This is a post-print version of the paper that was accepted in "Enviromental Modelling and Software", 2013 (DOI: 10.1016/j.envsoft.2013.03.002).

TerraME: an extensible toolbox for modelling nature-society interactions

Tiago Garcia de Senna Carneiro[1], Pedro Ribeiro de Andrade[2], Gilberto Câmara[3], Antônio Miguel Vieira Monteiro[3], Rodrigo Reis Pereira[1]

[1]Earth System Simulation Laboratory (TerraLAB), Federal University of Ouro Preto (UFOP), Campus Universitário, Morro do Cruzeiro, Ouro Preto, MG, 35900-000, Brazil

[2]Earth System Science Center (CCST), National Institute for Space Research (INPE), Av. dos Astronautas, 1758, São José dos Campos, SP, 12227-001, Brazil

[3]Image Processing Division (DPI), National Institute for Space Research (INPE), Av. dos Astronautas, 1758, São José dos Campos, SP, 12227-001, Brazil

Abstract

Modeling interactions between social and natural systems is a hard task. It involves collecting data, building up a conceptual approach, implementing, calibrating, simulating, validating, and possibly repeating these steps again and again. There are different conceptual approaches proposed in the literature to tackle this problem. However, for complex problems it is better to combine different approaches, giving rise to a need for flexible and extensible frameworks for modeling nature-society interactions. In this paper we present TerraME, an open source toolbox that supports multi-paradigm and multi-scale modeling of coupled human-environmental systems. It enables models that combine agent-based, cellular automata, system dynamics, and discrete event simulation paradigms. TerraME has a GIS interface for managing real-world geospatial data and uses Lua, an expressive scripting language.

[1] Corresponding author. Tel: +55 (31) 3559-1692; fax: +55 (31) 3559-1660.

E-mail addresses: tiago@iceb.ufop.br, pedro.andrade@inpe.br, gilberto.camara@inpe.br, miguel@dpi.inpe.br, rreisp@gmail.com.

Keywords

Nature-society models, multi-scale modeling, environmental modeling, discrete event simulation, cellular automata, multi-agent systems.

Software availability

Name: TerraME

Developer:

Federal University of Ouro Preto (UFOP), Brazil

National Institute for Space Research (INPE), Brazil

Contact: tiago@iceb.ufop.br, pedro.andrade@inpe.br

Programming language: Lua

Optional additional software: MySQL and TerraView

License: GNU LGPL (open source)

Website: http://www.terrame.org

1 Introduction

Planners and policy makers need models that capture how human actions act on natural systems (Turner et al., 1995). These models represent coupled nature-society systems in different ways. Their capacity to capture the impact of human actions in nature depends on the spatial and temporal scales used. It also hinges on the chosen hypotheses about human behavior and environmental response. Despite the challenges involved in building them, these models have an important role. They bring forth unstated assumptions hidden in policy proposals, helping us to understand the possible results of different choices (Moran, 2010).

In this paper, we use the term *paradigm* to mean a worldview intrinsic to a scientific theory. Models of nature-society interactions use different paradigms, including cellular automata, agent-based models, map algebra, and system dynamics (White and Engelen, 1997; Parker et al., 2003; Karssenberg and De Jong, 2005; Batty, 2012). In many cases using a single paradigm is not enough. For complex problems, it is better to combine different methods to learn more about how human societies interact with nature (Rindfuss et al., 2004).

Most designers of nature-society modeling tools choose a paradigm and build a toolbox that supports it. Supporting a single paradigm has many advantages. Most paradigms have a lot of documentation and user communities, which helps potential adopters. However, designer choices may also limit a software's ability to grow. Tool designers have to choose a programming environment, user interfaces, data types and their relations, algorithms, data handling, and storage. A design suited for one paradigm may not be adequate to support others. Although multi-paradigm modeling tools can in theory combine different ways of modeling, building such tools is a hard task. This begs the question: *"What kinds of software architecture are better suited for multi-paradigm modeling of nature-society interactions?*" In what follows, we refer to this challenge as the *multi-paradigm model design problem*.

This paper presents a possible response to this question. We were inspired by how Bjarne Stroustrup built C++ (Stroustrup, 1994). He designed C++ in a bottom-up, modular fashion, allowing object-oriented, generic programming, and procedural programming styles. The flexibility of C++ has no doubt contributed to its

widespread use. Following these ideas, our proposed solution for the multi-paradigm model design problem stems from three conjectures. First, the tool should provide a *collection of data types and functions* needed by different paradigms. This leads to a bottom-up design based on building blocks that are combined by the modeller. The second conjecture is that *nature-society interactions happen in geographical space*. Unlike human and capital resources, that are mobile, natural resources are fixed. When dealing with environmental problems, we have to capture geographical features such as soil, climate, vegetation, and biodiversity in a spatially explicit way. Thus, models for nature-society interactions need a spatial component that represents natural landscapes and the results of human interactions with them. Third, *nature-society interactions occur at different scales*. Many problems need to be expressed as multi-scale models where matter, energy, and information flow between different scales. The toolkit should allow the user to break a complex model into simpler sub-models. Each sub-model is a micro-world with its own temporal and spatial resolution and behaviour. Sub-models can then be nested and combined in different ways. Thus, our proposed architecture puts together a *set of data types* with *methods to build and connect geospatial micro-worlds*.

Based on these conjectures, we have designed and implemented the TerraME toolbox. It has building blocks for model development, allowing the user to specify the spatial, temporal, and behavioral parts of a model independently. Its components are expressive, enabling different approaches to be combined. TerraME's main aim is flexibility. It does not enforce a unique modeling paradigm, but provides the tools needed by the modeller. TerraME is an open source software distributed under the GNU LGPL license and is available at www.terrame.org.

In the next section, we consider the challenges for designing software to model nature-society interactions, pointing out the choices we made. We describe the general architecture of TerraME in section 3. Section 4 has examples that show the main features of TerraME. We finish the paper by reflecting on the contributions and the limits of our proposed solution to the multi-paradigm model design problem.

2 Design choices for nature-society interaction modeling toolboxes

In this section, we discuss four decisions faced by designers of modeling tools that support nature-society interactions. In each case, we point out the choices we made in TerraME.

- Choosing which modeling paradigms to support.
- Selecting the model interface.
- Defining how the model interfaces with databases and GIS.
- Providing tools for verification, calibration, and validation.

2.1 Choice of modeling paradigms

Nature-society modeling paradigms include *Cellular Automata* (von Neumann, 1966), *System Dynamics* (Forrester, 1961), *Agent Based-Systems* (Wooldridge and Jennings, 1995), *Map Algebra* (Tomlin, 1990), and *Discrete Event System Specification* (Zeigler et al., 2005). Cellular automata (CA) are finite machines organized in a lattice connected by neighborhood relations. CAs can produce complex patterns from simple rules. In the system dynamics view, the world consists of stocks of energy, information, or matter. Model rules are differential equations defining flows that transport energy, information or matter between stocks. Agent-based models represent autonomous individuals that interact with themselves, the environment, and other agents. Map algebra uses raster maps to allocate properties in space and provides functions over maps to convey change. In the discrete event formalism, an event is an individual temporal episode. Instead of having functions that compute the next step of the simulation, an event-based model has a set of events and conditions when they occur.

