TerraME Types and Functions

Pedro Ribeiro de Andrade
DSSA/CCST/INPE
pedro.andrade@inpe.br

Tiago Garcia de Senna Carneiro
TerraLab/UFOP
tiago@iceb.ufop.br

Version 0.8
April 26, 2013

This document presents a detailed description of each type and function of TerraME, ordered alphabetically by its types. TerraME adopts *American English* (e.g., neighbor instead of neighbour), with the following syntax convention:

- Names of types have the **upper** CamelCase style, starting with a capital letter, followed by other words starting with capitalized letters (e.g., Agent, Trajectory, CellularSpace).
- Functions and parameters names have the **lower** CamelCase style, with names starting with lowercase letters, followed by other words starting with capitalized letters (e.g., load, database, forEachCell, dbType).

There are two signatures for functions in TerraME. The first one uses the structure "function(v1, v2, ...)", where v1 is the 1st argument, v2 is the 2nd, and so forth. The arguments of a call to a function that has this signature must follow the specified order. It is possible to use fewer arguments than the function signature, with missing arguments taking their default values. Parameters of functions following this format are described as 1st, 2nd, etc. in this document. Every parameter that does not have a default value is compulsory. The second signature is "function{arg1 = v1, arg2 = v2, ...}", where v1 is the value of named argument arg1, v2 is the value of named argument arg2, and so on. These arguments can be used in any order, but the function call needs to use braces. Every type constructor of TerraME and some of its functions have this kind of signature. In this document, such arguments are described with their names.

Agent

Function	Description

Agent
`agent = Agent {`
`  id = "MyAgent",`
`  State {...},`
`  -- ...`
`  State {...}`
`}`
 	
`singleFooAgent = Agent {`
`  size = 10,`
`  name = "foo",`
`  execute = function(self)`
`    self.size = self.size + 1`
`    self:randomWalk()`
`  end,`
`  on_hello = function(self, m)`
`    self:message {`
`      receiver = m.sender,`
`      content = "hi"`
`    }`
`  end`
`}`
	Type that defines an Agent that is capable of performing actions and interact with other Agents and the spatial representation of the model. It can be described as a simple table or as a hybrid state machine that has a unique internal state. The initial State of the Agent is the first declared State. The Agent constructor gets a table containing the attributes and functions of the Agent.
	
	Attributes of Agent that can be used as *read-only* by the modeler:
	cells: a vector of Cells necessary to use forEachCell(agent). This value is the same of "agent.placement.cells".
	id: the unique identifier of the Agent within the Society (only when the Agent was not loaded from an external source),
	parent: the Society it belongs.
	placement: a Trajectory representing the default placement of the Agent. (Only when the Agent belongs to an Environment - itself directly or through a Society)
	socialnetworks: a set of SocialNetwork, with the connections of the Agent.
	type: a string containing "Agent".

add
`agent:add(state)`
`agent:add(trajectory)`
	Add a new Trajectory or State to the Agent.
	1st) A State or Trajectory.

addSocialNetwork	Add a new SocialNetwork to the Agent. **1st)** A SocialNetwork.

Function	Description

`agent:addSocialNetwork(network)`
`agent:addSocialNetwork(network, "friends")`
	2nd) Name of the relation.

Build `agent:build()`	Check if the state machine was correctly defined, verifying whether the targets of Jump rules match the ids of the States.
Die `agent:die()`	Remove the Agent from the Society it belongs. **1st)** A boolean value indicating whether the relations of the Agent should be removed. Default is true, but it only works with simple placements, where one Agent is connected to a single Cell in each placement. If more complex relations are used in the model, then the modeler should set this parameter as false and remove the relations by himself/herself.

Enter
`agent:enter(cell)`
`agent:enter(cell, "renting")`
	Put the Agent into a Cell, using the placement attributes of both.
	1st) A Cell.
	2nd) A string representing the index to be used. Default is "placement".

Execute
`agent:execute()`
`agent:execute(event)`
	The entry point for executing a given Agent. When the Agent is described as a state machine, execute is automatically defined by TerraME. It activates the Jump of the current State while it jumps from State to State. After that, it executes all the Flows of the current State. Usually, this function is called within an Event, thus the time of the Event can be got directly from the Timer. When the Agent is not defined as a composition of States, the modeler should use follow a signature to describe this function.
	1st) An Event.

getCell `cell = agent:getCell()`	Return the Cell where the Agent is located according to its placement. It assumes that each Agent belongs to at most one Cell.
getCells `cells = agent:getCells()`	Return the Cells pointed by the Agent according to its placement.
getID `id = agent:getID()`	Return the unique identifier of the Agent.
getLatency `latency = agent:getLatency()`	Return the time when the machine executed the transition to the current state. Before running, the latency is zero.
getSocialNetwork `net = agent:getSocialNetwork("friends")`	Return a SocialNetwork of the Agent given its name. **1st)** Name of the relation.
getStateName `name = agent:getStateName()`	Return a string with the current state name.
Init `agent:init()`	A user-defined function that is used to initialize an Agent when it enters in a given Society.

Leave
`agent:leave()`
`agent:leave(cell)`
`agent:leave(cell, "renting")`
	Remove the Agent from a given Cell.
	1st) A Cell. Default is the first (or the only) Cell of the placement.
	2st) A string representing the index to be used. Default is "placement".

message
`agent:message {`
`  receiver = agent2,`
`  delay = 2,`
`  content = "money",`
`  quantity = 20`
`}`
	Send a message to another Agent as a table. They can arrive exactly after they are sent (synchronous) or have some delay (asynchronous). In the later case, it is necessary to call function *synchronize* from the Society they belong to activate such messages.
	receiver: The Agent that will get the message.
	subject: A string describing the function that will be called in the receiver.

Function	Description
	Given a string x, the receiver will get the message in a function called on_x. Default is “message”. The function to receive the message must be implemented by the modeler. See Agent::on_* for more details. **delay:** An integer indicating the number of times synchronize needs to be called before activating this message. Default is zero (no delay, no synchronization required). Whenever a delayed message is received, it comes with the element delay = true. Other arguments are allowed to this function, as the message is a table. The receiver will get all the attributes sent plus a sender value.
move `agent:move(cell, "renting")`	Move the Agent to a new Cell. **1st)** The new Cell. **2nd)** A string representing the index to be used. Default is "placement".
on_* `agent:on_message()`	Signature of a function that can be implemented by the modelers when the Agents can receive messages from other ones. This function receives a message as argument, with the same content of the message sent plus the attribute sender, representing the Agent that has sent the message. In the case of non-delayed messages, the returning value of this function (executed by the receiver) is also returned as the result of message (executed by the sender).

randomWalk
`agent:randomWalk()`
`agent:randomWalk("moore")`
	Execute a random walk to a neighbor Cell.
	1nd) A string representing the index to be used. Default is "placement".

reproduce
`child = agent:reproduce()`
`child = agent:reproduce{age=0}`
`child = agent:reproduce{`
`  age = 0,`
`  house = agend.house:clone()`
`}`
	Create an Agent with the same behavior in the same Cell where the original Agent is (according to its placement). The new Agent is pushed into the same Society the original Agent belongs and placements created using the Society are instantiated with size zero if the only argument does not contain such placements. This function returns the new Agent.
	1nd) An optional table with attributes of the new Agent.

setTrajectoryStatus `agent:setTrajectoryStatus(true)`	Activate or not the trajectories defined for a given Agent. **1st)** Use or not the trajectories. As default, trajectories are turned off. If status is true, when executed, the Agent that contains States will automatically traverse all trajectories defined within it.

Automaton

Function	Description

Automaton
`automaton = Automaton {`
`  id = "MyAutomaton",`
`  State {...},`
`  -- ...`
`  State {...}`
`}`
	A hybrid state machine that needs to be located on a CellularSpace, and is replicated over each Cell of the space. It has independent States in each Cell. The initial State in each Cell is the first declared state.
	
