population may have multiple contacts attached, e.g., a random mixing contact pattern and a network contact pattern.

This class must be subclassed in order to implement specific functionality. To subclass, we must implement three methods, namely contact, addAgent, and build. See more details in the documentation of each method.

This method should be called from the C++ side. Users should not call this directly.

When an agent is added to a population, it is added to each of the contact patterns. When a contact pattern is added to a population, all agents in a population is added to the contact pattern, one by one.

Note that, immediately before the simulation is run, while reporting the states to the simulation object, the population will call the build method for each Contact object. Thus a contact object may choose to ignore adding agents before build is called, and handle all agents within the finalize method. However, the contact object must handle adding an agent after build is called.

When an agent leaves a population, it is removed from each of the contact patterns.

This method may also be called in event handlers to remove an agent

This method is called immediately before the simulation is run, when the attached population reports the states to the simulation object.

Thus this method can be considered as a callback function to notify the contact object the population state, such as its agents, states, events, and contact patterns are all initialized, so the contact pattern should finish initialization, for example, building the contact network.

This is needed because some contact patterns, such as a configuration- model contact network, cannot be built while adding agents one by one. It must be generated when all agents are present. This is unlike the Albert-Barabasi network m which can be built while adding the agents.

Active bindings

get .The external pointer pointing to the C++ RContact object.

attached a logical value indicating whether the object has been attached to a population
Methods

Public methods:

• Contact$new()
• Contact$attach()
• Contact$contact()
• Contact$addAgent()
• Contact$remove()
• Contact$build()
• Contact$clone()

Method new(): the constructor
Usage:
Contact$new()

Method attach(): attach to a population
Usage:
Contact$attach(population)
Arguments:
population the population to attach to. An external pointer

Method contact(): Returns the contacts of the given agent
Usage:
Contact$contact(time, agent)
Arguments:
time the current time in the simulation, a number
agent the agent whose contacts are requested. An external pointer
Returns: a list of external pointers pointing to the contacting agents

Method addAgent(): Add an agent to the contact pattern
Usage:
Contact$addAgent(agent)
Arguments:
agent the agent to be added. An external pointer

Method remove(): Remove an agent from the contact pattern
Usage:
Contact$remove(agent)
Arguments:
agent the agent to be removed. An external pointer

Method build(): Build the contact pattern
Usage:
Method clone(): The objects of this class are cloneable with this method.

Usage:
Event$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

Description
R6 class to create and represent an event

Active bindings
time returns the event time
gt returns the external pointer, which can then be passed to functions such as schedule and un-schedule.

Methods

Public methods:
- Event$new()
- Event$clone()

Method new():

Usage:
Event$new(time, handler)

Arguments:
time the time that this event will occur. A length-1 numeric vector.
handler an R function that handles the event when it occurs.

Details: The R handler function should take exactly 3 arguments
1. time: the current time in the simulation
2. sim: the simulation object, an external pointer
3. agent: the agent to whom this event is attached to.
The return value of the handler function is ignored.

Method clone(): The objects of this class are cloneable with this method.

Usage:
Event$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.
Examples

```r
# This handler prints increases a counter in the state of the
# Simulation object, and schedule another event every 0.1 time unit.
handler = function(time, sim, agent) {
    x = getState(sim)
    x$counter = x$counter + 1
    setState(sim, x)
    schedule(agent, newEvent(time + 0.1, handler))
}
# create a new simulation with no agents. but the simulation itself is
# an agent. So we can use all the methods of agent
sim = Simulation$new()
# set the state of the simulation, initialize the counter
sim$state = list(counter = 0)
# schedule a new event at time 0
sim$schedule(Event$new(0, handler))
# add a logger for the counter. Note that, because sim is an R6 class
# to use it in the newStateLogger function, we need to access the
# external pointer using its $get method
sim$addLogger(newStateLogger("counter", sim$get, "counter"))
# run the simulation for 10 time units.
print(sim$run(0:10))
```

getAgent

---

**getAgent**

*Get the agent at an index in the population*

---

**Description**

Get the agent at an index in the population

**Arguments**

- `population`: an external pointer to a population, for example, one returned by `newPopulation()`
- `i`: the index of the agent, starting from 1.

**Value**

the agent at index `i` in the population.
**getID**

Get the ID of the agent.

**Description**

Get the ID of the agent.

**Arguments**

agent an external pointer returned by newAgent

**Details**

Before an agent is added to a population, its id is 0. After it is added, its id is the index in the population (starting from 1).