Most existing modeling tools are centered on a paradigm, although they may support others. Examples of agent-based modeling tools are NetLogo (Tisue and Wilensky, 2004) and RePast (North et al., 2006). System modeling tools include STELLA (Roberts et al., 1983), Vensim (Eberlein and Peterson, 1992), and Simile (Muetzelfeldt and Massheder, 2003). PCRaster is a map algebra toolbox with extensions for dynamic modeling (Karssenberg et al., 2001; Karssenberg et al., 2009;

Wesseling et al., 1996). JDEVS is an event-based modeling software (Filippi and Bisgambiglia, 2004). Focusing in a paradigm favors knowledge reuse. Users familiar with one modeling paradigm will be comfortable when facing a new toolbox based on similar ideas. If one knows STELLA, learning Vensim and Simile is straightforward. Models developed in NetLogo can be ported to RePast without excessive work (Crooks and Castle, 2012). Designers can also extend an existing tool to support other paradigms than their original choice.

The alternative is to build a multi-paradigm modeling tool in a bottom-up way. This is what we did in TerraME since we hold that nature-society relations are inherently complex. As expressed by Mike Batty: "*in modeling, the quest for parsimony, simplicity, and homogeneity is increasingly being confronted by the need for plausibility, richness, and heterogeneity"* (Batty, 2012). A multi-paradigm toolbox allows modellers to combine different paradigms when solving a problem. However, such tools are harder to learn since there are many concepts to be grasped. Flexibility comes at a price. We recognize that not all users will be willing to make it, although we believe the effort is worthwhile.

2.2 Selecting the model interface

Modeling toolboxes need to provide analytical power to express complex problems. Nearly all tools use a programming language with additional high-level statements. Some tools also provide icon-based graphical programming, like the *system dynamics* tools STELLA and Simile. Visual interfaces are appealing and enable decision-makers to quickly grasp model behavior. However, it is not easy to express spatial variation using icons. Thus, most spatially-based tools use a programming language as their main interface.

In TerraME, we chose a programming language interface. To support rapid model implementation we chose Lua, an open-source interpreted language with extensible semantics (Ierusalimschy et al., 1996). The modeller uses a clear and expressive language that calls demanding operations in C++, hidden from him. This provides a good trade-off between source code directness and computational efficiency.

2.3 Interfaces with databases and GIS

Nature-society models need to work with geospatial data for real-world applications. Many tools use flat files to store model input and output. However, databases are more suitable than flat files to store these datasets because they provide consistency, durability, and sharing (Gray, 1981). Using a database also helps the user to organize data. The modeller relies on the same database to do exploratory analysis, run the simulation, and examine the results. Most recent GIS (geographical information systems) have interfaces to databases to provide spatial data access and storage. By linking with a GIS, modeling tools inherit its capacity for data handling. Among the toolboxes that provide integration with a GIS are NetLogo, RePast, Simile, and PCRaster.

In TerraME, we chose the TerraLib open source geospatial library (Câmara et al., 2008) to serve as its GIS and database interface. TerraLib supports open source database management systems such as MySQL and PostgreSQL and its vector data model is compatible with OGC (Open Geospatial Consortium) standards. The library has functions to read data in different formats and convert them into regular or irregular cellular spaces. It also ensures persistent storage and retrieval of modeling data. It also has tools for viewing data such as TerraView (Câmara et al., 2008). The downside is that adopters of TerraME will also have to use the TerraLib support for geospatial databases. Considering the growing acceptance of open source GIS tools (Steiniger and Bocher, 2009), we believe this is a manageable risk.

2.4 Tools for verification, calibration and validation

The model building steps include conception, structuring, calibration, verification, and validation (Jakeman et al., 2006). Toolboxes should provide services and tools to support its users in all these stages. Faulty results are hard to spot when shown as numbers. Users find and fix conceptual and implementation mistakes more efficiently if real-time visualization interfaces are available during simulations. In TerraME, as in similar tools, we provide a real-time visualization interface of simulation outputs.

Nature-society models need to be calibrated with spatially explicit data. There is a considerable body of recent research concerning data assimilation and calibration (Beven and Binley, 1992; Janssen and Heuberger, 1995; Lin and Beck, 2012). Stochastic data assimilation methods allow models to update their initial conditions as new input data becomes available. Applications such as PCRaster have developed sophisticated calibration tools that can be used in hydrology, crop growth, and air pollution (Karssenberg et al., 2009; Verstegen et al., 2012). In TerraME, we chose calibration tools that use aggregated values and spatial explicit model validation methods, such as those proposed by Costanza (1989) and Pontius Jr and Millones (2011).

3 TerraME: Terra Modeling Environment

3.1 System conception and architecture

The TerraME architecture is shown in Figure 1. Its lowest tier uses the TerraLib C++ library (Câmara et al., 2008). The second tier provides support for modeling in C++ including agent-based, cell-space, systems-oriented and event-based paradigms. The third tier is the interface between TerraME and Lua. It adds data types and functions for model simulation and evaluation to Lua. Other mathematical and statistical libraries can have their APIs exported to the Lua interpreter. The next tier is the Lua interpreter, which takes model source code as input and executes the simulation. The last tier consists of end user models. The top of Figure 1 shows four examples of models that can be implemented using TerraME.

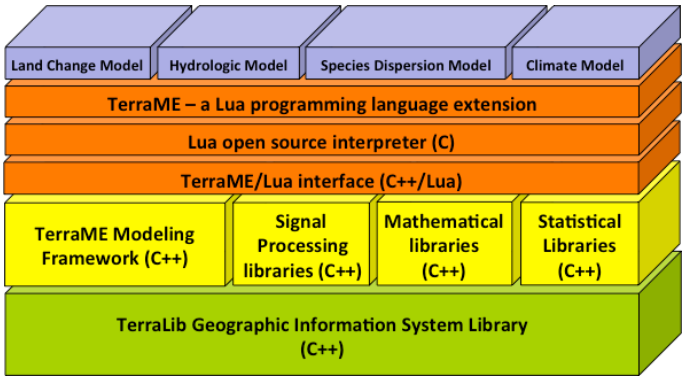

Figure 1: TerraME architecture

TerraME considers that a model has *spatial*, *temporal*, and *behavioral* dimensions. The spatial dimension deals with the geographical area under study and the spatial resolution used for data sampling. The behavioral dimension refers to the rules (for example, agent behavior) and to the indirect techniques (for example, statistical methods) that represent change. The temporal dimension includes the period considered by the model and the frequency when change occurs. To define a model, the user sets up instances of TerraME's spatial, behavioral, and temporal types, which are described below.

3.2 Spatial types

TerraME provides four spatial types: *Cell*, *CellularSpace*, *Neighborhood, and Trajectory*. A cell is a spatial location which has persistent and runtime attributes. Persistent attributes are stored in geospatial databases, while runtime values exist only during the simulation. A cellular space is a set of cells representing a geographical area divided in regular or irregular partitions. Cellular spaces can be saved and recovered from TerraLib databases. Each entity of a geospatial database (cell, pixel, point, line, or polygon) is loaded as a cell in TerraME. Figure 2 shows a database with three different layers: (1) a set of roads represented as lines, (2) Brazilian states within Amazonia represented as polygons, and (3) 25x25km cells composing a sparse grid representing protected areas in Amazonia. Each of them can be read into a cellular space.

