

Agent-Based Modeling in TerraME

Pedro Ribeiro de Andrade
Talita Oliveira Assis
Tiago Garcia de Senna Carneiro
Antônio Miguel Vieira Monteiro
Ana Paula Aguiar
Gilberto Câmara

pedro.andrade@inpe.br, talitaoliveiraassis@gmail.com, tiago@iceb.ufop.br, miguel@dpi.inpe.br,
{ana.aguiar, gilberto.camara}@inpe.br

Introduction

TerraME is a multiparadigm framework for developing dynamic models of geospatial phenomena. The lowest level of model development where it is possible to describe the minimum entities that act over space is through agent-based modeling, or simply ABM. This chapter describes the main TerraME functionalities related to this topic. Some good introductions to ABM can be found in Gilbert (2007) and Gilbert and Troitzsch (2005).

The main components and traversing functions of the architecture for ABM presented in this document is shown in Figure 1. For a complete description of each function and their parameters see Andrade and Carneiro (2011).

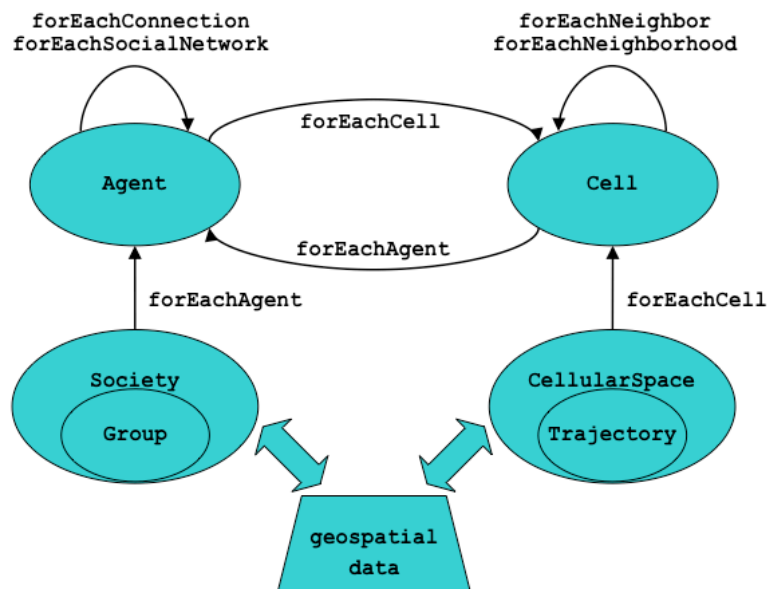


Figure 1: Entities and relations for ABM in TerraME.

Some basic definitions related to the syntax adopted in this document are noteworthy. To allow a smooth reading, we use names of classes and objects in plain English, avoiding capital letters and words without space between them whenever possible (e.g. cellular space instead of CellularSpace). Functions are always described with “()” in the end, with the ones that belong to classes being

described as `Class::function()`. The complete source code of each example is also available at www.terrame.org.

Basic Classes

The basic entity of any agent-based model is the *agent*. In TerraME, one agent can be built simply by using a table with attributes and functions, instantiated by the constructor `Agent()`. It enables the agent to perform basic actions that will be described along this chapter. The only restriction is that the agent must own at least a function named `execute()`, which receives the agent itself as argument and describes its behavior.

Take as example Code 1. In this simple script, we create a single foo agent with two properties, *size* and *name*, plus an execution function that increases its size and performs a random walk. It is necessary to call the constructor `Agent()` with foo agent to provide it basic structures such as `Agent::move()` and `Agent::getCell()`, which are used in the example. A random walk then takes place moving foo agent to a random neighbor of its current cell.

```
singleFooAgent = {size = 10, name = "foo"}
singleFooAgent.execute = function(self)
    self.size = self.size + 1
    self:move(self:getCell():getNeighborhood():sample())
end
singleFooAgent = Agent(singleFooAgent)
```

Code 1: A simple agent.