	Attributes of Automaton that can be used as *read-only* by the modeler:
	parent: The Environment it belongs.
	type: A string containing “Automaton”.

Add
`automaton:add(state)`
`automaton:add(trajectory)`
	Add a new Trajectory or State to the Automaton.
	1st) A State.

Build `automaton:build()`	Check if the state machine was correctly defined, verifying whether the targets of Jump rules match the ids of the States.
execute `automaton:execute(event)`	Execute the state machine. First, it executes the Jump of the current State while it jumps from State to State. When the machine stops jumping, it executes all the Flows of the current State. Usually, this function is called within a Message, thus the time of the Event can be got from the Timer. **1st)** An Event.

Function	Description

getLatency
`latency =`
`automaton:getLatency()`
	Return the time when the machine executed the transition to the current state. Before running, the latency is zero.

setTrajectoryStatus `automaton:setTrajectoryStatus(true)`	Activate or not the trajectories defined for a given automata. **1st)** Use or not the trajectories. As default, trajectories are turned off. If status is true, when executed the automaton will automatically traverse all trajectories defined within it. Otherwise, the automaton will not run at all.

Cell

Function	Description

Cell
`cell = Cell {`
`  cover = "forest",`
`  soilWater = 0`
`}`
	A spatial location, with properties and nearness relations. It is a table that includes persistent and runtime attributes. Persistent attributes are loaded from and saved to databases, while runtime attributes exist only along the simulation.
	
	Attributes of Cell that can be used as *read-only* by the modeler:
	past: a copy of the attributes at the time of the last synchronization.
	parent: the CellularSpace the Cell belongs.
	type: a string containing "Cell".
	placement: a Group representing the default placement of the Cell. (Only when the CellularSpace of the Cell belongs to an Environment.)
	agents: a vector of Agents necessary to use forEachAgent(cell) (only when the CellularSpace of the Cell belongs to an Environment).

addNeighborhood
`cell:addNeighborhood(n)`
`cell:addNeighborhood(n, "east")`
	Add a new Neighborhood to a Cell.
	1st) A Neighborhood.
	2nd) Neighborhood's name (default "1"). It can be a string or a number, but it is always converted to string.

first `cell:first()`	Start a Neighborhood iterator, pointing to the first element of the Neighborhood list.
getAgent `agent = cell:getAgent()`	Return the Agent that belongs to a given Cell. It assumes that there is at most one Agent per Cell.
getAgents `agent = cell:getAgents()[1]`	Return the Agents that belong to a given Cell.
getCurrentNeighborhood `cell:getCurrentNeighborhood()`	Retrieve the Neighborhood currently pointed by the Neighborhood iterator, or nil otherwise.

getNeighborhood
`n = cell:getNeighborhood()`
`n =`
`cell:getNeighborhood("moore")`
	Return one of the Neighborhoods of a Cell.
	1st) A string with the Neighborhood's name to be retrieved (default is "1").

getPast `cellPast = cell:getPast()`	Return the values of the Cell in the last time synchronize() was called.

isFirst
`if cell:isFirst() then`
`    ...`
`End`
	Return whether the Neighborhood iterator is pointing to the first Neighborhood of the list.

isLast	Return whether the Neighborhood iterator has already passed by the last

Function	Description
`l = cell:isLast()`	Neighborhood of the list, or whether the iterator does not exist.
Last `last = cell:last()`	Set the Neighborhood iterator to the last element of the Neighborhood list.
Next `cell:next()`	Update the Neighborhood iterator to the next Neighborhood of the list.
Notify `cell:notify()`	Notify every observer connected to the Cell. **1st)** The time to be used in the observer. Most of the strategies available ignore this value; therefore it can be left empty. See the Observer documentation for details.
size `size = cell:size()`	Return the number of Neighborhoods of a Cell.
synchronize `cell:synchronize()`	TerraME can keep two copies of the attributes of a Cell in memory: one stores the past values and the other stores the current (present) values. Synchronize copies the current values to a table named *past*, within the Cell.

CellularSpace

Function	Description

CellularSpace
`cs = CellularSpace {`
`  database = "amazonia",`
`  theme = "cells",`
`  user = "root"`
`}`
 	
`cs2 = CellularSpace {`
`  database = "d:\\db.mdb",`
`  layer = "cells_10",`
`  theme = "cells_10",`
`  select = "height3 as h",`
`  where = "height3 > 2"`
`}`
 	
`cs3 = CellularSpace {`
`  database = "d:\\file.shp",`
`}`
 	
`cs4 = CellularSpace {`
`  xdim = 20,`
`  ydim = 20`
`}`
	A multivalued set of Cells, which can be retrieved from TerraLib databases or created directly within TerraME (rectangular CellularSpaces). These two ways of creating CellularSpaces have different mandatory arguments: **database** and **theme** for reading from a DBMS, and **xdim** and **ydim** for CellularSpaces only in memory. Cellular spaces stored in databases need to be loaded to TerraME before using it. Calling forEachCell() traverses CellularSpaces.
	
	database: Name of the database. It can also describe the location of a shapefile. In this case, the other arguments will be ignored.
	theme: Name of the theme to be loaded.
	dbType: Name of DBMS. The default value depends on the **database** name. If it has a ".mdb" extension, the default value is "ado", otherwise it is "mysql"). TerraME always converts this string to lower case.
	host: Host where the database is stored (default is "localhost").
	port: Port number of the connection.
	user: Username (default is "").
	password: The password (default is "").
	layer: Name of the layer the theme was created from. It must be used to solve a conflict when there are two themes with the same name (default is "").
	load: a boolean value indicating whether the CellularSpace will be loaded automatically (true, default value) or the user by herself will call load (false).
	select: A table containing the names of the attributes to be retrieved (default is all attributes). When retrieving a single attribute, you can use select = "attribute" instead of select = {"attribute"}. It is possible to rename the attribute name using "as", for example, select= {"lc as landcover"} reads lc from the database but replaces the name to landcover in the Cells. Attributes that contain "." in their names (such as results of table joins) will be read with "_" replacing "." in order to follow Lua syntax to manipulate data.
	where: A SQL restriction on the properties of the Cells (default is "", applying no restriction. Only the Cells that reflect the established criteria will be loaded). The where argument ignores the "as" flexibility of select.
	xdim: Number of columns, in the case of creating a CellularSpace without needing to load from a database.

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td></td><td><b>ydim:</b> Number of lines, in the case of creating a CellularSpace without needing to load from a database. Default is equal to xdim.

Attributes of CellularSpace that can be used as <i>read-only</i> by the modeler:
<b>cells:</b> A vector of Cells pointed by the CellularSpace.
<b>cObj_:</b> A pointer to a C++ object.
<b>parent:</b> The Environment it belongs.
<b>type:</b> A string containing “CellularSpace”.</td></tr>
<tr><td><b>add</b><pre>cs:add(cell)</pre></td><td>Add a new Cell to the CellularSpace. It will be the last Cell of the CellularSpace.
<b>1st)</b> A Cell.</td></tr>
<tr><td><b>createNeighborhood</b><pre>-- moore
cs:createNeighborhood()

cs:createNeighborhood {
   name = "moore"
}

cs:createNeighborhood {
  strategy = "vonneumann",
  self = false
}

cs:createNeighborhood {
  strategy = "mxn",
  M = 4,
  N = 4
}

-- c2 is nested in cs1
cs1:createNeighborhood {
  strategy = "mxn",
  target = cs2, -- other cs
  M = 3,
  N = 2,
  name = "spatialCoupling"
}</pre></td><td>Create a Neighborhood for each Cell of the CellularSpace. It gets a table as argument, with the following attributes:
<b>strategy:</b> A string with the strategy to be used for creating the Neighborhood. See the table below.