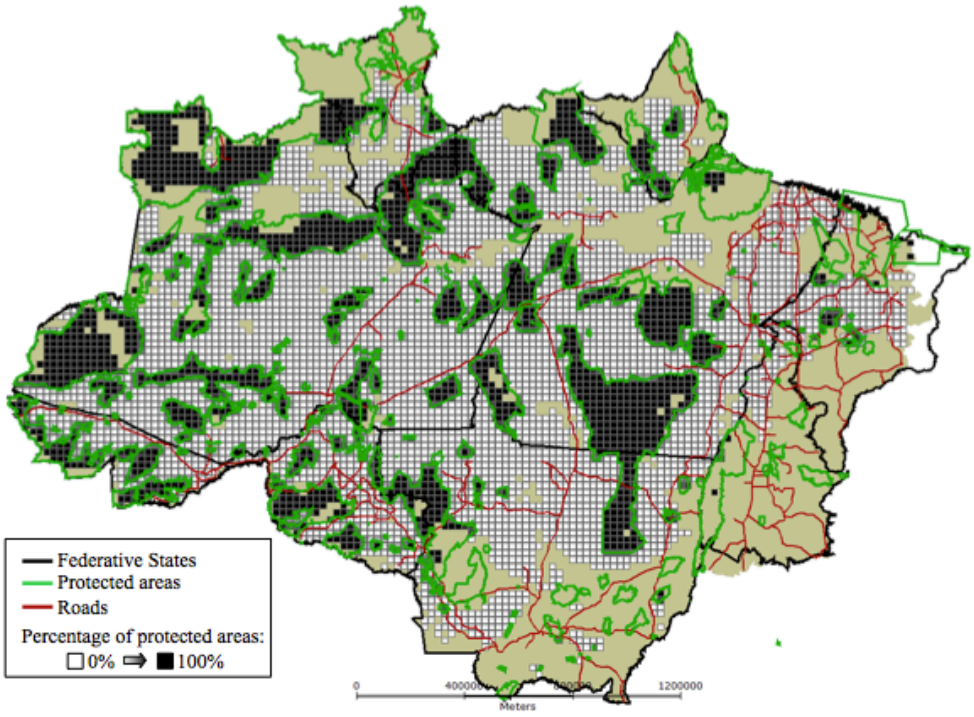

Figure 2: Squared cells representing a cellular space for Brazilian Amazonia.

The third spatial type is *Neighborhood*, a topological representation of proximity relations. A neighborhood is a set of pairs (c, w), where c is a neighbor cell and w is the weight of the relation. Neighborhoods connect cells inside the same cellular space or between spaces. Each cell can have more than one neighborhood. TerraME has functions to create simple neighborhoods such as Moore and von Neumann. Complex spatial relations use a generalized proximity matrix (GPM). A GPM is a directed graph whose weights express relations between geographic objects (Aguiar, 2006) that can be loaded from a TerraLib database during simulations. TerraME does not work with vector geometries explicitly as most operations over such geometries are computationally intensive tasks. This is a limitation, but it has the advantage of not computing spatial operations repeatedly during simulations, which reduces computational cost. Figure 3 shows different types of neighborhoods. Upper tiles show Moore neighborhoods. The lower ones depict neighbors built from roads using a GPM.

TerraME supports any algorithm that uses a Euclidean representation of space. During simulations, it is possible to compute raster-based operations using the (x, y) positions of cells. Neighborhood relations from exogenous vector-based data, such as connectivity to markets through roads, can change by loading GPMs registered for different simulation times. Once relations are already stored in files, loading them in different executions of the model reduces simulation time because they do not need to be computed repeatedly.

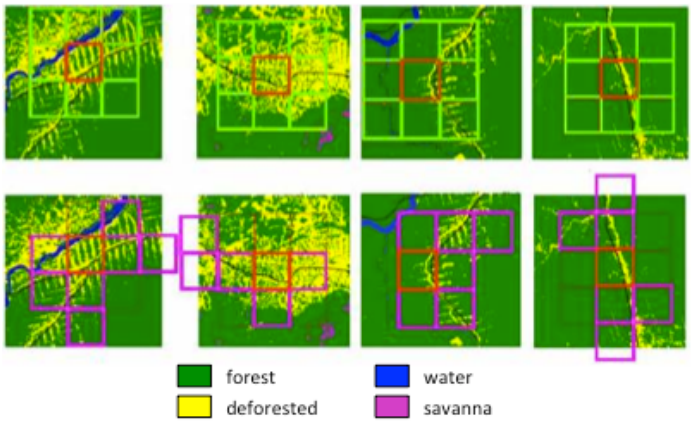

Figure 3: Different types of neighborhoods.

The fourth spatial type, *Trajectory,* allows the user to define how to go through a cellular space. A trajectory is an iterator that selects a subset of a cellular space and defines an order for traversing this subspace. Defining trajectories is especially useful for allocating change in space. For example, consider a land change model where the user is interested in modelling the transition from forest to agriculture. The modeller can define a trajectory by selecting all cells representing forest and ordering them by their potential for change. Cells with higher potential can then be traversed first.

3.3 Behavioral types

To describe model behavior, TerraME has two types: *Agent and Automaton.* Agents are uniquely identifiable individuals situated in space. They can represent actors, institutions, or even whole systems. Each agent has a state, can move over cellular spaces, and can communicate with other agents. TerraME provides functionalities to agents such as synchronous and asynchronous messages, connections to cells and

other agents, and life span. For model development, agents can be grouped in a *Society*. A society is a collection of agents with the same set of properties and temporal resolution. Societies can be created from scratch or retrieved from geospatial databases during the simulation. An agent is related to a society as a cell is to a cellular space.

An *automaton* is a spatial process that has independent states at each location. While an agent acts globally in the cellular space, the automaton acts locally. A single agent with a unique internal state can control several cells. An automaton has many instances that share the same set of states and attributes, but change independently from each other. At a given time, each instance of an automaton can be in a different state and have different attribute values.

TerraME supports both agents and automata because of the different needs of nature-society modeling. Societal models need agents that can move freely in space and interact with other agents. By contrast, many natural models (such as hydrological ones) need local variations of global laws. The physical laws are the same, but the local behavior is constrained by natural variations. Thus, the automaton type is better suited for modeling natural processes.

3.4 Temporal types

Once spatial structures and behavioral rules are described, it is necessary to define temporal structures. TerraME has two temporal types: *Event* and *Timer*. An event is a time instant when the simulation engine executes operations. A timer is a clock that registers a continuous simulation time. It manages an event queue ordered according to their priorities and timestamps. Figure 4 shows how event scheduling works in TerraME. It contains a timer with a queue of four events. As each event is removed from the head of the queue, the timer's clock is updated with its timestamp. After that, the event's action is executed and the event may be deleted or requeued according to its result.

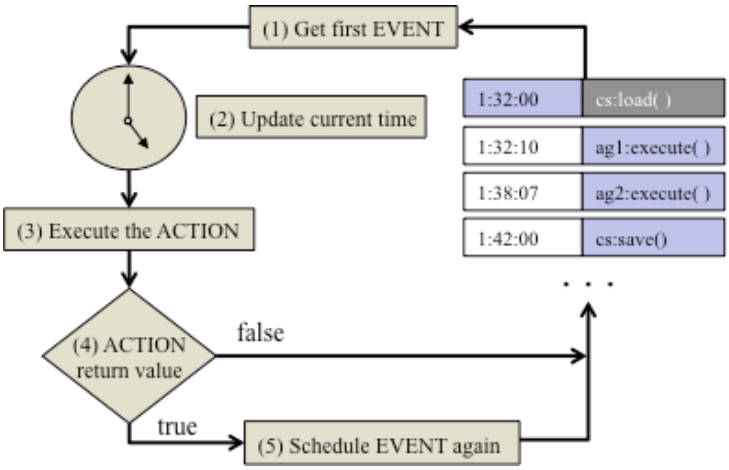

Figure 4: Timer and event, the temporal types of TerraME.

3.5 The Environment type

In TerraME, the *Environment* type allows the user to set up multi-scale models. An *Environment* represents a micro-world containing data and commands to be executed. It includes the spatial, behavioral, and temporal parts of a model. Environments can be nested, supporting multi-scale models. Thus, combining different environments, users can build complex models.

When developing multi-scale models, the user first defines one environment for each model. Then, the internals of each environment are set by defining appropriate instances of TerraME's types. Breaking up a multi-scale model in different and independent environments favors interdisciplinary research. Each environment may use a different combination of disciplinary knowledge. Figure 5 shows one environment that covers the whole Amazon region with 50x50km^2 cells. It has two nested environments, one modeling the Pará state at 10x10km^2 and the other modeling the Amapá state at 5x5km^2.

