TerraME Classes 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 0.6
February 29, 2012

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

- Names of classes have the Pascal 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 CamelCase[1] 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. It is possible to use less arguments than the function signature, with missing arguments taking their default values, but the arguments must follow the specified order. 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 argument arg1, v2 is the value of argument arg2, and so on. These arguments can be used in any order. Every class 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 { ... }`
`}`
	Class 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:
	• **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.
	• **type:** a string containing "Agent".
	• **socialnetworks:** a set of SocialNetwork, with the connections of the Agent.
	• **placement:** a Trajectory representing the default placement of the Agent. (Only when the Agent belongs to an Environment - by itself or through a Society)
	• **cells:** a vector of Cells necessary to use forEachCell(agent). This value is the same of "placement.cells".

add	Add a new Trajectory or State to the Agent/Automaton. **1st)** A State or Trajectory.
addSocialNetwork	Add a new SocialNetwork to the Agent. **1st)** A SocialNetwork. **2nd)** Name of the relation.

[1] http://en.wikipedia.org/wiki/CamelCase

Function	Description
build	Check if the state machine was correctly defined, verifying whether the targets match the ids of the States.
dye	Remove the Agent from the Society it belongs and clear its placement relations.
enter	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	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	Return the Cell where the Agent is located according to its placement. It assumes that each Agent belongs to at most one Cell.
getCells	Returns the Cells pointed by the Agent according to its placement.
getLatency	Return the time when the machine executed the transition to the current state. Before running, the latency is zero.
getSocialNetwork	Return a SocialNetwork of the Agent given its name. **1st)** Name of the relation.
getStateName	Return a string with the current state name.
leave	Remove the Agent from a given Cell. **1st)** A string representing the index to be used. Default is "placement".

message
 	
`ag: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.
	type: A string describing the function that will be called in the receiver. 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	Move the Agent to a new Cell. **1st)** The new Cell. **2nd)** A string representing the index to be used. Default is "placement".
on_*	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).
reproduce `child = agent:reproduce{age=0}`	Create an Agent with the same behavior in the same place where the original Agent is (according to its placement). Additional properties of the new Agent can be passed as argument (a table). The new Agent is pushed into the same Society the original Agent belongs. The returning value of this function is the new Agent.
setTrajectoryStatus	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 described as a state machine will automatically traverse all trajectories defined within it.

Automaton

Function	Description

Automaton
 	
`automaton = Automaton {`
`  id = "MyAutomaton",`
`  State { ... },`
`}`
	A hybrid state machine that is 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.
	• **parent:** The Environment it belongs.
	• **type:** A string containing "Automaton".

add	Add a new State to the Automaton. **1st)** A State.
build	Check if the state machine was correctly defined, verifying whether the targets match the ids of the States.
execute	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.
getLatency	Return the time when the machine executed the transition to the current state. Before running, the latency is zero.
setTrajectoryStatus	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	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.
createObserver	Create a new observer for the Cell. See the Observer documentation below.
first	Start a Neighborhood iterator, pointing to the first element of the Neighborhood list.
getAgent	Return the Agent that belongs to a given Cell. It assumes that there is at most one Agent per Cell.
getAgents	Return the Agents that belong to a given Cell.
getCurrentNeighborhood	Retrieve the Neighborhood currently pointed by the Neighborhood iterator, or nil otherwise.
getNeighborhood	Return one of the Neighborhoods of a Cell. **1st)** A string with the Neighborhood's name to be retrieved (default is "1").
getPast	Return the values of the Cell in the last time synchronize() was called.
isFirst	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 Neighborhood of the list, or whether the iterator does not exist.
last	Clear the Neighborhood iterator.
next	Update the Neighborhood iterator to the next Neighborhood of the list.
notify	Notify every observer connected to the Cell. **1st)** The time to be used in the observer. Most of the strategies available ignore this

Function	Description
	value; therefore it can be left empty. See the Observer documentation for details.
size	Return the number of Neighborhoods of a 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"
}
 	
cs = CellularSpace {
database = "d:\\GoF.mdb",
layer = "cells_10",
theme = "cells_10",
select = "height3 as height",
where = "height3 > 200"
}
 	
cs2 = 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.
	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.
	ydim: 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 *read-only* by the modeler:
	• **cells:** A vector of Cells pointed by the CellularSpace.
	• **cObj_:** A pointer to a C++ object.
	• **parent:** The Environment it belongs.
	• **type:** A string containing "CellularSpace".