<table>
<tr><th>Strategy</th><th>Description</th><th>Parameters (<b>bold</b> are compulsory)</th></tr>
<tr><td>"3x3"</td><td>A 3x3 (Couclelis) Neighborhood.</td><td>name, filter, weight</td></tr>
<tr><td>"coord"</td><td>A bidirected relation between two CellularSpaces connecting Cells with the same (x, y) coordinates.</td><td>name, <b>target</b></td></tr>
<tr><td>"function"</td><td>A Neighborhood based on a function where any other Cell can be a neighbor.</td><td>name, filter, weight</td></tr>
<tr><td>"moore" (default)</td><td>A Moore (queen) Neighborhood.</td><td>name, self, wrap</td></tr>
<tr><td>"mxn"</td><td>A MxN (columns x rows) Neighborhood within the CellularSpace or between two CellularSpaces if target is used.</td><td>name, <b>M</b>, <b>N</b>, <b>filter</b>, <b>weight,</b> target</td></tr>
<tr><td>"vonneumann"</td><td>A von Neumann (rook) Neighborhood</td><td>name, self</td></tr>
</table>

<b>filter:</b> A function(Cell, Cell)→bool, where the first argument is the Cell itself and the other represent a possible neighbor. It returns true when the neighbor will be included in the relation. In the case of two CellularSpaces, this function is called twice for each pair of Cells, first filter(c1, c2) and then filter(c2, c1), where c1 belongs to cs1 and c2 belongs to cs2.
<b>M:</b> Number of columns. If M is even then it will be increased by one to keep the Cell in the center of the Neighborhood.
<b>N:</b> Number of rows. If N is even then it will be increased by one to keep the Cell in the center of the Neighborhood.
<b>name:</b> A string with the name of the Neighborhood to be created. Default is "1".
<b>self:</b> Add the Cell as neighbor of itself? Default is false. Note that the functions that do not require this argument always depend on a filter function, which will define whether the Cell can be neighbor of itself.
<b>target:</b> Another CellularSpace whose Cells will be used to create neighborhoods.
<b>weight:</b> A function(Cell, Cell)→number, where the first argument is the Cell itself and the other represent its neighbor. It calculates the weight of the relation. The weight will be computed only if filter returns true.
<b>wrap:</b> Whether Cells in the borders will be connected to the Cells in the opposite border. Default is false.</td></tr>
<tr><td><b>getCell</b><pre>cs:getCell(coord)</pre></td><td>Retrieve a Cell from the CellularSpace, given its index.
<b>1st)</b> A Coord.</td></tr>
<tr><td><b>getCells</b></td><td>Return a vector containing all Cells of the CellularSpace.</td></tr>
</table>

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td>cells = cs:getCells()
cell = cs:getCells()[1]</td><td></td></tr>
<tr><td><b>load</b>

cs:load()</td><td>Load the CellularSpace from the database. TerraME automatically executes this function when the CellularSpace is created, but one can execute this to load the attributes again, erasing each other attribute and relations created by the modeler.</td></tr>
<tr><td><b>loadNeighborhood</b>

cs:loadNeighborhood{
  source = "n.gpm"
}

cs:loadNeighborhood{
  source = "mtab",
  name = "mtab"
}</td><td>Load a Neighborhood stored in an external source. Each Cell receives its own set of neighbors.
<b>name</b>: A string with the location of the Neighborhood to be loaded. See below.

<table>
<tr><th>Source</th><th>Description</th></tr>
<tr><td>"*.gal"</td><td>Load a Neighborhood from contiguity relationships described as a GAL file.</td></tr>
<tr><td>"*.gwt"</td><td>Load a Neighborhood from a GWT (generalized weights) file.</td></tr>
<tr><td>"*.gpm"</td><td>Load a Neighborhood from a GPM (generalized proximity matrix) file.</td></tr>
<tr><td>Any other</td><td>Load a Neighborhood from table stored in the same database of the CellularSpace.</td></tr>
</table>

<b>source</b>: A string with the name of the Neighborhood to be loaded within TerraME. Default is "1".</td></tr>
<tr><td><b>notify</b>

cs:notify()
cs:notify(event:getTime())</td><td>Notify every observer connected to the CellularSpace.
<b>1st)</b> The time to be used in the observer. Most of the strategies available ignore this value; therefore it can be left empty. See the Observer documentation for details.</td></tr>
<tr><td><b>sample</b>

cell = cs:sample()</td><td>Return a random Cell from the CellularSpace.</td></tr>
<tr><td><b>save</b>

cs:save(20,"table")

cs:save(100, "ntab", "attr")</td><td>Save the attributes of a CellularSpace into the same database it was retrieved.
<b>1st)</b> A temporal value to be stored in the database, which can be different from the simulation time.
<b>2nd)</b> Name of the table to store the attributes of the Cells.
<b>3rd)</b> A vector with the names of the attributes to be saved (default is all of them). When saving a single attribute, you can use attrNames = "attribute" instead of attrNames = {"attribute"}.</td></tr>
<tr><td><b>size</b>

print(cs:size())</td><td>Retrieve the number of elements in the CellularSpace.</td></tr>
<tr><td><b>split</b>

ts = cs:split("cover")
print(ts.forest:size())
print(ts.pasture:size())

ts2 = cs:split(function(cell)
  if cell.forest > 0.5 then
    return "gt"
  else
    return "lt"
  end
end)
print(ts.gt:size())</td><td>Split the CellularSpace into a set of Trajectories according to a classification strategy. The generated Trajectories have empty intersection and union equals to the whole CellularSpace (unless function below returns nil for some Cell). It works according to the type of its only and compulsory argument, that can be:

<table>
<tr><th>Type of argument</th><th>Description</th></tr>
<tr><td>string</td><td>The argument must represent the name of one attribute of the Cells of the CellularSpace. Split then creates one Trajectory for each possible value of the attribute using the value as index and fills them with the Cells that have the respective attribute value.</td></tr>
<tr><td>function</td><td>The argument is a function that receives a Cell as argument and returns a value with the index that contains the Cell. Trajectories are then indexed according to the returning value.</td></tr>
</table></td></tr>
<tr><td><b>synchronize</b>

cs:synchronize()
cs:synchronize("landuse")
cs:synchronize{"water","use"}</td><td>Synchronize the CellularSpace, calling the function synchronize() for each of its Cells.
<b>1st)</b> A string or a vector of strings with the attributes to be synchronized. If empty, TerraME synchronizes every attribute read from the database but the (x, y) coordinates and the attributes created along the simulation.</td></tr>
</table>

Coord

Function	Description

Coord
`coord = Coord()`
`coord2 = Coord{x = 2, y = 3}`
`print(coord2.x) -- nil`
	Type that stores a pair (x, y). Once created, it is only possible to retrieve (x, y) by using get().
	x: A position on the horizontal axis of a two-dimensional Cartesian coordinate system.
	y: A position on the vertical axis of a two-dimensional Cartesian coordinate system.

get
`print(coord:get().x)`
`print(coord:get().y)`
	Return a table with (x, y) as values.

set
`coord:set{x = 3, y = 2}`
`coord:set{x = 4}`
	Change the pair (x, y), or only one of its original values.
	x: A position on the horizontal axis. Default is not changing.
	y: A position on the vertical axis. Default is not changing.