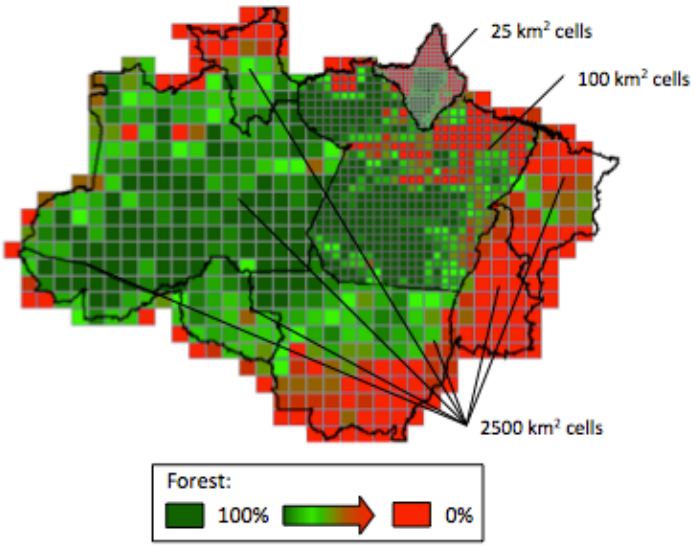

Figure 5: Environments with cellular spaces of different resolutions.

3.6 Calibration and high performance tools

TerraME provides a genetic algorithm for model calibration. It optimizes model parameters to find the best adjustment, using goodness of fit metrics to avoid local minima. It can calibrate several parameters simultaneously, even when the model is stochastic and the error function is noisy (Fraga et al., 2010). Currently, we are using the goodness-of-fit measure proposed by Costanza (1989). Future versions of TerraME will include other goodness-of-fit metrics and optimization methods to improve calibration. We have also built a high performance layer to use multiple cores in shared memory architectures. High performance services can be used during model calibration to explore larger search spaces (Silva et al., 2011). A version for distributed memory architectures is currently under development.

4 Examples of dynamic models in TerraME

This section shows case studies that explore the functions of TerraME. We focus mainly on the toolbox instead of showing details of each model.

4.1 A simple land change model

The first example is a land change model whose spatial support is a cellular space of 25x25 km^2 cells representing the Brazilian Amazonia rain forest (shown in Figure 6). This model is a simplified version of the model developed by (Aguiar, 2006).

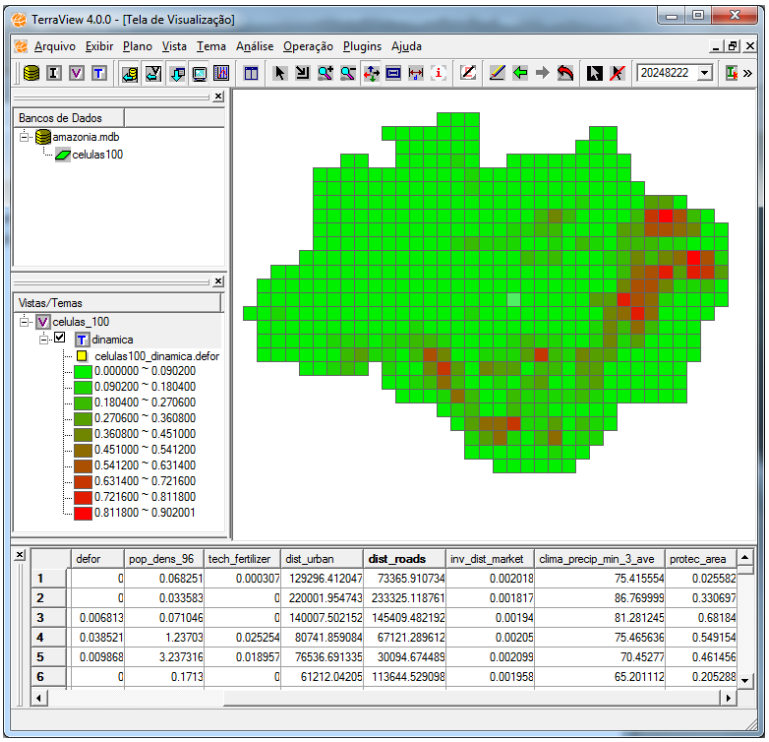

Figure 6: Brazilian Amazonia database. The attribute percentage of deforestation is used to colour the map, with green representing the cells with forest and red the percentage of deforestation.

The first part of the model (shown in Figure 7) describes the spatial entities. An object of type *CellularSpace* is created to read data from the Amazonia database. It requires a database location, the name of the theme within the database, and the attributes to be read. The "amazonia" CellularSpace connects to a Microsoft Access database and loads the attributes "percent_defor" (percentage of deforestation, from zero to one), "distance_urban" (distance to urban centers), "inv_distance_market"

(inverse of the square of distance to markets), and "protection_area" (percentage of protected areas in the cell). These attributes are read for all cells. The last line defines a Moore neighborhood for each cell.

```
amazonia = CellularSpace {
    database = "C:\\amazonia.mdb",
    theme = "dinamica",
    select = {"percent_defor", "distance_urban",
              "inv_distance_market", "protection_area"}
}

amazonia:createNeighborhood{strategy = "moore"}
```

Figure 7: Defining a CellularSpace.

Once the attributes are read into the cellular space, we define a function called *calculatePotential()* to estimate the deforestation potential of each cell, as shown in Figure 8. It takes a cellular space as argument and uses the second order functions *forEachCell()* and *forEachNeighbor()*. A second order function takes an object and another function as arguments and applies this function to every element of the given object. We use *forEachCell() to* traverse a cellular space, applying a function to all cells. Inside this function, we call *forEachNeighbor()* to traverse the neighborhood of each cell. In this example, *forEachNeighbor()* is used to sum the deforestation of all neighbors of a cell. The expected deforestation for each cell is a weighted sum of the average deforestation of its neighbors, its distance to urban centers, its connection to markets, and its percentage of protected areas. Each cell will get a new attribute called *potential* that represents its deforestation potential, computed as the difference between the expected deforestation and the current deforestation. The function returns the total potential for change, calculated as the sum of each individual potential.

```
calculatePotential = function(cellular_space)
    total_potential = 0

    forEachCell(cellular_space, function(cell)
        cell.potential = 0
        sum_neighbors  = 0
        if cell.percent_defor > 0.9999 then return end
        forEachNeighbor(cell, function(cell, neighbor)
            sum_neighbors = sum_neighbors + neighbor.percent_defor
        end)
        expected = - 0.15 * math.log10(cell.distance_urban)
                   + 0.73 * sum_neigh / cell:getNeighborhood():size()
                   + 0.05 * cell.inv_distance_market
                   - 0.07 * cell.protection_area
                   + 0.77

        if expected > cell.percent_defor then
            cell.potential = expected - cell.percent_defor
            total_potential = total_potential + cell.potential
        end
    end)
    return total_potential
end
```

Figure 8: Land change potential procedure.