add	Add a new Cell to the CellularSpace. The new Cell is added to the end of the cells vector. **1st)** A Cell.
createObserver	Create a new observer for the CellularSpace. See the Observer documentation below.

createNeighborhood
 	
cs:createNeighborhood( )
-- *moore*
 	
cs:createNeighborhood {
strategy = "vonneumann",
self = false
}
	Create a Neighborhood for each Cell of the CellularSpace. It gets a table as argument, with the following attributes:
	strategy: A string with the strategy to be used for creating the Neighborhood. See the table below.
	
	Strategy | Description | Parameters (**bold** are compulsory)
	"moore" (default) | A Moore (queen) Neighborhood. | name, self, wrap
	"vonneumann" | A von Neumann (rook) Neighborhood | name, self

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

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

</td><td>
<table>
<tr><td>"mxn"</td><td>(M*2+1) x (N*2+1) (columns x rows) Neighborhood</td><td>name, M, N, filter, weight</td></tr>
<tr><td>"3x3"</td><td>A 3x3 (Couclelis) Neighborhood</td><td>name, filter, weight</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>
</table>
filter: A function(Cell, Cell)→bool, where the first argument is the Cell itself and the other represent its neighbor. It returns whether neighbor will be included in the relation.
M: Number of columns.
N: Number of rows.
name: A string with the name of the Neighborhood to be created. Default is "1".
self: 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.
weight: 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.
wrap: Whether Cells in the borders will be connected to the Cells in the opposite border. Default is false.
</td></tr>
<tr><td>getCell</td><td>Retrieve a Cell from the CellularSpace, given its index.
1st) A Coord.</td></tr>
<tr><td>getCells</td><td>Return a vector containing all Cells of the CellularSpace.</td></tr>
<tr><td>load</td><td>Load the CellularSpace from the database. TerraME automatically executes this function when the CellularSpace is created, but it can be execute this to load the attributes again, erasing each other attribute and relations created by the modeller.</td></tr>
<tr><td>loadNeighborhood


```
cs:loadNeighborhood("n.gpm")

cs:loadNeighborhood("mtable"
)
```

</td><td>Load a Neighborhood stored in an external source. Each Cell receives its own set of neighbors.
source: 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>
name: A string with the name of the Neighborhood to be loaded within TerraME. Default is "1".
</td></tr>
<tr><td>notify</td><td>Notify every observer connected to the CellularSpace.
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.</td></tr>
<tr><td>sample</td><td>Return a random Cell from the CellularSpace.</td></tr>
<tr><td>save


```
cellularspace:save(20, "table")

cellularspace:save(20, "ntable",
                   {"def", "p"})
```

</td><td>Save the attributes of a CellularSpace into the same database it was retrieved.
1st) A temporal value to be stored in the database, which can be different from the simulation time.
2nd) Name of the table to store the attributes of the Cells.
3rd) 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>size</td><td>Retrieve the number of elements in the CellularSpace.</td></tr>
<tr><td>split


```
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
```

</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. 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>
</table>
</td></tr>
</table>

Function	Description
end) print(ts.gt:size())	function: 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.
synchronize cs:synchronize() cs:synchronize("landuse") cs:synchronize{"water","use"}	Synchronize the CellularSpace, calling the function synchronize() of each Cell. **1st)** 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.

Coord

Function	Description
Coord coord = Coord() coord2 = Coord{x=2, y=3} print(coord2.x) -- nil	Class that stores a pair (x, y). Once created, it is only possible to retrieve (x, y) by using get().
get	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. **1st)** Default is not changing. **2nd)** Default is not changing.

Environment

Function	Description

Environment
 	
environment = Environment {
cs1 = CellularSpace{ ... },
ag1 = Agent{ ... },
aut2 = Automaton{ ... },
t1 = Timer{ ... },
env1 = Environment{ ... }
}
	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.

add	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. **1st)** An Agent, Automaton, CellularSpace, Timer, or Environment.
createPlacement	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. **strategy:** A string containing the strategy to be used for creating a placement between Agents and Cells. See the options below.