Environment

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td><b>Environment</b>

environment = Environment {
  cs1 = CellularSpace{...},
  ag1 = Agent{...},
  aut2 = Automaton{...},
  t1 = Timer{...},
  env1 = Environment{...}
}</td><td>A container that encapsulates space, time, behavior, and other environments. Objects can be added directly when the Environment is declared or after it has been instantiated. It can control the simulation engine, synchronizing all the Timers within it. Calling forEachElement() traverses each object of Environments.</td></tr>
<tr><td><b>add</b>
environment:add(agent)
environment:add(cellularSpace)</td><td>Add an object to the Environment. The functions below are more efficient because they do not have to find out the type of the parameter.
<b>1st)</b> An Agent, Automaton, CellularSpace, Timer, or Environment.</td></tr>
<tr><td><b>createPlacement</b>
environment:createPlacement{
  strategy = "uniform"
}</td><td>Create relations between behavioural entities (Agents) and spatial entities (Cells). The Environment must have only one CellularSpace. It is possible to have more than one behavioural entity in the Environment.
<b>strategy:</b> A string containing the strategy to be used for creating a placement between Agents and Cells. See the options below.

<table>
<tr><th>Strategy</th><th>Description</th><th>Parameters</th></tr>
<tr><td>"random" (default)</td><td>Create placements between Agents and Cells randomly, putting each Agent in a Cell randomly chosen.</td><td>name, max</td></tr>
<tr><td>"uniform"</td><td>Create placements uniformly. The first Agents enter in the first Cells. The last Cells will contain fewer Agents if the number of Agents is not proportional to the number of Cells. For example, placing a Society with four Agents in a CellularSpace of three Cells will put two Agents in the first Cell and one in each other Cell.</td><td>name</td></tr>
<tr><td>"void"</td><td>Create only the pointers for each object in each side, preparing the objects to be manipulated by the modeler.</td><td>name</td></tr>
</table>

<b>name:</b> Name of the relation in TerraME objects. Default is “placement”, which means that the modeler can use enter(), move(), and leave() directly. If the name is different from the default value, the modeler will have to use the last argument of these functions to indicate which relation they are changing or perform changes on these relations manually.</td></tr>
</table>

Function	Description
	max: The maximum number of Agents that can enter in the same Cell. Default is having no limit. Using max is computationally efficient only when the number of Agents is considerably lower than the number of Cells times max. Otherwise, it is better to consider using the uniform strategy.
execute `environment:execute(1000)`	Execute the Environment until a given time. It activates the Timers it contains, the Timers of the Environments it contains, and so on. **1st)** Time to stop the simulation. Timers stop when there is no Event scheduled to a time less or equal to the final time.

loadNeighborhood
`environment:loadNeighborhood{`
`  source = "file.gpm",`
`  name = "newNeigh"`
`}`
	Load a Neighborhood between two different CellularSpaces.
	source: Name of the file to be loaded.
	name: Name of the relation to be created. Default is "1".
	bidirect: For each relation from Cell a to Cell b, create also a relation from b to a. Default is false.

Event

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td><b>Event</b>

<code>event = Event {</code>
<code> time = 1985,</code>
<code> period = 2,</code>
<code> priority = -1,</code>
<code> action = function(event)</code>
<code>   print(event:getTime())</code>
<code> end</code>
<code>}</code>

<code>event2 = Event {</code>
<code> time = 2000,</code>
<code> action = my_society</code>
<code>}</code></td><td>An Event represents a time instant when the simulation engine must execute some computation.

<b>time:</b> The first instant of time when the Event will occur (default is one).
<b>period:</b> The periodicity of the Event (default is one).
<b>priority:</b> Define the priority of the Event over other Events. The default priority is zero. Smaller values have higher priority.
<b>action:</b> Function from where, in general, the simulation engine services are invoked. This function has one single argument, the Event itself. If the action returns false, the Event is removed from the Timer and will not be executed again. Action can also be a TerraME object. In this case, each type has its own set of functions that will be activated by the Event. See below how the objects are activated. Arrows indicate the execution order.
<table>
<tr><th>Object</th><th>Function(s) activated</th></tr>
<tr><td>Agent/Automaton</td><td>execute → notify</td></tr>
<tr><td>CellularSpace/Cell</td><td>synchronize → notify</td></tr>
<tr><td>function</td><td>function</td></tr>
<tr><td>Society</td><td>execute → synchronize → notify</td></tr>
<tr><td>Timer</td><td>notify</td></tr>
<tr><td>Trajectory/Group</td><td>rebuild → notify</td></tr>
</table></td></tr>
<tr><td><b>config</b>
<code>event:config(1)</code>
<code>event:config(1, 0.05)</code>
<code>event:config(1, 0.05, -1)</code></td><td>Change the attributes of an Event that belongs to a Timer in such a way that it will be scheduled again according to its new attributes.
<b>1st)</b> The first instant of time when the Event will occur (default is the current time of the Timer it will belong).
<b>2nd)</b> The periodicity of the Event (default is 1).
<b>3rd)</b> Define the priority of the Event over other Events. The default priority is 0 (zero). Smaller values have higher priority.</td></tr>
<tr><td><b>getPeriod</b>
<code>period = event:getPeriod()</code></td><td>Return the period of a given Event. This function can be used only along the simulation, when the Event is activated and comes as a parameter to a message.</td></tr>
<tr><td><b>getPriority</b>
<code>priority = event:getPriority()</code></td><td>Return the priority of a given Event. This function has restrictions of use as above.</td></tr>
<tr><td><b>getTime</b>
<code>time = event:getTime()</code></td><td>Return the current simulation time. This function has restrictions of use as above.</td></tr>
</table>

Flow

Function	Description

Flow
`Flow { function(ev, agent, cell)`
`  agent.value = agent.value + 2`
`end}`
	A Flow describes the behavior of an automaton or Agent in a given State. It is a user-defined function that receives three parameters: the Event that activated the Flow, the automaton/Agent that owns the Flow, and the Cell over which the Flow will be evaluated.

Group (Inherits Society)

Function	Description

Group
`group = Group {`
`  target = society,`
`  select = function(agent)`
`    return agent.money > 90`
`  end,`
`  greater = function(a, b)`
`    return a.money > b.money`
`  end`
`}`
 	
`groupBySize = Group {`
`  target = society,`
`  greater = function(a1, a2)`
`    return a1.size > a2.size`
`  end`
`}`
	Type that defines an ordered selection over a Society. It inherits Society; therefore it is possible to use all functions of such type within a Group. For instance, calling forEachAgent() also traverses Groups.
	
	target: The Society over which the Group will take place.
	select: A function (Agent)→boolean to filter the Society, adding to the Group only those Agents whose returning value is true. If this argument is missing, all Agents will be included in the Group.
	greater: A function (Agent, Agent)→boolean to sort the generated subset of Agents. It returns true if the first one has priority over the second one. If this argument is missing, no sorting function will be applied. See greaterByAttribute() for a pre-defined options.
	build: A boolean value indicating whether the Group will be computed or not when created. Default is true.
	
	Attributes of Trajectory that can be used as *read-only* by the modeler:
	agents: A vector of Agents pointed by the Group.
	parent: The Society where the Group takes place.
	select: The last function used to filter the Group.
	greater: The last function used to sort the Group.

add `group:add(agent)`	Add an Agent to the Group. **1st)** The Agent been added to the Group.
clone `group:clone()`	Return a copy of the Group, with the same parent, select, greater and Agents.

filter
`group:filter(function(agent)`
`    return agent.age > 18`
`end)`
	Apply a filter over the original Society.
	1st) A function such as the argument select.

randomize `group:build()`	Randomizes the Agents, changing the traversing order.
rebuild `group:build()`	Rebuild the Group from the original data using the last filter and sort functions.

sort
`group:sort(function(ag1, ag2)`
`   return ag1.money > ag2.money`
`end)`
	Sort the current Society subset.
	1st) An ordering function with the same signature of argument greater.

Jump

Function

Description

Jump

```
Jump { function(ev, agent, c)
    return c.water > c.capInf
  end,
  target = "wet"
}
```

Control a discrete transition between States. If the method in the first argument returns true, the target becomes the new active State.

1st) a function that returns a boolean value and takes as arguments an Event, an Agent/Automaton, and a Cell, respectively.

target: a string with another State id.