After calculating the potential of each cell, the model allocates 30,000 km^2 of deforestation in the Brazilian Amazonia over a 50-year time span. To do this, it uses the algorithm presented in Figure 9, which takes a cellular space and its total potential for change as inputs. It defines a *trajectory* to traverse the cells that have a positive deforestation potential, running from higher to lower potential values. To select the cells with positive potential for change, it uses the parameter *filter*. By taking the attribute "potential" as reference, the parameter *sort* arranges the cells from higher to lower potential values. The deforestation area of each cell is then allocated as a function of its potential for change. There is an extra check to avoid the percent of deforestation of a cell going over 100%. Deforestation takes place until at least 99.9% of the initial demand has been allocated.

```
deforest = function(cellular_space, total_potential)
    trajectory = Trajectory {
        target = cellular_space,
        filter = function(cell) return cell.potential > 0 end,
        sort = compareByAttribute("potential", ">")
    }
    total_demand = 30000

    while total_demand > 30 do
        forEachCell(trajectory, function(cell)
            newarea= (cell.potential/total_potential) * total_demand
            cell.percent_defor= cell.percent_defor + newarea / 10000
            excess = 0
            if cell.percent_defor >= 1 then
                total_potential = total_potential - cell.potential
                cell.potential = 0
                excess = (cell.percent_defor - 1) * 10000
                cell.percent_defor = 1
            end
            total_demand = total_demand - (newarea - excess)
        end)
    end
end
```

Figure 9: Land change allocation procedure.

To wrap up the model, we define its temporal component, composed by a *timer* with a single *event*, as shown in Figure 10. The event calls *calculatePotential()* to compute the potential and then *deforest()* to allocate deforestation. The simulation starts in 2000 and runs until 2050. Figure 11 shows three parameters of the model and the evolution of deforestation along a simulation.

```
t = Timer {
    Event {time = 2000, action = function(event)
        total_potential = calculatePotential(amazonia)
        deforest(amazonia, total_potential)
    end}
}
t:execute(2050)
```

Figure 10: A Timer with a single Event to simulate deforestation.

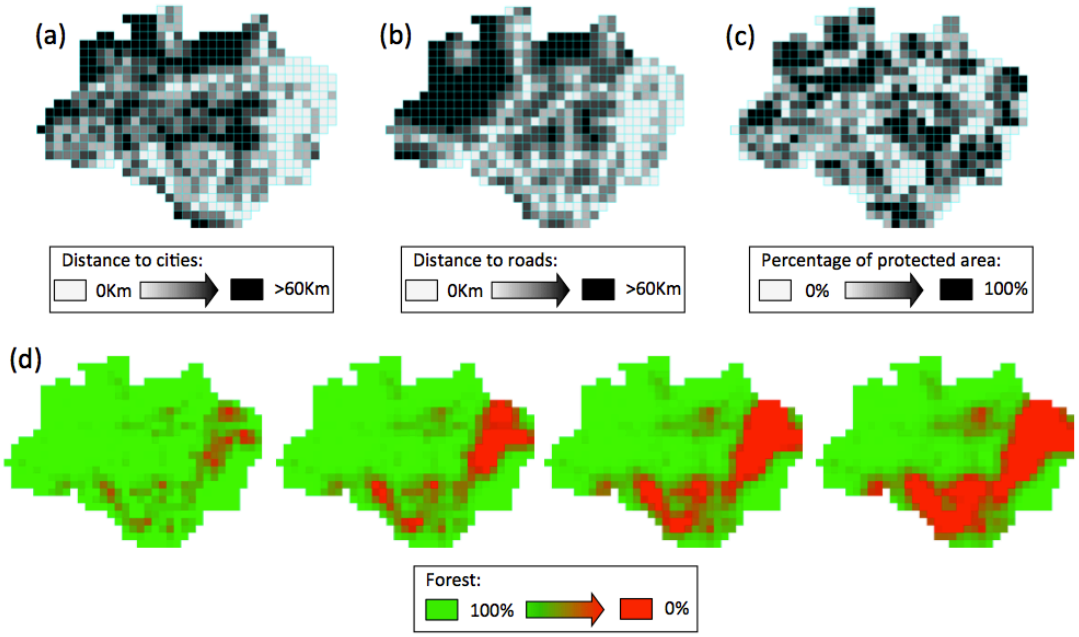

Figure 11: Amazonia deforestation model. (a) Distance to cities; (b) Distance to roads; (c) Percentage of protected areas; (d) Deforestation in the first, 15th, 35th, and 50th year.

4.2 A multi-scale continent-ocean-atmosphere model

The second example simulates a water cycle involving atmosphere, continent, and ocean, as follows:

- Water in the continent flows by gravity into the ocean;
- The height of the ocean is kept the same among its cells;
- Water in the ocean evaporates to the atmosphere;
- Water vapour in the atmosphere goes to higher altitudes by convection;
- High concentrations of water vapor turn into rain, moving water from the atmosphere to the continent.

The model has three cellular spaces. The atmosphere has a spatial overlay with continent and ocean, while some cells in the border of the continent touch other cells in the ocean. Figure 12 shows the layers and the water flows.

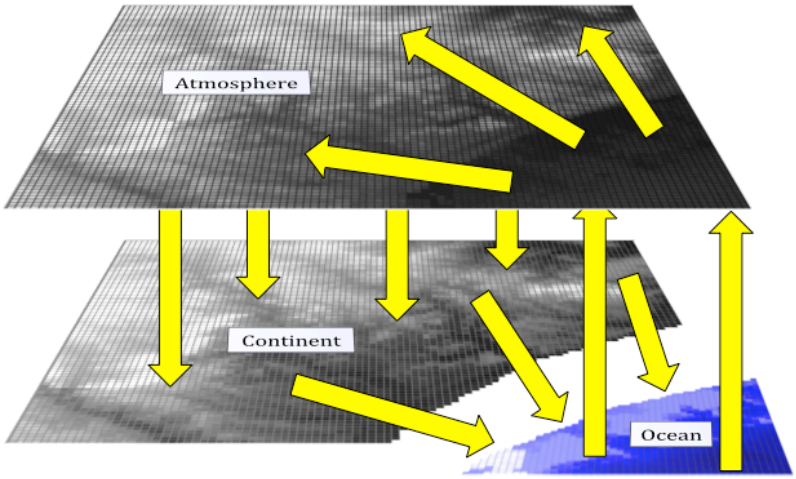

Figure 12: Atmosphere-continent-ocean database and water flows.

The first step to implement this model is to define three cellular spaces (*ocean*, *athmosphere*, and *continent*). Figure 13 shows the source code for reading the continent cellular space from a database. The continent has three attributes: height, quantity of water, and infiltration capacity. The other cellular spaces are created in a similar way.

```
continent = CellularSpace {
    database = "C:\\bd_sergipe.mdb",
    layer = "sergipe_100x100_mod_con",
    select = {"height", "qty_water", "infiltration_capacity"}
}
```

Figure 13: A cellular space representing the continent.

The next step defines the neighborhoods. In the continent, the neighborhood of a cell depends on its height and that of its adjacent cells. Only cells with a lower height belong to a cell's neighborhood. This strategy sets up a local drainage direction for each cell to simulate the water flow by gravity. Figure 14 shows the code to create the continent's neighborhood using a filter over a 3x3 neighborhood. In the end of this procedure, cells where all 3x3 neighbors are higher will have no neighbors. Such cells correspond to depression areas.

```
continent:createNeighborhood{
    strategy = "3x3",
    filter = function(cell, neighbor)
        return cell.height > neighbor.height
    end}
```

Figure 14: Creating a local drainage direction neighborhood.

We also need to set connections *between* cellular spaces to simulate evaporation, precipitation, and discharge. Figure 15 shows how to connect the atmosphere to the continent using *createNeighborhood()*. The argument *target* indicates that a connection will be created between cellular spaces, from the one that calls the function to its target. The geometric matching between the cellular spaces is defined by the argument *strategy*. The strategy "*coord*" connects two cellular spaces whose spatial positions are the same. Other connections in the model are created similarly. As we have more than one neighborhood associated to each cell, we need to give a name to the new neighborhood. In this case, the name is "atmosphere_continent".

```
atmosphere:createNeighborhood{
    strategy = "coord",
    target = continent,
    name = "atmosphere_continent"
}
```

Figure 15: Coupling the atmosphere with the continent.