If agent is an R6 object, then we should either use agent$id, or use getID(agent$get)

**Value**

an integer value

**getSize**

Get the size of a population

**Description**

Get the size of a population

**Arguments**

population an external pointer to a population, for example, one returned by newPopulation()

**Value**

the population size, an integer
### getState

**Get the state of the agent**

**Description**

Get the state of the agent

**Arguments**

- **agent**
  - an external pointer returned by `newAgent`

**Details**

If `agent` is an R6 object, then we should either use `agent$schedule`, or use `schedule(agent$get, event)`

**Value**

- a list holding the state

---

### getTime

**returns the event time**

**Description**

returns the event time

**Arguments**

- **event**
  - an external pointer returned by the `newEvent` function.

**Value**

- a numeric value

  This function avoids the overhead of an R6 class, and is thus faster. This is the recommended method to get event time in an event handler.
getWaitingTime

Generate a waiting time from a WaitingTime object

Description

Generate a waiting time from a WaitingTime object

Arguments

generator an external pointer to a WaitingTime object, e.g., one returned by newExpWaitingTime or newGammaWaitingTime
time the current simulation time, a numeric value

Value

a numeric value

leave

leave the population that the agent is in

Description

leave the population that the agent is in

Arguments

agent an external pointer returned by newAgent

Details

If agent is an R6 object, then we should use either agent$leave() or leave(agent$get)
**matchState**  
*Check if the state of an agent matches a given state*

**Description**

Check if the state of an agent matches a given state

**Usage**

`matchState(agent, rule)`

**Arguments**

- **agent**: an external pointer returned by `newAgent`
- **rule**: a list holding the state to match against

**Details**

This function is equivalent to `stateMatch(getState(agent), rule)`. The state matches the rule if and only if each domain (names of the list) in rule has the same value as in state. The domains in domains of the state not listed in rule are not matched.

**Value**

a logical value

---

**

newAgent  
*Create an agent with a given state*

**Description**

Create an agent with a given state

**Arguments**

- **state**: a list giving the initial state of the agent, or NULL (an empty list)
- **death_time**: the death time for the agent, an optional numeric value.

**Details**

Setting death_time is equivalent to calling the `setDeathTime()` function.

**Value**

an external pointer pointing to the agent
**newConfigurationModel**  
*Creates a random network using the configuration model*

**Description**

Creates a random network using the configuration model

**Arguments**

- **rng**  
a function that generates random degrees

**Details**

The population must be an external pointer, not an R6 object. To use an R6 object, we should use its pointer representation from its $get method.

The function `rng` should take exactly one argument `n` for the number of degrees to generate, and should return an integer vector of length `n`.

**Value**

an external pointer.

**Examples**

```r
# creates a simulation with 100 agents
sim = Simulation$new(100)
# add a Poisson network with a mean degree 5
sim$addContact(newConfigurationModel(function(n) rpois(n, 5)))
```

---

**newCounter**  
*Create a logger of the Counter class*

**Description**

When state changes occur, it is passed to each logger, which then change its value. At the specified time points in a run, the values of the logger are reported and recorded in a data.frame object, where the columns represent variables, and rows represent the observation at each time point given to each run. Each logger has a name, which becomes the column name in the data.frame.
newEvent

**Arguments**

- **name**: the name of the counter, must be a length-1 character vector
- **from**: a list specifying state of the agent, or a character or numeric value that is equivalent to list(from). Please see the details section
- **to**: a list (can be NULL) specifying the state of the agent after the state change, or a character or numeric value that is equivalent to list(from). Please see the details section
- **initial**: the initial value of the counter. Default to 0.

**Details**

If the argument "to" is not NULL, then the counter counts the transitions from "from" to "to". Otherwise, it counts the number of agents in a state that matches the "from" argument. Specifically, if the agent jumps to "from", then the count increases by 1. If the agents jump away from "from", then the count decreases by 1.

**Value**

An external pointer that can be passed to the `Simulation` class' `$addLogger`.

---

The function `newEvent` creates a new event in R.

**Description**

Creates a new event in R

**Arguments**

- **time**: the time that this event will occur. A length-1 numeric vector.
- **handler**: an R function that handles the event when it occurs.

**Details**

The R handler function should take exactly 3 arguments:

1. time: the current time in the simulation
2. sim: the simulation object, an external pointer
3. agent: the agent to whom this event is attached to.