Legend

Function

Legend

```
coverLeg = Legend {
  grouping = "uniquevalue",
  colorBar = {
    {value = 0, color = "white"},
    {value = 1, color = "red"},
    {value = 2, color = "green"}
  }
}

deforLeg = Legend {
  grouping = "equalsteps",
  slices = 10,
  colorBar = {
    {value = 0, color = "green"},
    {value = 1, color = "red"}
  }
}
```

Type that defines a legend to be used in an observer. It is used only when the observer is of type map. The configuration of a legend can be changed visually within the graphical interface along the simulation.

grouping: A string to define the strategy to slice and color the data. See below.

Grouping	Description	Parameters
"equalsteps"	Draw objects according to their attributes, which are divided into a set of slices with the same range. Each slice is associated to a given color. Equalsteps require only two colors in the colorBar.	colorBar, slices, maximum, minimum, precision, type, width
"quantil"	Classify a set of objects according to a given attribute. Classes, or slices, have approximately the same size and similar atributes. Slices are ordered from the lowest values to the higher ones, associating colors to this order.	colorBar, slices, maximum, minimum, precision, type, width
"stdeviation"	Define slices to group objects according to the distribution of a given attribute. Objects with similar positive or negative distances to the average will belong to the same slice.	stdDeviation, colorBar, stdColorBar, precision, type, width
"uniquevalue"	Draw objects with each attribute value corresponding to a given color. String attributes can only belong to uniquevalue groupings.	colorBar, type, width

type: The type of the attribute to be observed. It has to be one of "bool", "number", "string", and "datetime" (an ordered string).

slices: The number of colors to be used for plotting.

precision: The number of decimal digits for slicing.

stdDeviation: When the grouping mode is stddeviation, it has to be one of "full", "half" "quarter", or "none".

maximum: The maximum value of the attribute (used only for numbers).

minimum: The minimum value of the attribute (used only for numbers).

width: The width of the line to be drawn. Used for drawing Neighborhoods (default is 10).

colorBar: A table where each position is a table with a 'color' and the respective 'value'. Colors can be described as string ("red", "green", "blue", "white", "black", "yellow", "brown", "cyan", "gray", "magenta"), or as tables with three values

Function	
	representing their RGB compositions. **stdColorBar:** A table just as colorBar. It is needed only when standard deviation is the chosen strategy.

Neighborhood

Function	Description
Neighborhood `n = Neighborhood()`	Each Cell has one or more Neighborhoods to represent proximity relations. A Neighborhood is a set of pairs (cell, weight), where cell is a neighbor Cell and weight is a number storing the relation's strength.

addCell
`n:addCell(coord, cs)`
`n:addCell(coord, cs, 0.001)`
	Add a new Cell to the Neighborhood.
	1st) A Coord.
	2nd) The CellularSpace that contains the Cell to be added.
	3rd) A number representing the weight of the connection (default 0).

Clear `n:clear()`	Remove all Cells from the Neighborhood. In practice, it has almost the same behavior as calling Neighborhood() again.
eraseCell `n:eraseCell(coord)`	Remove a Cell from the Neighborhood. **1st)** A Coord.
First `n:first()`	Start a neighbor iterator, pointing to the first Cell in the neighbors list.
getCellNeighbor `n:getCellNeighbor(coord)`	Return a neighbor, given its coords. **1st)** A Coord.
getCellWeight `n:getCellWeight(coord)`	Return the weight of the connection to a given neighbor Cell. **1st)** A Coord.
getCoord `coord = n:getCoord()`	Return the coordinates of the neighbor pointed by the current iterator.

getID
`c1:addNeighborhood(n, "n")`
`c2:addNeighborhood(n, "n2")`
`id = n:getID() -- "n2"`
	Return the name of the Neighborhood in the last Cell it was added.

getNeighbor `neigh = n:getNeighbor()`	Return the neighbor pointed by the current iterator.
getWeight `weight = n:getWeight()`	Return the weight of the connection to a neighbor pointed by the current iterator.
isEmpty `bool = n:isEmpty()`	Return whether the Neighborhood does not contain any Cell.
isFirst `bool = n:isFirst()`	Return whether the neighbor iterator is pointing to the first Cell of the list.

isLast
`if n:isLast() then`
`  print("is last")`
`end`
	Return whether the neighbor iterator has already passed by the last Cell of the list, or whether the iterator does not exist.

last `n:last()`	Set the neighbor iterator to the last element in the neighborhood. .
next `n:next()`	Change the neighbor iterator to the next Cell of the neighborhood.
sample	Return a single sample from the Neighborhood.

Function	Description
`cell = n:sample()`	
setCellWeight `n:setCellWeight(coord, 0.001)`	Update the weight of a connection to a neighbor. **1st)** A Coord. **2nd)** A number pointing out the new weight.
setWeight `n:setWeight(0.001)`	Update the weight of the connection to a neighbor pointed by the current iterator. **1st)** A number representing the new weight.
size `print(n:size())`	Retrieve the number of neighbors the Neighborhood has.

Observer

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr>
<td><b>Observer</b>
<pre>
observer1 = Observer {
  subject = cs,
  attributes = "water",
  legends = {soilWaterLeg}
}

Observer {
  subject = trajectory,
  observer = observer
}

observer3 = Observer {
  subject = cell,
  type = "chart",
  attributes = {"water"}
}
</pre></td>
<td>
<p>Observer is the way to collect data from the objects of a model in order to save, to graphically plot them, or to send them to another computer. Observers can be created from any TerraME object that has a built-in function called notify(). This function needs to be called to update its observers because they are passive objects. Observers do not need to be put into an object to exist, as in the second example on the left side.</p>
<p><b>type</b>: A string to define the way to observe a given object. See the table below.</p>
<table>
<tr><th>Type</th><th>Description</th><th>Parameters (<b>bold</b> are compulsory)</th></tr>
<tr><td>"chart"</td><td>Create a line chart showing the variation of an attribute (y axis) of an object. X axis can be another attribute (described as a second argument for parameter attributes) or a temporal value coming from the argument of notify().</td><td><b>subject, attributes</b>, xaxis, xLabel, yLabel, title, curveLabels</td></tr>
<tr><td>"image"</td><td>Create a map with the spatial distribution of a given Agent, CellularSpace, Society or Trajectory, saving it in a png file for each notify(). It works in the same way of the observer map.</td><td>subject, attribute, file, legend</td></tr>
<tr><td>"logfile"</td><td>Save attributes of an object into a csv text file, with one row for each notify().</td><td><b>subject</b>, file, <b>attributes</b>, separator, mode</td></tr>
<tr><td>"map"</td><td>Create a map with the spatial distribution of a given CellularSpace, Trajectory, Agent, or Society. It draws each element into the screen, according to one or two attributes (two is allowed only for CellularSpace) colored from one or two Legends, respectively. The second attribute and Legend are used as background.</td><td><b>subject</b>, <b>attribute</b>, <b>observer</b> (unless when the subject is a CellularSpace), legend</td></tr>
<tr><td>"neighborhood"</td><td>Draw the Neighborhood of a Cell, or the Neighborhoods of each Cell within a Trajectory, CellularSpace, or Environment. They are drawn as lines, according to a neighType.</td><td><b>subject, observer</b>, neighIndex, neighType</td></tr>
<tr><td>"scheduler"</td><td>Create a display with the current time and Event queue of a given Timer.</td><td>subject</td></tr>
<tr><td>"statemachine"</td><td>Draw the state machine of an Automaton in a Cell or an Agent. As default, states are drawn</td><td><b>subject, location</b> (only when the</td></tr>
</table>
</td>
</tr>
</table>