After describing the spatial entities and connecting them, we now set the water flows. For the sake of simplicity, we show only the continent's behavior, as the other cellular spaces use similar strategies. Water flows downstream (runoff) and also permeates the continent (infiltration). We express these two processes separately in the model.

In the runoff calculation, water in the higher cells flows to the lower ones. Recall the continent's neighborhood is a local drainage. Using the neighborhood, we divide the water flow from a cell to its neighbors, as shown in Figure 16. To compute the water flows, we need to keep two copies of each cell. One contains the water that will flow out of the cell. The other will receive water from upstream neighbors,

which will be kept for the next iteration. For this purpose, TerraME has two versions of the attributes of a cellular space in memory. One stores past values of each cell's attributes, while the other stores the current (updated) values. This helps to simulate processes that occur in parallel in space. Past attributes are read only, as changes take place in the current time. Before updating the cells, it is necessary to *synchronize()* the cellular space. This updates the past values with the current attributes, so we can start another simulation step.

```
continent_water_balance = function()
    continent:synchronize()
    forEachCell(continent, function(cell)
        cell.qty_water = 0
    end)

    forEachCell(continent, function(cell)
        qty_neigh = cell:getNeighborhood():size()
        if qty_neigh > 0 then
            runoff = cell.past.qty_water / qty_neigh
            forEachNeighbor(cell, function(cell, neighbor)
                neighbor.qty_water = neighbor.qty_water + runoff
            end)
        else
            cell.qty_water = cell.past.qty_water
        end
    end)
end
```

Figure 16: Continent water runoff balance.

Water infiltration to the continent is a continuous process that needs to be discretized within the simulation. It is described as an event-driven function that computes a numerical integration algorithm using the built-in function *integrate()*, as shown in Figure 17. When the simulation triggers the event to execute water infiltration, the integration is computed for each cell using the period between the current time and the last time the event was executed. The main parameter of *integrate()* is the *equation* to be integrated. In this example, the numerical integration uses an *infiltration()* function that states the water in a cell will be reduced by 0.03 units per unit of time. The other arguments are the integration *method* ("euler"), an *initial* value, the triggering *event*, and the integration *step*.

```
continent_water_infiltration = function(event)
    forEachCell(continent, function(cell)
        cell.qty_water = integrate{
            equation = infiltration,
            method = "euler",
            initial = cell.qty_water,
            event = event,
            step = 0.001
        }
    end)
end

infiltration = function(t)
    return -0.03 * t
end
```

Figure 17: Continent water infiltration.

After creating the behavior within the continent, we define temporal entities. Figure 18 shows the timer that controls the continent's simulation. It has two events, which may have *priorities* to define their execution order. Lower values denote higher priority, with zero being the default value. The first event simulates water balance flows in the continent, while the second simulates the water infiltration. This timer and the cellular space representing the continent are then joined to make up an Environment. Using similar procedures as those that set up the continent environment, we can create the ocean environment and the atmosphere environment.

```
continent_timer = Timer {
    Event {time = 1, action = continent_water_balance, priority = 5},
    Event {time = 1, action = continent_water_infiltration},
}
continent_environment = Environment {
    continent,
    continent_timer
}
```

Figure 18: A Timer and Environment for the continent.

The next step describes how water moves between cellular spaces: discharge (continent to ocean), rain (atmosphere to continent), and evaporation (ocean to

atmosphere). Figure 19 describes the source code for the water discharge. As water arrives in the lower cells on the border of the continent, the model sends water from the continent to the ocean. In this case, functions *getNeighborhood()* and *forEachNeighbor()* use the name of the neighborhood that connects the continent to the ocean as their argument.

```
execute_discharge = function()
    forEachCell(continent, function(cell)
        qty_neighbors = cell:getNeighborhood("continent_ocean"):size()
        if qty_neighbors == 0 then
            return
        end
        qty_water = cell.qty_water / qty_neighbors
        cell.qty_water = 0
        forEachNeighbor(cell,    "continent_ocean",    function(cell,
neighbor)
            neighbor.qty_water = neighbor.qty_water + qty_water
        end)
    end)
end
```

Figure 19: Source code for water discharge.

The three environments (ocean, continent, and atmosphere) are enclosed in a global one, as shown in Figure 20. The global environment also has a timer that triggers events to distribute the initial flow of water, make it rain, and execute evaporation and discharge. The event that executes rain has a parameter *period* to indicate that it will execute three times less frequently than the other events. To change the amount of rain along the simulation, one could change function *execute_rain()* or reduce its periodicity. Finally, we set the global environment to be executed until time 2000. During the simulation, the global environment synchronizes the timers so that all events occur in the correct order. Figure 21 shows the flow of water in each cellular space at the end of a simulation. It is possible to see the emergence of global patterns of water from the local rules defined by the model.

```
world = Environment {
    atmosphere_environment,
    continent_environment,
    ocean_environment,
    Timer {
        Event {time = 1, action = input_initial_water, priority = -
10},
        Event {time = 3, action = execute_rain, period = 3},
        Event {time = 1, action = execute_sun, priority = 1},
        Event {time = 1, action = execute_discharge, priority = 1}
    }
}
world:execute(2000)
```

Figure 20: The world environment.

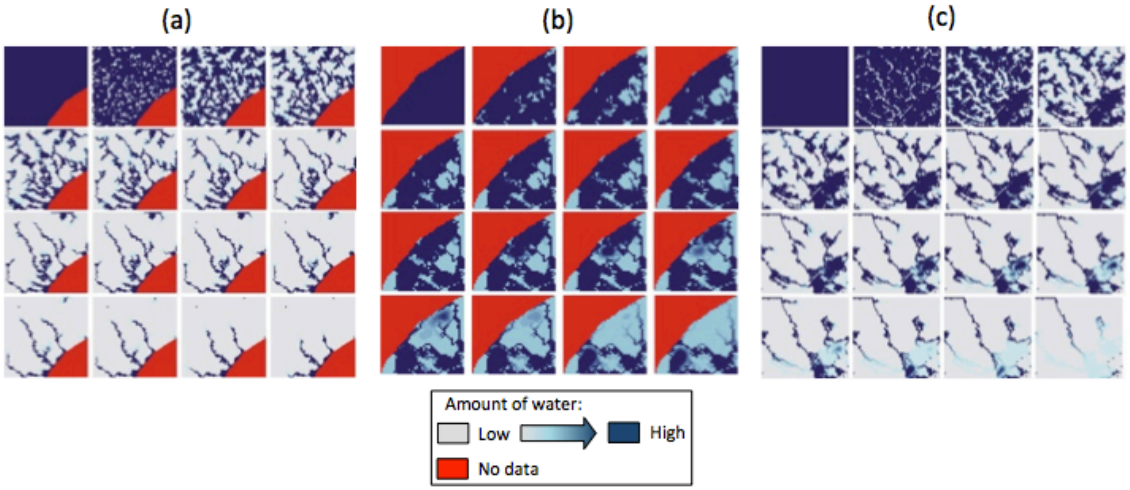

Figure 21: Results of the water cycle simulation: (a) continent, (b) ocean, (c) atmosphere.