The return value of the handler function is ignored.

This function avoids the overhead of an R6 class, and is thus faster. This is the recommended method to create an event in an event handler.

**Value**

An external pointer, which can then be passed to functions such as schedule and unschedule.
newExpWaitingTime

Creates an exponentially distributed waiting time

Description

Creates an exponentially distributed waiting time

Arguments

rate the rate of the exponential distribution

Details

This function creates an C++ object of type ExpWaitingTime. It can be passed to addTransition or Simulation$addTransition to specify the waiting time for a transition. As a C++ object, it is faster than using an R function to generate waiting times because there is no need to call an R function from C++.

Value

an external pointer

newGammaWaitingTime

Creates an gamma distributed waiting time

Description

Creates an gamma distributed waiting time

Arguments

shape the shape parameter of the gamma distribution
scale the scale parameter of the gamma distribution, i.e., 1/rate

Details

This function creates an C++ object of type ExpWaitingTime. It can be passed to addTransition or Simulation$addTransition to specify the waiting time for a transition. As a C++ object, it is faster than using an R function to generate waiting times because there is no need to call an R function from C++.

Value

an external pointer
newPopulation  
Create a new population

Description
Create a new population

Arguments
n  an integer specifying the population size.

Details
The population will be created with "n" individuals in it. These individuals have an empty state upon created. Note that individuals can be added later by the "add" method, the initial population size is for convenience, not required

newRandomMixing  
Creates a RandomMixing object

Description
Creates a RandomMixing object

Value
an external pointer.

Examples
# creates a simulation with 100 agents
sim = Simulation$new(100)
# add a random mixing contact pattern for these agents.
sim$addContact(newRandomMixing())
newStateLogger

Create a logger of the StateLogger class

Description
When state changes occur, it is passed to each logger, which then change its value. At the specified
time points in a run, the values of the logger are reported and recorded in a data.frame object, where
the columns represent variables, and rows represent the observation at each time point given to each
run. Each logger has a name, which becomes the the column name in the data.frame.

Arguments

name the name of the logger. A length-1 character vector
agent the agent whose state will be logged. An external pointer
state.name the state name of the state of the agent to be logged. A character vector of length

Details
If a state changed happened to any agent, the specified state of the agent given by the "agent"
argument will be logged. If state.name==NULL then the state of the agent who just changed is
logged.
The agent must be an external pointer. To use an R6 object, we need to use its $get method to get
the external pointer.
The state to be logged must have a numeric value.

Population

R6 class that represents a population

Description
A population is a collection of agents. There are two important tasks for a population:

1. to manage the agents in it
2. to define the contact patterns of the agents

The contact patterns are defined by objects of the Contact class that are associated with the popu-
lation. A population may have multiple Contact objects, for example, one for random mixing, one
for close contacts represented by a contact network, and another for social network.

Super class

ABM::R6Agent -> R6Population
Active bindings

size  The population size, an integer

Methods

Public methods:

• Population$new()
• Population$addAgent()
• Population$removeAgent()
• Population$addContact()
• Population$agent()
• Population$setState()
• Population$setStates()
• Population$clone()

Method new():

Usage:
Population$new(population = 0, initializer = NULL)

Arguments:

population can be either an external pointer pointing to a population object returned from newPopulation, or an integer specifying the population size, or a list.

initializer a function or NULL

Details: If population is a number (the population size), then initializer can be a function that take the index of an agent and return its initial state. If it is a list, the length is the population size, and each element corresponds to the initial state of an agent (with the same index). Add an agent

Method addAgent():

Usage:
Population$addAgent(agent)

Arguments:

agent  either an object of the R6 class Agent, or an external pointer returned from newAgent.

Details: The agent is scheduled in the population. If the population is already added to a simulation, the agent will report its state to the simulation. remove an agent

Returns: the population object itself (invisible) for chaining actions

Method removeAgent():

Usage:
Population$removeAgent(agent)

Arguments:

agent  either an object of the R6 class Agent, or an external pointer returned from newAgent.

Details: The agent is scheduled in the population. If the population is already added to a simulation, the agent will report its state to the simulation. Add a contact pattern
Population

Returns: the population object itself (invisible) for chaining actions

**Method** addContact():

*Usage:*

Population$addContact(contact)

*Arguments:*

contact an external pointer pointing to a Contact object, e.g., created from newRandomMix-
ing.