Before performing any action, foo agent needs to belong to a cell. Let us put it in a random cell of a 10x10 cellular space with Moore neighborhood. To accomplish that, we need to create an environment with the two entities that will be coupled: foo agent and the cellular space. Then, `environment::createPlacement()` can put every agent within the environment in random cells, as shown in Code 2. In the end, we create an event to activate foo agent ten times.

```
cs = CellularSpace {
    xdim = 10,
    ydim = 10
}

cs:createNeighborhood{strategy = "moore"}

e = Environment {
    cs,
    singleFooAgent
}

e:createPlacement{strategy="random"}

t = Timer {
    Event{target = singleFooAgent}
}

t:execute(10)
```

Code 2: Placement of a simple agent.

Creating agents as presented above is simple, but it may be contraproductive in the case where one needs to have even a small set of agents acting and interacting with each other. The entity that represents a collection of agents with the same set of properties and temporal resolution is called *society*. The constructor of a society always requires a table with the basic properties and an `execute()` to describe the general behavior of each instance to be created. This table can also have a `build()`, which is called only when the agent is created, useful when one wants to generate properties that have heterogeneous values within the society.

The simplest way to create a society and its agents is by using a description of an instance and a quantity of agents to be created. Code 3 describes a non-foo society with 50 agents that look for a foo agent in the cellular space they belong. In this example, every agent will start with name “nonfoo” and a random age from a uniform random integer in the range [1, 10].

```
nonFooAgent = {
  name = "nonfoo",
  build = function(self)
    self.age = math.random(10)
  end,
  execute = function(self)
    self.age = self.age + 1
    self:move(self:getCell():getNeighborhood():sample())
    forEachAgent(self:getCell(), function(agent)
      if agent.name == "foo" then
        print("Found a foo agent")
      end
    end)
  end
}

nonFooSociety = Society {
  instance = nonFooAgent,
  quantity = 50
}
```

Code 3: Basic constructor for a society.

The same procedure of the last example can be used to distribute the society over space, as shown in Code 4. A call to `Environment::createPlacement()` puts every agent within the environment in the cellular space, including every agent of non-foo society and foo agent. The only difference of the new code is that the allocation needs to have at most one agent within any cell (max = 1), what could not happen in the last example.

```
env = Environment { nonFooSociety, cs, singleFooAgent }

env:createPlacement{strategy = "random", max = 1}

t = Timer {
  Event {target = nonFooSociety},
  Event {target = singleFooAgent}
}

t:execute(10)
```

Code 4: A society within an environment.

Groups

Groups are ordered subsets of a society. They are created in the same way of trajectories, except by the fact that they use societies as target instead of cellular spaces. A *filter* function returns whether an agent will belong to the group, while a *sort* defines the group's traversing order. Groups inherit all functions of a society, allowing them to call `Society::execute()` directly.

```
biggers = Group {  
  target = society,  
  filter = function(agent)  
    return agentcell.size > 10  
  end,  
  sort = function(a1, a2)  
    return a1.size > a2.size  
  end  
}  
  
biggers:execute()
```

Code 5: Creating groups.

Every time a society is activated, each of its agents is executed in the same order they were pushed into the society. Because of that, using `Society::execute()` directly from the society is only recommended when the order of execution makes no difference in the simulation results. Whenever it may affect the results, it is better to use groups instead of societies. As parameter `select` is optional, it is possible to create a group covering the whole society and establish a new traversing order according to some rule. `Sort` is also optional, which is useful when one wants to create subsets of a society where the execution order makes no difference, or when the group needs to be executed randomly by calling `Group::randomize()`, as shown in Code 6.

```
males = Group {  
  target = society,  
  select = function(agent)  
    return agent.sex == "male"  
  end  
}  
  
males:randomize()  
males:execute()
```

Code 6: Groups randomly ordered.

Life span

Mainly in models that do not use real world data, agents may have a life span. TerraME provides functions `dye()` and `reproduce()` deal with life span. `Dye()` removes the agent from the society as well as its placement relations. It is recommended that the agent dies at the end of its execution because it does not stop the agent immediately. The other function, `reproduce()`, creates a new agent similar to the original agent, with the same placement relations, and puts it into the same society of its parent.