Function	Description

	as gray circles with a green circle to represent the current state. Unique value Legends can be used to map state names to colors, putting the current state in evidence with bold font.	subject is an Automaton), legend
"table"	Display a table with the current attributes of an object. Each notify() overwrites the previous values.	**subject**, attribute
"textscreen"	Create a display in a tabular format with the current attributes of an object. It will have one row for each notify().	**subject**, attribute
"udpsender"	Send observed attributes of an object through a UDP port of a given IP.	**subject**, attribute, host, port

attributes: A vector of strings with the name of the attributes to be observed. When empty, the observer will use every available attribute of the object that is not a table or an external pointer.

file: Name of the file to be saved. In the case of images, it represent the fixed part of the file name that will be concatenated with a timestamp and ".png". In the case of logfiles, it must be a file ending with ".csv". Default value is "result_" for image files and result_.csv for logfiles.

host: A string or a vector of strings with host names for udpsenders.

legends: A Legend or a vector of Legends to paint objects according to their properties.

location: A Cell representing a location to observe an Automaton.

mode: The open mode for a logfile observer, with "w" for writing a new file, "w+" for overwriting an existing file (default), or "a" to append an existing file.

neighIndex: A string or a vector of strings representing the neighborhood indexes to be drawn by a neighborhood observer. Default is "1".

neighType: One of three strings, "*basic*" (default), "*color*", or "*width*", for neighborhood observers. *Basic* type draws neighborhoods as lines with the same color and width. *Color* draws them using colors according to their weights. *Width* draws them with widths according to their weights. All them use the attribute *width* of Legends. The first two use it as width for all lines, while the last one interpolates the weights of the relations to draw widths between one pixel and the Legend width.

observer: An Observer that will be used as background for drawing properties of observed objects that canxnot be drawn alone.

port: A string or a vector of strings with ports for the respective host names to be used by udpsenders.

separator: The attribute separator character (i.e., ";"). Used only for logfiles.

subject: The TerraME object that will be observed.

title: An overall title to the observer.

xaxis: A string representing the attribute to be used as x axis in a chart observer. When nil, time will be used as axis.

xLabel: Name of the x-axis. When xaxis is not nil, default is value xaxis, otherwise it is "time".

yLabel: Name of the y-axis. Default is attribute[1] when table.getn(attribute) == 1. Otherwise is "".

curveLabels: Vector of the same size of attributes that indicates the labels for each line of a chart. Default is the name of the attributes.

Default values of observer types depend on the parameters. See table below for a description on how it works.

Parameters, from higher to lower priority	Default type
file == "*.csv"	logfile
file ~= nil	image

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><td>host ~= nil <i>or</i> port ~= nil</td><td>udpsender</td></tr>
<tr><td>neighIndex ~= nil <i>or</i> neighType ~= nil</td><td>neighborhood</td></tr>
<tr><td>type(subject) == "Timer"</td><td>scheduler</td></tr>
<tr><td>type(subject) == "Event"</td><td>table</td></tr>
<tr><td>type(subject) == "CellularSpace"</td><td>map</td></tr>
<tr><td>type(subject) == "Trajectory"</td><td>map</td></tr>
<tr><td>type(observer) == "Observer" <i>and</i> type(subject) == "Cell"</td><td>neighborhood</td></tr>
<tr><td>type(subject) == "Cell"</td><td>table</td></tr>
<tr><td>type(subject) == "Automaton"</td><td>map</td></tr>
<tr><td>type(subject) == "Agent"</td><td>statemachine</td></tr>
<tr><td>type(subject) == "Society"</td><td>map</td></tr>
<tr><td>type(subject) == "Group"</td><td>map</td></tr>
</table>
</td></tr>
</table>

SocialNetwork

Function	Description
SocialNetwork `sn = SocialNetwork()`	Each Agent has one or more social networks to represent its relations. A SocialNetwork is a set of pairs (connection, weight), where connection is an Agent and weight is a number storing the relation's strength. Calling forEachConnection() traverses SocialNetworks.

add
`sn:add(agent)`
`sn:add(agent, 0.5)`
	Add a new connection to the SocialNetwork.
	1st) An Agent.
	2nd) A number representing the weight of the connection (default nil **-** no weight).

clear `sn:clear()`	Remove all Agents from the SocialNetwork. In practice, it has almost the same behavior as calling SocialNetwork() again.
getConnection `sn = getConnection("1")`	Return a connection given its id. **1st)** The unique identifier of an Agent.
getID `print(sn:getID())`	Return the ID used to index the SocialNetwork into the Agent.
getWeight `print(sn:getWeight(agent))`	Return the weight of a given connection. **1st)** An Agent.

isEmpty
`if sn:isEmpty() then`
`  print("empty")`
`end`
	Return whether the SocialNetwork does not contain any connection.

isConnection
`if sn:isConnection(agent) then`
`  print("not connected")`
`end`
	Return whether a given Agent is a connection.
	1st) An Agent.

remove `sn:remove(agent)`	Remove an Agent from the SocialNetwork. **1st)** An Agent.
sample `agent = sn:sample()`	Return a single sample from the SocialNetwork.
setWeight `sn:setWeight(agent, 0.001)`	Update the weight of a connection. **1st)** An Agent. **2nd)** A number pointing out the new weight.
size `print(sn:size())`	Retrieve the number of connections the SocialNetwork has.

Society

Function	Description

Society
`my_instance = Agent {`
`  -- ...`
`}`
 	
`s = Society {`
`  instance = my_instance,`
`  quantity = 20`
`}`
 	
`s = Society {`
`  instance = my_instance,`
`  database = "c:\\datab.mdb",`
`  layer = "farmers"`
`}`
 	
`mydata = readCSV(...)`
 	
`s = Society {`
`  instance = my_instance,`
`  data = mydata`
`}`
	Type to create and manipulate a set of Agents. Each Agent within a Society has a unique id, which is initialized while creating the Society. There are three ways to create a Society: the first one uses a “quantity” to indicate the number of copies of the instance to be created. The second uses “data” with positions representing basic attributes of each Agent to be created. The last one uses a “layer” from a database to load attributes to the Agents. Calling forEachAgent() traverses Societies.
	
	database: Name of the database.
	dbType: Name of DBMS. The default value depends on the **database** name. If it has a “.mdb” extension, the default value is "ado", otherwise it is "mysql"). TerraME always converts this string to lower case.
	file: A filename (.csv) where the Society is stored.
	host: Host where the database is stored (default is "localhost").
	id: The unique identifier attribute used when reading the Society from a file.
	instance: A table with the description of the attributes and functions of an Agent. Some functions that may have internal TerraME use are:
	execute(self): a function with the behavior of the Agent when activated.
	init(self): a function called at the end of the instantiation process.
	on_*(self, message): a function called when the Agent receives a message. See Agent:message() for more details.
	layer: Name of the layer the theme was created from. It must be used to solve a conflict when there are two themes with the same name (default is "").
	password: The password (default is "").
	port: Port number of the connection.
	quantity: Number of Agents to be created. It is used when the Society will not be loaded from a file or database.
	select: A table containing the names of the attributes to be retrieved (default is all attributes). When retrieving a single attribute, you can use select = "attribute" instead of select = {"attribute"}. It is possible to rename the attribute name using "as", for example, select = {"lc as landcover"} reads lc from the database but replaces the name to landcover in the Cells.
	theme: Name of the theme to be loaded.
	user: Username (default is "").
	where: A SQL restriction on the properties of the Agents (default is "", applying no restriction. Only the Agents that reflect the established criteria will be loaded). This argument ignores the "as" flexibility of select.
	