*Details:* If the contact has already been added, this call does nothing. return a specific agent by index

**Method** agent():

*Usage:*

Population$agent(i)

*Arguments:*

i the index of the agent (starting from 1)

*Returns:* an external pointer pointing to the agent set the state of a specific agent by index

**Method** setState():

*Usage:*

Population$setState(i, state)

*Arguments:*

i the index of the agent (starting from 1)

state a list holding the state to set

*Returns:* the population object itself (invisible) for chaining actions Set the states for the agents

**Method** setStates():

*Usage:*

Population$setStates(states)

*Arguments:*

states either a list holding the states (one for each agent), or a function

*Details:* If states is a function then it takes a single argument i, specifying the index of the agent (starting from 1), and returns a state.

*Returns:* the population object itself for chaining actions

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

Population$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.
schedule  Schedule (attach) an event to an agent

Description
Schedule (attach) an event to an agent

Arguments
agent  an external pointer returned by newAgent

event  an external pointer returned by newEvent

Details
If agent is an R6 object, then we should use either agent$schedule(event) or schedule(agent$get, event)
Similarly, if event is an R6 object, then we should use schedule(agent, event$get)

setDeathTime  set the time of death for an agent

Description
set the time of death for an agent

Arguments
agent  an external pointer returned by newAgent or getAgent()

time  the time of death, a numeric value

Details
If agent is an R6 object, then we should use either agent$leave() or leave(agent$get)
At the time of death, the agent is removed from the simulation. Calling it multiple times causes the agent to die at the earliest time.
setState

Set the state of the agent

Description

Set the state of the agent

Arguments

agent an external pointer returned by newAgent
state an R list giving the components of the state to be updated.

Details

In this framework, a state is a list, each named component is called a domain. This function only updates the values of the domain given in the "value" list, while leave the other components not in the "value" list unchanged.

If agent is an R6 object, then we should either use agent$schedule, or use schedule(agent$get, event)

setStates

Set the state for each agent in a population

Description

Set the state for each agent in a population

Arguments

population an external pointer to a population, for example, one returned by newPopulation()
states either a list holding the states (one for each agent), or a function

Details

If states is a function then it takes a single argument i, specifying the index of the agent (starting from 1), and returns a state.
Simulation

R6 class Create and represent a Simulation object

Description

The `Simulation` class inherits the `Population` class. So a simulation manages agents and their contact. Thus, the class also inherits the `Agent` class. So a simulation can have its own state, and events attached (scheduled) to it. In addition, it also manages all the transitions, using its `addTransition` method. As a last, it maintains loggers, which record (or count) the state changes, and report their values at specified times.

Super classes

`ABM::R6Agent` -> `ABM::R6Population` -> `R6Simulation`

Methods

Public methods:

- `Simulation$new()`
- `Simulation$run()`
- `Simulation$resume()`
- `Simulation$addLogger()`
- `Simulation$addTransition()`
- `Simulation$clone()`

Method `new()`:

Usage:

`Simulation$new(simulation = 0, initializer = NULL)`

Arguments:

`simulation` can be either an external pointer pointing to a population object returned from `newSimulation`, or an integer specifying the population size, or a list `initializer` a function or NULL.

Details: If simulation is a number (the population size), then `initializer` can be a function that take the index of an agent and return its initial state. If it is a list, the length is the population size, and each element corresponds to the initial state of an agent (with the same index). Run the simulation

Method `run()`:

Usage:

`Simulation$run(time)`

Arguments:

`time` the time points to return the logger values.
Details: the returned list can be coerced into a data.frame object which first column is time, and other columns are logger results, each row corresponds to a time point. The Simulation object first collect and log the states from all agents in the simulation, then set the current time to the time of the first event, then call the resume method to actually run it. Continue running the simulation

Returns: a list of numeric vectors, with time and values reported by all logger.

Method resume():

Usage: Simulation$resume(time)

Arguments:
  time  the time points to return the logger values.

Details: the returned list can be coerced into a data.frame object which first column is time, and other columns are logger results, each row corresponds to a time point. The Simulation object repetitively handle the events until the the last time point in "time" is reached. ASat each time point, the logger states are collected in put in a list to return. Add a logger to the simulation

Returns: a list of numeric vectors, with time and values reported by all logger.

Method addLogger():

Usage: Simulation$addLogger(logger)

Arguments:
  logger,  an external pointer returned by functions like newCounter or newStateLogger.