Code 7 shows part of the source code of the well-known predator-prey model. In this model, there are two types of agents, predators and preys, which belong to two different societies. Preys feed grass, while predators feed preys. Whenever one of these agents reaches 50 of energy, it reproduces. On the other side, if someone's energy ends up, it dies. In this code, both have a common set of actions that are implemented within a single function.

```

commonActions = function(ag)
    ag.energy = ag.energy - 1
    ag:move(ag:getCell():getNeighborhood():sample())
    if ag.energy >= 50 then
        ag.energy = ag.energy/2
        ag:reproduce()
    end
    if ag.energy <= 0 then
        ag:dye()
    end
end

predator = {
    energy = 40, type = "wolf",
    execute = function(ag)
        forEachAgent(ag:getCell(), function(other)
            if other.type == "sheep" then
                ag.energy = ag.energy + other.energy / 2
                other:dye()
                return false
            end
        end)
        commonActions(ag)
    end
}

prey = {
    energy = 40, type = "sheep",
    execute = function(ag)
        if ag:getCell().cover == "pasture" then
            ag:getCell().cover = "soil"
            ag.energy = ag.energy + 5
        end
        commonActions(ag)
    end
}

```

Code 7: Behaviour for agents of a predator-prey model.

There is no problem on calling `dye()` to agents in other societies, as in the predator-prey example. However, it is not recommended to kill an agent that is scheduled to execute in the same Event of the current execution of an agent (e.g. when they share the same society).

Messaging

TerraME has a very simple environment for exchanging messages between agents. It uses Lua facilities to describe the content of a message, storing them as tables. Function `Agent::message()` is the way an agent can exchange information with other agents. The only compulsory argument of the table is the *receiver*, the

other ones depend on the model and can be used freely by the modeler. Code 8 describes a simple message, where an agent sends a message for its fellow with 100 units of money.

```
agent:message{receiver = myFellow, content = "money", value = 100}
```

Code 8: A simple message.

When a message is sent, the receiver gets it through an internal function called `onMessage()`, which must be implemented by the modeler in order to correctly follow up the communication. Usually, `onMessage` functions are implemented in the constructor of the agent, allowing all the agents within the same society to have the same general behavior. If `onMessage()` is not implemented for an agent that needs to receive a message then a warning will be generated by the simulation. Code 9 shows an example of receiving a message.

```
function agent.onMessage(self, message)
  if message.content == "money" then
    self.money = self.money + message.value
  end
end
```

Code 9: Receiving a message.

The receiver can identify who sent the message through the element *sender*. It may answer a message in two ways. If it sends normally using `Agent::message()`, the original sender will get the message in its own `onMessage()`, interpret it, and then continue its execution at the point where the first message was sent. Another option is to send a message through the returning value of `onMessage()`, as shown in Code 10. In this case, the sender will receive the answer as a result of its `Agent::message()` call, avoiding large stacks of messages when the communication involves exchanging lots of messages.

```
function agent.onMessage(self, message)
  if message.content == "money" then
    self.money = self.money + message.value
    return{content="greetings"}
  end
end
```

Code 10: Receiving a message.

The messages presented so far require an immediate response of the receiver for the agent to continue its behavior. This type of message is called *synchronous*. Another option to exchange messages in TerraME is by using *asynchronous* messages. Messages sent asynchronously go to a pool within the society the agent belongs and stay there until some `Society:synchronize()` call. These messages have a temporal delay that indicates the number of synchronization steps required for each message to be finally delivered. Code 11 shows an example of asynchronous messages.

```
agent:message{receiver = john, delay = 1, content = "greetings"}
agent:message{receiver = john, delay = 3, content = "farewell"}

society:snchronize() -- greetings
society:snchronize()
society:snchronize() -- farewell
```

Code 11: Asynchronous messages.

Messages sent asynchronously belong to the society until delivered, but they are not connected to the simulation time by default. To link them, it is possible to put a society as the target of an event, as described in Code 12. Every time this event is activated, the society synchronizes its messages after executing its agents.

```
t = Timer {
  e = Event {target = soc} -- soc:execute() then soc:synchronize()
}
```

Code 12: Creating an event targeting a society.