Strategy	Description	Parameters
"random" (default)	Create placements between Agents and Cells randomly, putting each Agent in a Cell randomly chosen.	name, max
"uniform"	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.	name
"void"	Create only the pointers for each object in each side, preparing the objects to be manipulated by the modeler.	name

name: Name of the relation in TerraME objects. Default is "placement", which means that the modeller can use enter(), move(), and leave() directly. If the name is

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td></td><td>different from the default value, the modeller will have to use the last argument of these functions to indicate which relation they are changing or perform changes on these relations manually.
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.</td></tr>
<tr><td>createNeighborhood</td><td>Create Neighborhoods between Cells belonging to two different CellularSpaces. It uses the two first CellularSpaces pushed into the Environment.
strategy: 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 (bold are compulsory)</th></tr>
<tr><td>"mxn"</td><td>Create an [M*2+1] x [N*2+1] (columns x rows) bidirected Neighborhood (Couclelis) bettween two different CellularSpaces. It supposes that both CellularSpaces have the same resolution and extent.</td><td>name, M, N, filter, weight</td></tr>
<tr><td>"id"</td><td>Create a 1:1 Neighborhood between two different CellularSpaces by connecting Cells with the same id. Note that it only uses two CellularSpaces; when someone wants to connect three or more, they need to be connected two by two.</td><td>name</td></tr>
<tr><td>“hierarchy”</td><td>Create a Neighborhood based on the differences of resolutions between two CellularSpaces. For example, Cell (1,1) of a given CellularSpace will be connected to Cells (1,1), (1,2), (2,1), and (2,2) of another rectangular CellularSpace with four times the number of Cells.</td><td>name</td></tr>
</table>

filter: A function (Cell, Cell)→bool, where the first argument is the Cell itself and the other represent its neighbor. It returns whether neighbor will be included in the relation. The first (second) Cell comes from the first (second) CellularSpace pushed into the Environment.
M: Number of columns.
N: Number of rows.
name: A string with the name of the Neighborhood to be created. Default is "1".
weight: 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. This function is also called twice for each pair of Cells.
filter: A function (Cell, Cell)→bool, where the first argument is a Cell and the other is its neighbor, one from each CellularSpace. It returns whether neighbor will be included in the relation. This function is called twice for each pair of Cells, first filter(c1, c2) and then filter(c2, c1), wher c1 belongs to cs1 and c2 belongs to cs2.
weight: A function (Cell,Cell)→number, where the first argument is a Cell and the second is its neighbor. It calculates the weight of the Neighborhood relation.</td></tr>
<tr><td>createSocialNetwork</td><td>as mesmas estrategias: random, quantity, etc.??</td></tr>
<tr><td>execute</td><td>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.</td></tr>
<tr><td>loadNeighborhood</td><td>Load a Neighborhood between two different CellularSpaces.
1st) Name of the file to be loaded.
2nd) Name of the relation to be created. Default is "1".</td></tr>
<tr><td>loadPlacement</td><td>Load a Placement between a Society and a CellularSpace.
1st) Name of the file to be loaded.
2nd) Name of the relation to be created. Default is "placement".</td></tr>
<tr><td>loadSocialNetwork</td><td>Load a SocialNetwork between two different Societies.</td></tr>
</table>

Function	Description
	1st) Name of the file to be loaded. **2nd)** Name of the relation to be created. Default is "1".
saveNeighborhood	Save a Neighborhood between two CellularSpaces in a file. It overwrites any previous content of the file. **1st)** Name of the file to be saved. **2nd)** Name of the relation to be saved. Default is "1".
savePlacement	Save a Neighborhood between a Society and a CellularSpace in a file. It overwrites any previous content of the file. **1st)** Name of the file to be saved. **2nd)** Name of the relation to be saved. Default is "placement".
saveSocialNetwork	Save a Neighborhood between two Societies in a file. It overwrites any previous content of the file. **1st)** Name of the file to be saved. **2nd)** Name of the relation to be saved. Default is "1".