Details: without adding a logger, there will be no useful simulation results returned. Add a transition to the simulation

Returns: the simulation object itself (invisible)

Method addTransition():

Usage: Simulation$addTransition(
  rule,  waiting.time,  
  to_change_callback = NULL,  
  changed_callback = NULL
)

Arguments:
  rule  is a formula that gives the transition rule
  waiting.time  either an external pointer to a WaitingTime object such as one returned by new-ExpWaitingTime or newGammaWaitingTime, or a function (see the details section)
  to_change_callback  the R callback function to determine if the change should occur. See the details section.
  changed_callback  the R callback function after the change happened. See the details section.
Details: If `waiting.time` is a function then it should take exactly one argument `time`, which is a numeric value holding the current value, and return a single numeric value for the waiting time (i.e., should not add time).

Formula can be used to specify either a spontaneous transition change, or a transition caused by a contact.

A spontaneous transition has the form `from -> to`, where `from` and `to` are state specifications. It is either a variable name holding a state (R list) or the list itself. The list can also be specified by `state(...)` instead of `list(...)`. For a spontaneous transition, the callback functions take the following two arguments

1. `time`: the current time in the simulation
2. `agent`: the agent who initiate the contact, an external pointer

A transition caused by contact, the formula needs to specify the states of both the agent who initiate the contact and the contact agent. The two states are connected by a `+` sign, the one before the

- `•` sign is the initiator, and the one after the sign is the contact. The transition must be associated with a Contact object, using a `~` operator. The Contact object must be specified by a variable name that hold the external pointer to the object (created by e.g., the `newRandomMixing` function) For example, suppose `S=list("S"), I=list("I"), and m=newRandomMixing(sim)`, then a possible rule specifying an infectious agent contacting a susceptible agent causing it to become exposed can be specified by

```
I + S -> I + list("E") ~ m
```

For a transition caused by a contact, the callback functions take the third argument: 3. `contact`: the contact agent, an external pointer

Returns: the simulation object itself (invisible)

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
Simulation$clone(deep = FALSE)
```

Arguments:

- `deep`: Whether to make a deep clone.

---

State

<table>
<thead>
<tr>
<th>State</th>
<th>The state of an agent</th>
</tr>
</thead>
</table>

Description

In this framework, a state is a list, each named component is called a domain. The value of a domain can be any R value. The list can be at most one unnamed value, which corresponds to a domain with no name. This is useful if there is only one domain.

Details

A state can be matched to an R list (called a rule in this case). The state matches the rule if and only if each domain (names of the list) in rule has the same value as in state. The domains in domains of the state not listed in rule are not matched. In addition, to match to a rule, the domain values must be either a number or a character. This is useful for identifying state changes. See `newCounter()` and the `Simulation` class’ `addTransition` method for more details.
stateMatch

Description
Check if two states match

Arguments
- **state**: a list holding a state to check
- **rule**: a list holding the state to match against

Details
The state matches the rule if and only if each domain (names of the list) in rule has the same value as in state. The domains in domains of the state not listed in rule are not matched.

Value
a logical value

unschedule

Unschedule (detach) an event from an agent

Description
Unschedule (detach) an event from an agent

Arguments
- **agent**: an external pointer returned by `newAgent`
- **event**: an external pointer returned by `newEvent`

Details
If agent is an R6 object, then we should use either `agent$unschedule(event)` or `unschedule(agent$get, event)`
Similarly, if event is an R6 object, then we should use `unschedule(agent, event$get)`
Index

ABM (ABM-package), 3
ABM-package, 3
addAgent, 6
Agent, 3, 6, 26

clearEvents, 8
Contact, 9

Event, 3, 11

getAgent, 12
getAgent(), 6, 24
getId, 13
getSize, 13
getState, 14
getTime, 14
getWaitingTime, 15

leave, 15

matchState, 16

newAgent, 16
newAgent(), 6, 24
newConfigurationModel, 17
newCounter, 17
newCounter(), 3, 28
newEvent, 18
newExpWaitingTime, 19
newGammaWaitingTime, 19
newPopulation, 20
newPopulation(), 6, 12, 13, 25
newRandomMixing, 20
newRandomMixing(), 3
newStateLogger, 21
newStateLogger(), 3

Population, 3, 21, 26

schedule, 24
schedule(), 3

setDeathTime, 24
setDeathTime(), 16
setState, 25
setStates, 25
Simulation, 3, 18, 26, 26, 28
State, 3, 28
stateMatch, 29

unschedule, 29