Social Networks

Usually agents exchange messages with someone that it has some kind of connection. TerraME has a class called *social network* to work with connections between agents. A social network is a collection of agents that preferably belong to the same society. Code 13 creates a social network with two agents, *john* and *mary*, and puts it within *myself*. Second-order function `forEachConnection()` allows one to traverse the social network.

```
friends = SocialNetwork()

friends:add(john)
friends:add(mary)

myself:addSocialNetwork(friends)

print(myself:getSocialNetwork():size())

forEachConnection(myself, function(self, friend)
  print(friend.name)
end)
```

Code 13: Simple social network.

Each agent can have one or more social networks attached to it. They are indexed as strings, having an implicit value of “1” when not specified in a function call. Therefore, whenever the modeler wants to have more than one social network per agent, it is necessary to use explicit indexes. Every function of TerraME that uses social networks has an optional argument to describe which index will be used, such as the second parameter of `Agent::addSocialNetwork()`, as presented in Code 14. Note that `forEachConnection` has a second construction that takes three arguments instead of two, shifting the function to third argument in order to add the index as second value.

```
family = SocialNetwork()
-- ...
myself:addSocialNetwork(family, "family")
forEachConnection(myself, "family", function(self, familyMember)
  -- ...
end)
```

Code 14: Social networks with explicit index.

To facilitate creating social networks, TerraME provides high-level functions available in societies and environments. Function `Society::createSocialNetwork()` has some strategies to create networks for each agent within the society. Code 15

presents a code that creates a social network where each agent has exactly five connections.

```
soc:createSocialNetwork{quantity = 5, name = "veryfriendones"}

ag = soc:sample()

forEachConnection(ag, "veryfriendones", function(ag, other)
  ag:message{receiver = other, content = "imalive"}
end)
```

Code 15: Creating a social network within a society.

When it is required to build a social network between agents of different societies, it is necessary to put them into an environment. Code 16 creates relations between two societies, connecting five professors to each student and five students for each professor. Using symmetric option indicates that if a student is connected to a given professor then the professor will also be connected to the student, and vice-versa.

```
e = Environment{professors, students}

e:createSocialNetwork{quantity = 5, symmetric = true}
```

Code 16: Creating a bidirected social network.

Database Access

Societies can be retrieved from external sources, instead of being built from scratch along the simulation. Any data set can be materialized as a Society, in such a way that each entity of the given set generates the attributes of a single agent. This construction of a society requires the same parameters for building a cellular space, requiring only the additional argument *instance* to describe the overall behavior of its agents. Code 17 shows an example of loading a society from a database.

```
soc = Society {
  instance = nonFooAgent,
  database = "amazonia",
  dbms = "mysql",
  user = "andrade",
  layer = "farmers"
}

soc:execute()
```

Code 17: Creating a society from a database.

TerraME considers that, instead of agents having to query a geographic database whenever they need an answer about a specific spatial structure, the representation of space within the model is already filled with this data. It represents a suitability map, where the neighborhoods between cells represent criteria such as visibility or possible movements. We assume the modeller previously knows the queries the agents may perform along the simulation. It can be considered a limitation of this approach, but it is a way to consider the problem of using geospatial data, with the advantage of separating GIS functions

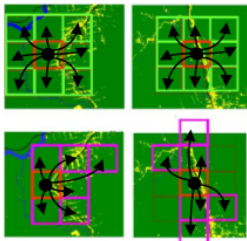
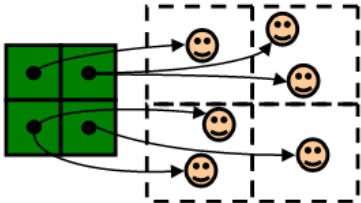
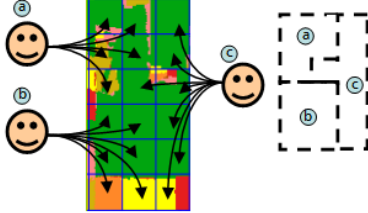
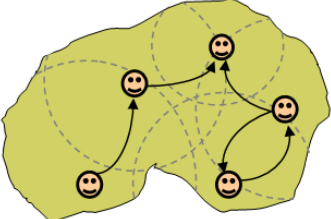
from the simulation. The idea is that both applications can work harmonically but separately, sharing only geospatial data.