4.3 A simple predator-prey model

The last example describes a predator-prey model using an agent-based approach. In this model, preys and predators are represented as individuals that live in a cellular space. The type *Agent* encapsulates the attributes and behaviour of autonomous individuals. A prey has two properties, *energy* and *name*, and a function, *execute()*. Energy represents its current fitness, starting with 50 quanta, while name

distinguishes preys from predators. The function *execute()* has a single parameter representing the prey itself. It describes the actions executed by the prey at each time step. In the beginning, the prey loses one quantum of energy to move from its current cell to a random neighbour. Then it checks its energy. When it has 60 or more quanta of energy, the prey reproduces asexually, creating a descendant in the same cell. When its energy is equal or less than zero, the prey dies. Finally, if there is grass in the cell, the prey feeds on it, converting the cell's cover from grass to soil to increasing its own energy by five quanta. Figure 22 represents the prey agent.

```
prey = Agent {
    energy = 40,
    name = "prey",
    execute = function(self)
        self.energy = self.energy - 1
        self:move(self:getCell():getNeighborhood():sample())
        if self.energy >= 60 then
            self.energy = self.energy / 2
            self:reproduce()
        elseif self.energy <= 0 then
            self:die()
        end
        if self:getCell().cover == "grass" then
            self:getCell().cover = "soil"
            self.energy = self.energy + 5
        end
    end
}
```

Figure 22: Describing a prey as an agent.

A predator is described similarly. It loses energy, moves, reproduces, and dies in the same way as a prey. The difference is that it looks for preys in the cell it belongs. We use *forEachAgent()* to go through a collection of agents, applying a function to each one. In this example, we use *forEachAgent()* to represent how predators feed on preys. A predator looks for a prey in the same cell it is located. If there is a prey, the predator kills and eats it, increasing its energy by half of the prey's energy. At each time step, the predator stops searching for other preys after finding the first one.

```
predator = Agent {
    energy = 40,
    name = "predator",
    execute = function(self)
        -- ... Lose energy, move, reproduce, and die as a prey
        forEachAgent(self:getCell(), function(agent)
            if agent.name == "prey" then
                self.energy = self.energy + agent.energy / 2
                agent:die()
                return false -- Found a prey, stop forEachAgent
            end
        end)
    end
}
```

Figure 23: Describing a predator as an agent.

The second part of this model creates one society of *predators, one of preys* and a cellular space where they will be located. The agents defined previously are used as prototypes that will be cloned to create both *societies*. In the example, both societies have 200 agents cloned from their respective prototypes. The model also creates an environment composed by the cellular space and the societies. To put agents in the cellular space, we call the function *createPlacement()* using a random strategy. A timer defines cycles of preys, predators, and grass regrowth (Figure 24).

The initial distribution of agents and the result of one simulation are shown in Figure 25. Green cells are filled with grass. Black asterisks represent preys, while red asterisks represent predators. In the beginning, all of the cells are green since the cellular space is filled with grass. As the simulation proceeds, preys feed grass, which changes the colour of cells to white. The number of agents within each society also changes, as they feed, reproduce, and die.

```
preys     = Society {instance = prey,     quantity = 200}
predators = Society {instance = predator, quantity = 200}

cs = CellularSpace {xdim = 100, ydim = 100}
cs:createNeighborhood()

env = Environment {cs, preys, predators}
env:createPlacement{strategy = "random"}

timer = Timer {
    Event {action = grass_regrowth},
    Event {action = preys},
    Event {action = predators}
}

timer:execute(40)
```

Figure 24: Societies and other objects for the predator-prey model.

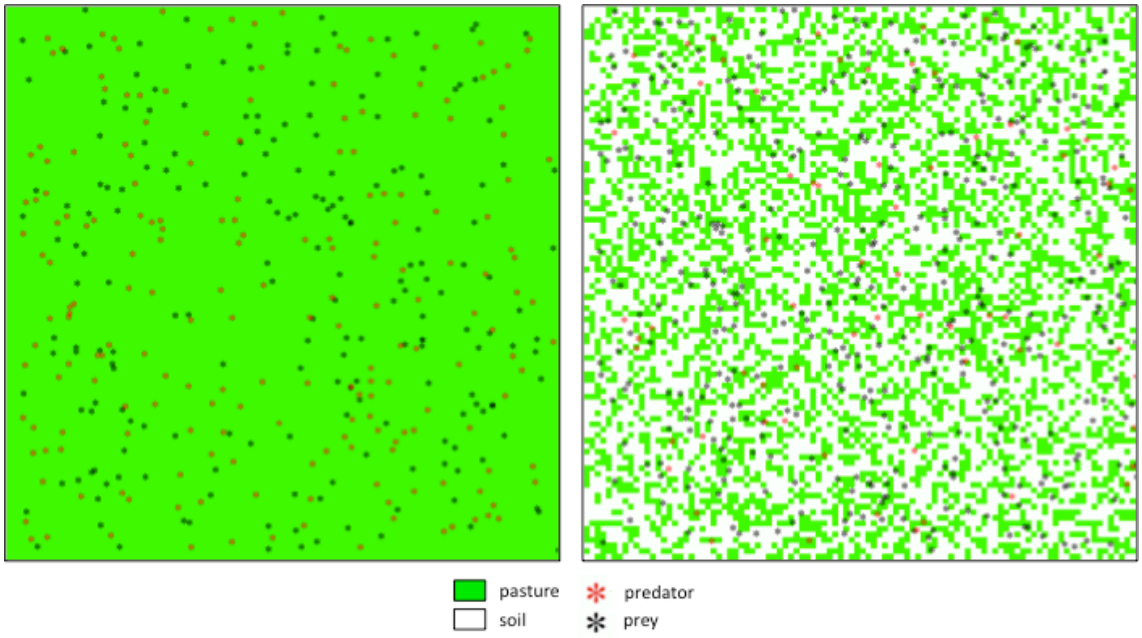

Figure 25: Simulation of a predator-prey model (left, initial condition; right, final state).

5 Discussion and Final Remarks

In this section, we consider the lessons learned when designing TerraME. We start by recalling our conjectures: a toolkit for modeling nature-society interactions needs to provide a set of data types with methods to build and connect geospatial micro-worlds. Thus, the lowermost level of TerraME has two data types: *Cell* and *Agent*. Cells represent the spatial partitions, with attributes that capture the variations of the natural and the human-built worlds. Agent represents autonomous individuals that can change the landscape. Two containers come right above both types. A set of cells representing a geographic area of interest with a given resolution and extent makes up a *CellularSpace*. A set of agents that have the same set of attributes and basic behaviour compose a *Society*. Both sets can have their entities loaded directly from a geospatial database, which simplifies dealing with real-world data. Some simulation toolkits have added interfaces to geospatial databases as an extension from their original concepts. By contrast, manipulating geospatial data is native in TerraME.

Another innovation in TerraME is the idea of environment. An environment represents a micro-world with one or more cellular spaces and one or more societies. Inside an environment, there is temporal coherence between its events. Using the idea of environments, models can be composed of sub-models with different spatial and temporal resolution and behaviour. This bottom-up logic allows for considerable flexibility. Simple models can be built using a single cellular space or a single society, without the need to define environments. Complex models will use environments to implement micro-worlds separately and couple them.

TerraME's flexibility comes at a price, however. To understand why, consider some of the alternative toolkits. If the user's problem can be expressed as sets of operations over maps, then map algebra toolkits such as PCRaster (Karssenberg et al., 2001) provide higher-level operations. Instead of iterating over every cell of a map as TerraME does, map algebra functions take a map (or a cellular space) as an atomic unit. Single statements in map algebra need a considerable number of lines in TerraME. Nevertheless, if the problem requires combining agents with maps, it is probably easier to express such models in TerraME than in a map algebra toolkit.

We recognize that prospective users will pay a price for the flexibility provided by TerraME. The learning curve will be steeper than that of a single-paradigm model. Also, there are no previous examples of similar tools that the user is likely to be familiar with. All of this may place a barrier for first-time users of TerraME. Nevertheless, we consider that there is space for a multi-paradigm modeling tool. Some problems will be too complex to fit in a single paradigm. Also, users that want to combine different approaches can benefit for having these concepts supported in a single tool.