Event

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

Event {
time = 1985,
period = 2,
priority = –1,
action = function(event)
print(event:getTime())
end
}

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

time: The first instant of time when the Event will occur (default is the current time of the Timer it will belong).
period: The periodicity of the Event (default 1).
priority: Define the priority of the Event over other Events. The default priority is 0 (zero). Smaller values have higher priority.
action: Function from where, in general, the simulation engine services are invoked. This function has one single argument, an Event. If the action returns false, the Event is removed from the Timer and will not be executed again. Action can also take a TerraME object. In this case, each object has a 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>config</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.
1st) The first instant of time when the Event will occur (default is the current time of the Timer it will belong).
2nd) The periodicity of the Event (default is 1).
3rd) Define the priority of the Event over other Events. The default priority is 0 (zero). Smaller values have higher priority.</td></tr>
<tr><td>getPeriod</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>getPriority</td><td>Return the priority of a given Event. This function has restrictions of use as above.</td></tr>
<tr><td>getTime</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**}	Describe 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
 	
richers = Group {
target = society,
filter = **function**(agent)
return agent.money > 90
end,
sort = **function**(a, b)
return a.money > b.money
end
}
 	
groupBySize = Group {
target = society,
sort = **function**(a1, a2)
return a1.size > a2.size
end
}
	Class that defines an ordered selection over a Society. It inherits Society; therefore it is possible to use all functions of such class within a Group. For instance, calling forEachAgent also traverses Groups.
	
	target: The Society over which the Group will take place.
	filter: 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.
	sort: 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.
	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.
	• **lastFilter:** The last filter function applied to the Group.
	• **lastSort:** The last sort function applied to the Group.

rebuild	Rebuild the Group from the original data using the last filter and sort functions.
randomize	Randomizes the Agents, changing the traversing order.

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{
type = "number",
grouping = "uniquevalue",
slices = 3,
maximum = 2,
minimum = 0,
colorBar = {
{WHITE, 0},
{RED, 1},
{GREEN, 2}
}
}
	Class 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	Paint 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

Function	
	<table><tr><td>uniquevalue</td><td>Paint objects with each attribute value corresponding to a given color. String attributes can only belong to uniquevalue groupings.</td><td>colorBar, type, width</td></tr></table> **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 also a table with the color. In the case of unique value, each position needs to have also a value of the respective attribute. **stdColorBar:** A table similar to the previous parameter. It is needed only when the 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	Add a new Cell to the Neighborhood. **1st)** A Coord. **2nd)** An object of class CellularSpace that contains the Cell to be added. **3rd)** A number representing the weight of the connection (default 0).
clear	Remove all Cells from the Neighborhood. In practice, it has almost the same behavior as calling Neighborhood() again.
eraseCel	Remove a Cell from the Neighborhood. **1st)** A Coord.
first	Start a neighbor iterator, pointing to the first Cell in the neighbors list.
getCellNeighbor	Return a neighbor, given its coords. **1st)** A Coord.
getCellWeight	Return the weight of the connection to a given neighbor Cell. **1st)** A Coord.
getCoord	Return the coordinates of the neighbor pointed by the current iterator.

getID
`c1:addNeighborhood(n, "n")`
`c2:addNeighborhood(n, "n2")`
`n:getID() -- "name2"`
	Return the name of the Neighborhood in the last Cell it was added.

getNeighbor	Return the neighbor pointed by the current iterator.
getWeight	Return the weight of the connection to a neighbor pointed by the current iterator.
isEmpty	Return whether the Neighborhood does not contain any Cell.
isFirst	Return whether the neighbor iterator is pointing to the first Cell of the list.
isLast	Return whether the neighbor iterator has already passed by the last Cell of the list, or whether the iterator does not exist.
last	Clear the neighbor iterator.
next	Change the neighbor iterator to the next Cell of the list.
sample	Return a single sample from the Neighborhood.
setCellWeight	Update the weight of a connection to a neighbor. **1st)** A Coord. **2nd)** A number pointing out the new weight.
setWeight	Update the weight of the connection to a neighbor pointed by the current iterator. **1st)** A number representing the new weight.
size	Retrieve the number of neighbors the Neighborhood has.

Observer

Function

Observer

```
observer = Observer {
   subject = cs,
   attributes = "soilWater",
   subtitles = soilWaterLeg
}

Observer {
   subject = mytrajectory,
   observer = observer
}
```

Description

Observer is the way to collect data from the objects of a model in order to save or to graphically plot them. Observers can be created from any TerraME object and do not need to be instantiated to a Lua object to work properly.

type: A string to define the way to observe a given object. See the table below.