Filling the whole space with the results of the queries commonly requires more time than performing a couple simulations. However, once the data is already in the database, the simulation runs faster because there is no need to maintain a connection to a geographic database to perform the same queries repeatedly. It can also be considered an advantage, once repeating simulations is a common procedure for studying the overall behaviour of a model.

In the case where the external source is a geospatial database, the relations within and between societies as well as placement relations can also be retrieved. To accomplish that, the modeller needs to execute the necessary algorithms to create the graph with the specific operations and save the results. Examples of operations related to each of the four possible types of relations* are depicted in Table 1. They are:

- A layer of polygons representing farms can be connected to a layer of squared cells to represent the minimum spatial partitions where the agent, a farmer, can take its land change decisions. Each farm can be connected to the cells whose overlay is more than half the cell.
- A set of cells can be connected to a set of points, representing the locations of human beings, according to the result of the within predicate.
- A cellular space can have its cells connected to those within a given traveling time radius, considering different velocities of each road.
- Factories represented as points can be connected to those within a given distance radius that depends on some property of the agents.

Table 1: Relations as graphs.

To From	Cell	Agent
Cell		
Agent		

* According to Torrens and Benenson (2005), the two basic types in models that simulate geographic processes, cells and agents, define four types of relations: cell→cell, agent→cell, cell→agent, and agent→agent.

Reading relations that involve a single society can be performed directly through `Society::loadSocialNetwork()`. When reading from a society loaded from a database, it is possible to load the relations simply by passing the id of the GPM in the database, as shown in Code 18. Loading relations from a database is safer because the responsibility of verifying whether the correct collection of objects is being used is in charge of TerraME. Otherwise, it will be up to the modeler.

```
soc:loadSocialNetwork{file = "myfile.gal"}
```

Code 18: Loading social networks within a society from a database.

To create relations involving a society and another set, be it a society or a cellular space, it is necessary to instantiate an environment previously. Two functions can then be used to retrieve the relations: `Environment::loadPlacement` and `Environment::loadSocialNetwork`. The first one establishes relations between a society and a cellular space. The latter function loads relations between two societies. The same rules are applied to manipulate the relations created from these two functions and from scratch. Code 19 shows an example of using both.

```
env = Environment {  
  nonFooSociety,  
  cs  
}  
  
env:loadPlacement{file = }  
env:loadSocialNetwork{name="friends"}  
  
forEachCell(cs, function(cell)  
  forEachAgent(cell, function(agent)  
    forEachConnection(agent, function(agent, friend)  
      -- ...  
    end)  
  end)  
end)
```

Code 19: Loading social networks between societies and placements from a database.

Multiple Placement

Placement is usually related to the physical location of an agent. However, it is possible that an agent needs to be connected to cells by multiple reasons. One can own one or more cells or have them as a target for something, for example. The basic way of using placement functions in TerraME allows the modeler to work with one type of placement, but each function that deals with placement has an optional argument that names the relation to be used. For example, Code 20 describes how to move to new cells in two different placements.

```
agent:move(onecell, "renting")  
agent:move(anothercell, "workplace")
```

Code 20: Moving with multiple placements.

However, before using multiple placements, they need to be instantiated. Societies, cellular spaces, and environments can accomplish that in different ways. When a placement is created from an environment, it needs to contain a society and a cellular space and creates bidirected placements. Using cellular spaces or societies it is only possible to create one-sided relations. Code 21 describes one example that creates two placements: a one-sided renting relation is created without any value (void), while a bidirected workplace is filled with one random placement for each element.

```
society:createPlacement{strategy="void", name = "renting"}

env = Environment {cellspace, society}

env:createPlacement{strategy="random", name = "workplace"}
```

Code 21: Creating new placements with different indexes.