	Attributes of Society that might be used carefully by the modeler:
	agents: a vector of Agents pointed by the Society.
	instance: a function used to build the Agent.
	counter: unique identifier used to represent the last Agent added to the Society. The next Agent will have ‘counter+1’ as id.
	lastSynchronize: the last time synchronize() was activated. It has zero as initial value.
	messages: a vector that contains the delayed messages.
	parent: the Environment it belongs.

add `soc:add(agent)`	Add a new Agent to the Society. **1st)** An Agent.
clear `soc:clear()`	Remove all the Agents from the Society.
createSocialNetwork	Create a directed SocialNetwork for each Agent of a Society. The following arguments represent the strategies, which must be only one for call:

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td><pre>soc:createSocialNetwork {
  quantity = 2
}

soc:createSocialNetwork {
  probability = 0.15
  name = "random"
}

soc:createSocialNetwork {
  neighbor = "1"
  name = "byneighbor"
}</pre></td><td><b>strategy:</b> a string with the strategy to be used for creating the SocialNetwork. See the table below.
<table>
<tr><th>Strategy</th><th>Description</th><th>Parameters (<b>bold</b> are compulsory)</th></tr>
<tr><td>"cell"</td><td>Create a dynamic SocialNetwork for each Agent of a Society with every Agent within the same Cell the Agent belongs.</td><td>name, self</td></tr>
<tr><td>"func"</td><td>Create a SocialNetwork according to a membership function.</td><td>name, <b>func</b></td></tr>
<tr><td>"neighbor"</td><td>Create a dynamic SocialNetwork for each Agent of a Society with every Agent within the neighbor Cells of the one the Agent belongs.</td><td>name, neighborhood</td></tr>
<tr><td>"probability"</td><td>Applies a probability for each pair of Agents.</td><td>name, self, <b>probability</b></td></tr>
<tr><td>"quantity"</td><td>Number of connections randomly taken from the Society.</td><td>name, self, <b>quantity</b></td></tr>
</table>
<b>func:</b> A function (Agent, Agent)→boolean that returns true if the first Agent will have the second Agent in its SocialNetwork. When using this argument, the default value of strategy becomes "func".
<b>name:</b> name of the relation. Default is "1".
<b>neighborhood:</b> a string with the index of the Neighborhood that will be used to compute the network. Default is "1".
<b>probability:</b> a number between 0 and 1 indicating the probability of each connection. The probability is applied for each pair of Agents. When using this argument, the default value of strategy becomes "probability".
<b>quantity:</b> a number indicating the number of connections each Agent will have, taking randomly from the whole Society. When using this argument, the default value of strategy becomes "quantity".
<b>self:</b> a boolean value indicating whether the Agent can be connected to itself. Default is false.</td></tr>
<tr><td><b>execute</b><pre>soc:execute()</pre></td><td>Execute the Society, activating function execute for each of its Agents.</td></tr>
<tr><td><b>getAgent</b><pre>agent = soc:getAgent("1")</pre></td><td>Return a given Agent based on its index.</td></tr>
<tr><td><b>getAgents</b><pre>agent = soc:getAgents()[1]</pre></td><td>Return a vector with the Agents of the Society.</td></tr>
<tr><td><b>sample</b><pre>agent = soc:sample()</pre></td><td>Return a single sample from the Society.</td></tr>
<tr><td><b>size</b><pre>print(soc:size())</pre></td><td>Return the number of Agents within a Society.</td></tr>
<tr><td><b>split</b><pre>gs = soc:split("sex")
print(gs.male:size())
print(gs.female:size())

gs2 = soc:split(function(ag)
  if ag.age > 60 then
    return "old"</pre></td><td>Split the Society into a set of Groups according to a classification strategy. The generated Groups have empty intersection and union equals to the whole CellularSpace (unless function below returns nil for some Agent). It works according to the type of its only and compulsory argument, that can be:
<table>
<tr><th>Type of argument</th><th>Description</th></tr>
<tr><td>string</td><td>The argument must represent the name of one attribute of the Agents</td></tr>
</table></td></tr>
</table>

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td><pre>
   else
      return "notold"
   end
end)
print(ts.old:size())
</pre></td><td><table>
<tr><td></td><td>of the Society. Split then creates one Group for each possible value of the attribute using the value as index and fills them with the Agents that have the respective attribute value.</td></tr>
<tr><td>function</td><td>The argument is a function that receives an Agent as argument and returns a value with the index that contains the Agent. Groups are then indexed according to the returning value.</td></tr>
</table></td></tr>
<tr><td><b>synchronize</b><pre>
soc:synchronize()
soc:synchronize(2)
</pre></td><td>Activate each asynchronous message sent by Agents belonging to the Society.
<b>1st)</b> A number indicating the current delay to be delivered. Messages with delay less or equal this value are sent, while the others have their delays reduced by this value. Default is one.</td></tr>
</table>

State

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td><b>State</b><pre>
State {
    id = "working",
    Jump{...},
    Flow{...}
}
</pre></td><td>A container of two kinds of rules: Jumps and Flows, plus one id, to identify itself in the Jumps of other States.</td></tr>
</table>

Timer

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td><b>Timer</b><pre>
timer = Timer {
  Event {...},
  Event {...}
}
</pre></td><td>A Timer is an event-based scheduler that executes and controls the simulation. It contains a set of Events. It allows the model to take into consideration processes that start independently and act in different periodicities. It starts with time 0 and, once it is in a given time n, it ensures that all the Events before that time were executed.</td></tr>
<tr><td><b>add</b><pre>
timer:add(Event{...})
</pre></td><td>Add a new Event to the timer.
<b>1st)</b> An Event.</td></tr>
<tr><td><b>execute</b><pre>
timer:execute(2013)
</pre></td><td>Execute the timer until a given time.
<b>1st)</b> The time to stop the simulation. The timer will stop when there is no Event scheduled to a time less or equal to the final time.</td></tr>
<tr><td><b>getTime</b><pre>
print(timer:getTime())
</pre></td><td>Return the current simulation time.</td></tr>
<tr><td><b>reset</b><pre>
timer:reset()
</pre></td><td>Resets the timer to time zero, keeping the same queue.</td></tr>
</table>

Trajectory (Inherits CellularSpace)

Function	Description

Trajectory
`traj = Trajectory {`
`  target = cs,`
`  select = function(c)`
`    return c.cover == "forest"`
`  end,`
`  greater = function(c, d)`
`      return c.dist < d.dist`
`  end`
`}`
 	
`traj = Trajectory {`
`  target = cs,`
`  greater = function(c, d)`
`      return c.dist < d.dist`
`  end`
`}`
 	
`traj = Trajectory {`
`  target = cs,`
`  build = false`
`}`
	Type that defines a spatial trajectory over Cells. It inherits CellularSpace; therefore it is possible to use all functions of such type within a Trajectory. For instance, calling forEachCell() also traverses Trajectories.
	
	target: The CellularSpace over which the Trajectory will take place.
	select: A function (Cell)→boolean to filter the CellularSpace, adding to the Trajectory only those Cells whose returning value is true. If this argument is missing, all Cells will be included in the Trajectory.
	greater: A function (Cell, Cell)→boolean to sort the generated subset of Cells. It returns true if the first one has priority over the second one. If this argument is missing, no sorting function will be applied. See greaterByAttribute() and greaterByCoord() as predefined options to sort objects.
	build: A boolean value indicating whether the Trajectory will be computed or not when created. Default is true.
	
	Attributes of Trajectory that can be used as *read-only*by the modeler:
	cells: A vector of Cells pointed by the Trajectory.
	parent: The CellularSpace where the Trajectory takes place.
	select: The last function used to filter the Trajectory.
	greater: The last function used to sort the Trajectory.

clone `copy = traj:clone()`	Return a copy of the Trajectory, with the same parent, select, greater and Cells.

filter
`traj:filter(function(cell)`
`  return cell.cover = "forest"`
`end)`
	Apply a filter over the original CellularSpace.
	1st) A function such as the parameter select.

randomize `traj:randomize()`	Randomize the Cells, changing their traversing order.
rebuild `traj:rebuild()`	Rebuild the Trajectory from the original data using the last filter and sort functions.

sort
`traj:sort(function(c, d)`
`  return c.dist < d.dist`
`end)`
	Sort the current CellularSpace subset.
	1st) An ordering function with the same signature of argument greater.