Type	Description	Parameters (**bold** are compulsory)
chart	Create a line chart showing the variation of an attribute (y axis) of an object. X axis can be another attribute or a temporal value coming from the argument of notify().	**subject,** attributes, xaxis, xLabel, yLabel, title, curveLabel
image	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.	**subject**, **attribute,** file, legend
logfile	Save attributes of an object into a csv text file, with one row for each notify().	**subject,** file, attributes**,** separator, mode
map	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.	**subject**, **attribute**, **observer** (unless when the subject is a CellularSpace), legend
neighborhood	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.	**subject, observer**, neighIndex, neighType
scheduler	Create a display with the current time and Event queue of a given Timer.	**subject**
statemachine	Draw the state machine of an Automaton in a Cell or an Agent. As default, states are drawn 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, location** (only when the 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

attribute: 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.
legend: 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

Function	Description
	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 cannot 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 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 "". **curveLabel:** 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
host ~= nil *or* port ~= nil	udpsender
neighIndex ~= nil *or* neighType ~= nil	neighborhood
type(subject) == "Timer"	scheduler
type(subject) == "Event"	table
type(subject) == "CellularSpace"	map
type(subject) == "Trajectory"	map
type(observer) == "Observer" *and* type(subject) == "Cell"	neighborhood
type(subject) == "Cell"	table
type(subject) == "Automaton"	map
type(subject) == "Agent"	statemachine
type(subject) == "Society"	map
type(subject) == "Group"	map

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	Add a new connection to the SocialNetwork. **1st)** An Agent. **2nd)** A number representing the weight of the connection (default nil - no weight).
clear	Remove all Agents from the SocialNetwork. In practice, it has almost the same behavior as calling SocialNetwork() again.
getConnection	Return a connection given its id. **1st)** The unique identifier of an Agent.
getID	Return the ID used to index the SocialNetwork into the Agent.
getWeight	Return the weight of a given connection. **1st)** An Agent.
isEmpty	Return whether the SocialNetwork does not contain any connection.
isConnection	Return whether a given Agent is a connection.

Function	Description
	1st) An Agent.
remove	Remove an Agent from the SocialNetwork. **1st)** An Agent.
sample	Return a single sample from the SocialNetwork.
setWeight	Update the weight of a connection. **1st)** An Agent. **2nd)** A number pointing out the new weight.
size	Retrieve the number of connections the SocialNetwork has.

Society

Function	Description
Society s = Society{instance=..., file=""}	Class to create and manipulate a set of Agents. Each Agent within a Society has a unique id, which is initialized while creating the Society. 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 • **build(agent):** 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. Ver como vai ficar o save para o as. Documentar a questao de que nem todos os atributos sao lidos como esta no banco. **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 can 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	Add a new Agent to the Society. This Agent is
clear	Remove all the Agents from the Society.
createSocialNetwork soc:createSocialNetwork {	Create a directed SocialNetwork for each Agent of a Society. The following arguments represent the strategies, which must be only one for call: **strategy:** a string with the strategy to be used for creating the SocialNetwork. See the table below.

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

```
    quantity = 2
}

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

soc:createSocialNetwork {
    neighbor = "1"
    name = "byneighbor"
}
```

</td><td>
<table>
<tr><th>Strategy</th><th>Description</th><th>Parameters (bold are compulsory)</th></tr>
<tr><td>"quantity"</td><td>Number of connections randomly taken from the Society</td><td>name, self, quantity</td></tr>
<tr><td>"probability"</td><td>Applies a probability for each pair of Agents.</td><td>name, self, probability</td></tr>
<tr><td>"func"</td><td>Create a dynamic SocialNetwork according to a membership function.</td><td>name, func</td></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>"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>
</table>
bidirected: a boolean value indicating that, for each connection from A to B, it will also exist a connection from B to A. Default is false.
dynamic: a boolean value indicating whether the relation is computed on-the-fly or statically. Default is false (statically). Computing dunamically requires less memory, but requires more processing time.
func: a function that receives an Agent as argument and returns its SocialNetwork. When using this argument, the default value of strategy becomes "func".
name: name of the relation. Default is "1".
neighborhood: a string with the index of the Neighborhood that will be used to compute the network. Default is "1".
probability: 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".
quantity: 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".
self: a boolean value indicating whether the Agent can be connected to itself. Default is false.
</td></tr>
<tr><td>execute</td><td>Execute the Society, activating function execute for each of its Agents.</td></tr>
<tr><td>getAgent</td><td>Return a given Agent based on its index.</td></tr>
<tr><td>getAgents</td><td>Return a vector with the Agents of the Society.</td></tr>
<tr><td>loadSocialNetwork</td><td>Load a social network stored in an external source.</td></tr>
<tr><td>sample</td><td>Return a single sample from the Society.</td></tr>
<tr><td>size</td><td>Return the number of Agents within a Society.</td></tr>
<tr><td>synchronize</td><td>Activate each asynchronous message sent by Agents belonging to the Society. Messages with delay one are sent, while the others have their delay reduced by one.
1st) A number indicating the current time. Default is Society::lastSynchronize+1.</td></tr>
</table>