Internally, TerraME uses groups and trajectories to store these relations. Each placement has an index, which must be a name different from any other attribute of the agents *and* cells it is going to be related. As default, placement relations are stored in a variable called “placement”, with object “agent.cells” (“cell.agents”) being a pointer to “agent.placement.cells” (“cell.placement.agents”) to allow using `forEach` functions directly. Other placement relations do not have this facility, requiring to be traversed manually, as presented in Code 22.

```
myPlacementFunction = function(cell)
  -- ...
end()

forEachCell(agent, myPlacementFunction)
forEachCell(agent.placement, myPlacementFunction) -- same as before
forEachCell(agent.renting, myPlacementFunction)
```

Code 22: Traversing different placements.

As placement relations are stored as groups and trajectories, the modeller can manipulate them directly as well as use the three basic placement functions (enter, leave, and move). Code 23 shows examples of both strategies, which have a small but important difference. Strategy (a) adds a new cell to the agent, while (b) does the same and, furthermore, adds the agent to the new cell. Because of that, (a) is recommended for one-sided relations, while (b) can only be used to handle symmetric relations. In this sense, Code 21 cannot be used to create the relations manipulated by Code 20 because renting is an one-sided relation.

```
forEachAgent(society, function(agent)
  for i = 1, 10 do
    agent.workplace:add(cellspace:sample()) -- (a)
    agent:enter(cellspace:sample(), "workplace") -- (b)
  end
end
```

Code 23: Changing an indexed placement.

Indirect Relations

The common way to store relations in TerraME is by using a direct connection between objects. Although simple, this way of working with relations has two limitations related to computational efficiency:

1. There may exist relations that depend upon other relations, also called *indirect* (Torrens and Benenson, 2005). With direct connections, such relations must be recomputed every time step, even if they are not used by any entity of the model.
2. Storing all relations explicitly is memory consuming, as they usually grow faster than the number of agents in the model. Sometimes it is preferable to spend more processing time to save memory.

To overcome this hurdle, TerraME allows the modeller to create relations which are computed dynamically and do not require memory to be stored in a specific variable along the execution of the model. Take for instance a social network where a given agent is connected to each other agent that belong to the neighbor cells of the agent's current cell. It is never efficient to store this relation explicitly as long as the agents relocate frequently. Code 24 describes how to create an indirect relation to compute such relation. Function `Agent:addSocialNetwork()` can get social networks or functions that return social networks as argument. Although they are quite different from static relations, once the criteria that creates them is established, they can be used transparently by the modeler, as if they were static relations, for instance to execute a `forEachConnection()`.

```
runfunction = function(agent)
  local rs = SocialNetwork()
  forEachNeighbor(agent:getCell(), function(_, neigh)
    forEachAgent(neigh, function(agentwithin)
      rs:add(agentwithin)
    end)
  end)
  return rs
end

agent:addSocialNetwork(runfunction, "neighborhood")

forEachConnection(agent, "neighborhood", function(agent, other)
  -- ...
end)
```

Code 24: Indirect relation through neighborhood.

Some strategies to generate indirect relations are available in TerraME by means of function `Society::createSocialNetwork()`. For example, Code 25 describes how to apply the indirect relation of Code 24 to a whole society using a moore neighborhood. Note that this neighborhood needs to be generated by the modeler in the cellular space where the agents have placement relations to ensure that indirect relations will work properly.

```
soc:createSocialNetwork {  
    neighbor = "moore"  
    name = "byneighbor"  
}  
  
agent = soc:sample()  
  
forEachConnection(agent, "byneighbor", function(agent, other)  
    -- ...  
end)
```

Code 25: Creating indirect relations from a society.

References

P. R. Andrade and T. G. S. Carneiro (2011). TerraME Classes and Functions. Available at www.terrame.org.

N. Gilbert (2007). Agent-Based Models (Quantitative Applications in the Social Sciences). 153. SAGE Publications.

N. Gilbert and K. G. Troitzsch (2005). Simulation for the Social Scientist. Open University Press.

P. Torrens and I. Benenson (2005). Geographical automata systems. International Journal of Geographical Information Science (IJGIS), v. 19, n. 4, p. 385–412.