When comparing nature-society modeling tools, it is useful to consider the lessons learned from programming languages in general. It is unlikely that a single programming language will fit the needs of all software developers. There is room for scripting, object-oriented, functional and multi-paradigm languages. The same view applies to modeling. The community will benefit for multiple solutions. We believe that TerraME is a new approach to nature-society modeling, which will find its niche alongside existing and mature tools.

References

Aguiar, A.P.D., 2006. Modeling Land Use Change in the Brazilian Amazon: Exploring Intra-Regional Heterogeneity, PhD Thesis, Remote Sensing Program. INPE: Sao Jose dos Campos.

Batty, M., 2012. A generic framework for computational spatial modelling, In: Heppenstall, A., Crooks, A., See, L., Batty, M. (Eds.), Agent-based models of geographical systems. Springer: Dordrecht, NL, pp. 19-50.

Beven, K., Binley, A., 1992. The future of distributed models: model calibration and uncertainty prediction. Hydrological Processes 6(3) 279-298.

Câmara, G., Vinhas, L., Ferreira, K., Queiroz, G., Souza, R.C.M., Monteiro, A.M., Carvalho, M.T., Casanova, M.A., Freitas, U.M., 2008. TerraLib: An open-source GIS library for large-scale environmental and socio-economic applications, In: Hall, B., Leahy, M. (Eds.), Open Source Approaches to Spatial Data Handling. Springer: Berlin, pp. 247-270.

Costanza, R., 1989. Model Goodness of Fit - a Multiple Resolution Procedure. Ecological Modelling 47(3-4) 199-215.

Crooks, A., Castle, C., 2012. The Integration of Agent-Based Modelling and Geographical Information for Geospatial Simulation, In: Heppenstall, A., Crooks, A., See, L., Batty, M. (Eds.), Agent-Based Models of Geographical Systems. Springer-Verlag: Heidelberg.

Eberlein, R.L., Peterson, D.W., 1992. Understanding models with Vensim (TM). European journal of operational research 59(1) 216-219.

Filippi, J., Bisgambiglia, P., 2004. JDEVS: an implementation of a DEVS based formal framework for environmental modelling. Environmental Modelling and Software 19(3) 261-274.

Forrester, J.W., 1961. Industrial dynamics. MIT Press Cambridge, MA.

Fraga, L., Carneiro, T., Lana, R., Guimarães, F., 2010. Calibração em Modelagem Ambiental na Plataforma TerraME usando Algoritmos Genéticos (Enviromental Modelling Calibration using TerraME using Genetic Algorithms), 42 Brazilian Symposium on Operations Research: Bento Gonçalves, Brazil.

Gray, J., 1981. The transaction concept: virtues and limitations, 7th International Conference on Very Large Data Bases (VLDB). IEEE Computer Society: Cannes, France, pp. 144-154.

Henzinger, T.A., 1996. The Theory of Hybrid Automata, IEEE Symposium on Logic in Computer Science (LICS'96). IEEE: New Brunswick, NJ , USA, pp. 278 - 292

Ierusalimschy, R., Figueiredo, L.H., Celes, W., 1996. Lua - an extensible extension language. Software: Practice & Experience 26(6) 635-652.

Jakeman, A.J., Letcher, R., Norton, J., 2006. Ten iterative steps in development and evaluation of environmental models. Environmental Modelling & Software 21(5) 602-614.

Janssen, P.H.M., Heuberger, P.S.C., 1995. Calibration of process-oriented models. Ecological Modelling 83(1–2) 55-66.

Karssenberg, D., Burrough, P.A., Sluiter, R., de Jong, K., 2001. The PCRaster software and course materials for teaching numerical modelling in the environmental sciences. Transactions in GIS 5 99-110.

Karssenberg, D. and K. De Jong (2005). "Dynamic environmental modelling in GIS: 1. Modelling in three spatial dimensions." International Journal of Geographical Information Science 19(5): 559-579.

Karssenberg, D., Schmitz, O., Salamon, P., De Jong, K., Bierkens, M.F.P., 2009. A software framework for construction of process-based stochastic spatio-temporal models and data assimilation. Environmental Modelling & Software 25 489-502.

Lin, Z., Beck, M.B., 2012. Accounting for structural error and uncertainty in a model: An approach based on model parameters as stochastic processes. Environmental Modelling and Software 27-28 97-111.

Moran, E. F. (2010). Environmental social science : human-environment interactions and sustainability. Malden, Mass., Wiley-Blackwell.

Muetzelfeldt, R.I., Massheder, J., 2003. The Simile visual modelling environment. European Journal of Agronomy 18 345-358.

North, M.J., Collier, N.T., Vos, J.R., 2006. Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit. ACM Transactions on Modeling and Computer Simulation 16(1) 1-25.

Parker, D. C., Manson, S.M., Janssen, M.A., Hoffmann, M.J., Deadman, P., 2003. "Multi-agent systems for the simulation of land-use and land-cover change: a review." Annals of the Association of American Geographers 93(2): 314-337.

Pontius Jr, R.G., Millones, M., 2011. Death to Kappa: birth of quantity disagreement and allocation disagreement for accuracy assessment. International Journal of Remote Sensing 32(15) 4407-4429.

Rindfuss, R.R., Walsh, S.J., Turner, B.L., Fox, J., Mishra, V., 2004. Developing a science of land change: Challenges and methodological issues. Proceedings of the National Academy of Sciences 101(39) 13976-13981.

Roberts, N., Anderson, D., Deal, R., Garet, M., Shaffer, W., 1983. Introduction to Computer Simulation: A System Dynamics Modeling Approach. Addison-Wesley., Reading, MA.

Silva, S., Lima, J., Carneiro, T., 2011. Parallel Calibration of Spatial Dynamic Models in TerraME, IADIS - International Conference on Applied Computing Rio de Janeiro, Brazil,, pp. 451-456.

Steiniger, S., Bocher, E., 2009. An Overview on Current Free and Open Source Desktop

GIS Developments. International Journal of Geographic Information Science 23(10) 1345-1370.

Stroustrup, B., 1994. Design and Evolution of C++. Addison-Wesley, New York.

Tisue, S., Wilensky, U., 2004. NetLogo: A Simple Environment for Modeling Complexity, Boston: International Conference on Complex System.

Tomlin, C.D., 1990. Geographic Information Systems and Cartographic Modeling. Prentice-Hall, Englewood Cliffs, NJ.

Turner, B., Skole, D., Sanderson, S., Fischer, G., Fresco, L., Leemans, R., 1995. Land-Use and Land-Cover Change (LUCC): Science/Research Plan, HDP Report No. 7. IGBP Secretariat: Stockholm.

Verstegen, J.A., Karssenberg, D., Van der Hilst, F., Faaij, A., 2012. Spatio-temporal uncertainty in Spatial Decision Support Systems: A case study of changing land availability for bioenergy crops in Mozambique. Computers, Environment and Urban Systems 36(1) 30-42.

von Neumann, J., 1966. Theory of Self-Reproducing Automata. Edited and completed by A.W. Burks., Illinois.

Wesseling, C.G., Karssenberg, D., Van Deursen, W.P.A., Burrough, P.A., 1996. Integrating dynamic environmental models in GIS: the development of a Dynamic Modelling language. Transactions in GIS 1 40-48.

White, R. and G. Engelen (1997). "Cellular automata as the basis of integrated dynamic regional modelling." Environment and Planning B: Planning and Design 24: 235-246.

Wooldridge, M.J., Jennings, N.R., 1995. Intelligent agents: Theory and practice. Knowledge Engineering Review 10(2).

Zeigler, B.P., Kim, T.G., Praehofer, H., 2005. Theory of modeling and simulation. Academic Press, Inc., Orlando, FL, USA.