State

Function	Description

State
 	
`State {`
`  id = "working",`
`  Jump{ ... },`
`  Flow{ ... }`
`}`
	A container of two kinds of rules: Jumps and Flows, plus one id, to identify itself in the Jumps of other States.

Timer

Function	Description

Timer
 	
timer = Timer {
  Event { ... },
  Event { ... }
}
	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.

add	Add a new Event to the timer. **1st)** An Event.
execute	Execute the timer until a given time. **1st)** 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.
getTime	Return the current simulation time.
reset	Resets the timer to time zero, keeping the same queue.

Trajectory (Inherits CellularSpace)

Function	Description

Trajectory
 	
traj = Trajectory{
  target = cs,
  filter = **function**(cell)
  return cell.cover == "forest"
  end,
  sort = **function**(c, d)
  return c.dist < d.dist
  end
}
 	
traj = Trajectory{
  target = cs,
  sort = **function**(c, d)
  return c.dist < d.dist
  end
}
 	
traj = Trajectory{
  target = cs,
  build = **false**
}
	Class that defines a spatial trajectory over Cells. It inherits CellularSpace; therefore it is possible to use all functions of such class within a Trajectory. For instance, calling forEachCell also traverses trajectories.
	
	target: The CellularSpace over which the Trajectory will take place.
	filter: 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.
	sort: 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 compareByAttribute() and compareByCoord() 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.
	• **lastFilter:** The last filter function applied to the Trajectory.
	• **lastSort:** The last sort function applied to the Trajectory.

filter	Apply a filter over the original CellularSpace. **1st)** A function such as the second parameter of the Trajectory constructor.
getCell	Return a Cell given its index. **1st)** A Coord.
randomize	Randomize the Cells, changing their traversing order.
rebuild	Rebuild the Trajectory from the original data using the last filter and sort functions.
remove	Remove a Cell from the Trajectory.
sort	Sort the current CellularSpace subset. **1st)** An ordering function.

Other Functions

Function	Description
coord2index idx = coord2index(2, 3, 10) c = Coord{x=2, y=3} cs:getCell(c).value = 3 print(cs.cells[7].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.

Function	Description
d df = **function**(x, y) **return** y - x^2+1 **end** a = 0 b = 2 init = 0.5 delta = 0.2 d{df, init, a, b, delta}	CONVERSAR COM TIAGO SOBRE ESTA FUNCAO. OS ARGUMENTOS ATUALMENTE SAO DESCRITOS NA FORMA DE UMA TABELA, MAS NAO SAO NOMEADOS! se o init nao for passado, o valor dele será df(a). a funcao d poderia receber um evento como argumento, e assim evitar os parametros 'a' e 'b', e assim integrar a modelagem de tempo continuo usando eventos discretos. Assim, os parametros de d poderiam ser nomeados. Por exemplo: d{equation=f, step=0.1, event = e, method = "euler"} RESOLVER AQUI TAMBEM A QUESTAO DAS CONSTANTES INTEGRATION_METHOD E DELTA. A second-order function to calculate a numerical integration of a given function. It uses two global variables, which define the INTEGRATION_METHOD (integrationEuler as initial value, others available are integrationHeun and integrationRungeKutta, but the user can also define a function) and DELTA (0.2 as initial value). **1st)** A differential equation, described as a function of two parameters, x and y, which returns a single value. **2nd)** The initial condition which must be satisfied, basically the value of f(a), where f is the first argument, and a is the third argument. **3rd)** The beginning of the interval. **4th)** The end of the interval. **5th)** The step of the interval (optional, using DELTA as default).
forEachAgent forEachAgent(s, **function**(a) a.age = a.age + 5 **end**)	Transverse a given Society, Group, or Cell, applying a function in each of its Agents. **1st)** A Society, Group, or Cell. **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**)	Transverse a given CellularSpace, applying a given function on each of its Cells. **1st)** A CellularSpace. **2nd)** A function that takes an object of class Cell as argument. If f returns false when processing a given Cell, forEachCell stops and does not process any other Cell.
forEachCellPair forEachCellPair(ca,cb, **function**(a,b) -- ... **end**)	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). **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)	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. **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, n) -- ... **end** forEachNeighbor(c, myf)	Transverse a given Neighborhood of a Cell, applying a function in each of its neighbors. There are two signatures for this function, according to the number of arguments used: forEachNeighbor(cell, f) or forEachNeighbor(cell, index, f) **1st)** A Cell object. **2nd)** 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. **3rd)** A string with the name of the Neighborhood to be used.
forEachNeighborhood myf = **function**(cell, nhood) -- ... **end**	Transverse all Neighborhoods of a Cell, applying a given function on them. **1st)** A Cell. **2nd)** A function that receives a Neighborhood as parameter.