Other Functions

Function	Description

coord2index
`idx = coord2index(2, 3, 10)`
`c = Coord{x = 2, y = 3}`
`cs:getCell(c).value = 3`
`print(cs.cells[idx].value) -- 3`
	Convert a pair (x, y), which represents a position in a squared and regular CellularSspace, into the position where the Cell is stored in the CellularSpace's vector of Cells.
	1st) The x position.
	2nd) The y position.
	3rd) Number of columns of the CellularSpace.

forEachAgent
`forEachAgent(group, function(ag)`
`  ag.age = ag.age + 1`
`end)`
	Second order function to transverse a given Society, Group, or Cell, applying a function in each of its Agents. It returns true if no call to the function taken as argument returns false.
	1st) A Society, Group, or Cell.

Function	Description

`forEachAgent(cs, function(cell)`
`  cell.water = cell.water + 1`
`end)`
	2nd) A function that takes one single Agent as argument. If some call to func returns false, forEachAgent stops and does not process any other Agent.

forEachCell
`forEachCell(cs, function(cell)`
`  -- ...`
`end)`
	Second order function to transverse a given CellularSpace, applying a given function on each of its Cells. It returns true if no call to the function taken as argument returns false.
	1st) A CellularSpace.
	2nd) A function that takes a Cell as argument. If it returns false when processing a given Cell, forEachCell stops and does not process any other Cell.

forEachCellPair
`func = function(cella, cellb)`
`  -- ...`
`end`
 	
`forEachCellPair(csa, csb, func)`
	Second order function to transverse two CellularSpaces with the same resolution and number of Cells, applying a function that receives as argument two Cells, one from each CellularSpace, that share the same (x, y). It returns true if no call to the function taken as argument returns false.
	1st) A CellularSpace.
	2nd) Another CellularSpace.
	3rd) A function that takes two Cells as arguments, one coming from cs1 and the other from cs2. If some call to f returns false, forEachCellPair stops and does not process any other pair of Cells.

forEachElement `forEachElement(ag, print)`	Second order function to transverse a given object, applying a function to each of its elements. It can be used for instance to trasverse all the elements of an Agent or an Enviroment. It returns true if no call to the function taken as argument returns false. **1st)** A TerraME object or a table. **2nd)** A function that takes three arguments: the index of the element, the element itself, and the type of the element.

forEachNeighbor
`myf = function(cell, neighbor)`
`  -- ...`
`end`
 	
`forEachNeighbor(c, myf)`
`forEachNeighbor(c, "idx", myf)`
	Second order function to transverse a given Neighborhood of a Cell, applying a function in each of its neighbors. It returns true if no call to the function taken as argument returns false. There are two ways of using this function because the second argument is optional.
	1st) A Cell.
	2nd) (Optional) A string with the name of the Neighborhood to be transversed. Default is “1”.
	3rd) A function that takes three arguments: the Cell itself, the neighbor Cell, and the connection weight. If some call to f returns false, forEachNeighbor stops and does not process any other neighbor. In the case where the second argument is missing, this function becomes the second argument.

forEachNeighborhood
`myf = function(cell, nhood)`
`  -- ...`
`end`
 	
`forEachNeighborhood(c, myf)`
	Second order function to transverse all Neighborhoods of a Cell, applying a given function on them. It returns true if no call to the function taken as argument returns false.
	1st) A Cell.
	2nd) A function that receives a Neighborhood as parameter.

forEachConnection

```
mf = function(a, r, w)
  a:message {
    receiver = r,
    type = "money",
    quant = 2 * w
  }
end

forEachConnection(ag,mf)
forEachConnection(ag, "job", mf)
```

Second order function to transverse the connections of a given Agent, applying a function to each of them. It returns true if no call to the function taken as argument returns false. There are two ways of using this function because the second argument is optional.

1st) An Agent.

2nd) (Optional) A string with the name of the SocialNetwork to be transversed. Default is “1”.

3rd) A function that takes three arguments: the Agent itself, its connection, and the connection weight. If some call to f returns false, forEachConnection stops and does not process any other connection. In the case where the second argument is missing, this function becomes the second argument.

forEachSocialNetwork

```
myf = function(a, socnet)
  -- ...
end

forEachSocialNework(ag, myf)
```

Second order function to transverse all SocialNetworks of an Agent, applying a given function over them. It returns true if no call to the function taken as argument returns false.

1st) An Agent.

2nd) A function that receives a SocialNetwork as parameter.

greaterByAttribute

```
s = greaterByAttribute("cover")

t = Trajectory {
  target = cs,
  sort = s
}
```

Return a function that compares two tables (which can be, for instance, Agents or Cells) and returns which one has a priority over the other, according to an attribute of the objects and a given operator.

1st) A string with the name of the attribute.

2nd) A string with the operator, which can be “>”, “<”, “<=”, or “>=”. Default is “<”.

greaterByCoord

```
t = Trajectory {
  target = cs,
  sort = greaterByCoord()
}
```

Return a function that compares two tables with x and y attributes (basically two regular Cells) and returns which one has a priority over the other, according to a given operator.

1st) A string with the operator, which can be “>”, “<”, “<=”, or “>=”. Default is “<”.

index2coord

```
mx, my = index2coord(7, 10)
c = Coord{x = mx, y = my}
cs:getCell(c).value = 3
print(cs.cells[7].value) -- 3
```

Convert the position where the Cell is stored in the CellularSpace’s vector of Cells into a pair (x, y), which represents a position in a squared and regular CellularSpace.

1st) The position of the Cell in the vector of Cells.

3rd) Number of columns of the CellularSpace.

integrate

```
v = integrate {
  equation = function(t, y)
    return t - 0.1 * y
  end,
  method = "euler",
  initial = 0,
  a = 0,
  b = 100,
  step = 0.1
}
```

A second order function to numerically solve ordinary differential equations with a given initial value.

method: the name of a numeric algorithm to solve the ordinary differential equations in a given [a,b[interval. See the options below.

Method	Description
"euler" (default)	Euler method
"heun"	Heun (Second Order Euler)
"rungekutta"	Runge-Kutta Method (Fourth Order)

equation: A differential equation or a vector of differential equations. Each equation is described as a function of one or two parameters that returns a value of its derivative f(t, y), where t is the time instant, and y starts with the value of attribute initial and changes according to the result of f() and the chosen method. The calls to f will use the first parameter (t) in the interval [a,b[, according to the parameter step.

initial: The initial condition, or a vector of initial conditions, which must be satisfied. Each initial condition represents the value of y when t (first parameter of f) is equal to the value of parameter a.

a: The beginning of the interval.

b: The end of the interval.

step: The step within the interval (optional, using 0.1 as default). It must satisfy the condition that (b - a) is a multiple of step.

	event: An Event, that can be used to set parameters a and b with values event:getTime() **-** event:getPeriodicity() and event:getTime(), respectively. The period of the event must be a multiple of step. Note that the *first* execution of the event will compute the equation relative to a time interval between event.time **-** event.period and event.time. Be careful about that.

elapsedTime
`x = os.time()`
`for i = 1, 400000000 do end`
`y = os.time()`
`elapsedTime(y - x)`
	Convert the time from the os library to a more readable value, a string in the format "hours:minutes:seconds", or "days:hours:minutes:seconds" if the elapsed time is more than one day.
	1st) A given time.

type
`c = Cell{value = 3}`
`print(type(c))`
	Return the type of an object. It extends the original Lua type() to support TerraME objects, whose type name (for instance "CellularSpace" or "Agent") is returned instead of "table".
	1st) Any object or value.