Function	Description
forEachNeighborhood(c, myf)	
forEachConnection myf = **function**(a, r, w) a:message{receiver=r, type="money", quant=2*w} **end** forEachConnection(ag, myf)	Transverse the connections of a given Agent, applying a function to each of them. **1st)** An Agent. **2nd)** A function that takes three arguments, two Agents (the Agent and its connection) and the weight of the relation. If some call to func returns false, forEachConnection stops and does not process any other connection.
forEachSocialNetwork myf = **function**(a, socnet) -- ... **end** forEachSocialNework(ag, myf)	Transverse all SocialNetworks of an Agent, applying a given function over them. **1st)** An Agent. **2nd)** A function that receives a SocialNetwork as parameter.
index2coord x,y=index2coord(7, 10) c = Coord{x=x, y=y} 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), that represents a position in a squared and regular CellularSpace. **1st)** The x position. **2nd)** The y position. **3rd)** Number of columns of the CellularSpace.
compareByAttribute s = compareByAttribute("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 "<".
compareByCoord g = compareByCoord() t = Trajectory{target=cs, sort=g}	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 "<".
integrationEuler df = **function**(x, y) **return** y - x^2+1 **end** integrationEuler(d,1,0,9,1)	Euler method to integrate ordinary differential equations in a given [a,b[interval. **1st)** A differential equation, described as a function of two parameters, x and y, which returns a single value. **2nd)** The initial condition which must be satisfied, basically the value of df(a). **3rd)** The beginning of the interval (a). **4th)** The end of the interval (b). **5th)** The step of the interval.
integrationHeun df = **function**(x, y) **return** y - x^2+1 **end** integrationHeun(d,1,0,9,1)	Heun (Second Order Euler) method to integrate ordinary differential equations in a given [a,b[interval. **1st)** A differential equation, described as a function of two parameters, x and y, which returns a single value. **2nd)** The initial condition which must be satisfied, basically the value of df(a). **3rd)** The beginning of the interval (a). **4th)** The end of the interval (b). **5th)** The step of the interval.
integrationRungeKutta d = **function**(x, y) **return** y - x^2+1 **end** integrationRungeKutta(d,1,0,9,1)	Runge-Kutta Method (Fourth Order) to integrate ordinary differential equations in a given [a,b[interval. **1st)** A differential equation, described as a function of two parameters, x and y, which returns a single value. **2nd)** The initial condition which must be satisfied, basically the value of df(a). **3rd)** The beginning of the interval (a). **4th)** The end of the interval (b). **5th)** The step of the interval.
performanceTime x = os.time() **for** i = 1, 400000000 **do end** y = os.time() performanceTime(y-x)	O NOME DESTA FUNCAO ESTA MUITO RUIM! Convert the time from the os library to a more readable value, a string in the format "days:hours:minutes:seconds". **1st)** A given time.
type	Return the type of an object. It extends the original Lua type() to support TerraME objects, whose class name (for instance "CellularSpace" or "Agent") is returned instead of "table".

Function	Description
	1st) A Lua object.