Notas em Matemática Aplicada e-ISSN 2236-5915

Volume 40, 2012

Editado por

Célia A. Zorzo Barcelos
Universidade Federal de Uberlândia - UFU
Uberlândia, MG, Brasil

Eliana X.L. de Andrade
Universidade Estadual Paulista - UNESP
São José do Rio Preto, SP, Brasil

Maurílio Boaventura
Universidade Estadual Paulista - UNESP
São José do Rio Preto, SP, Brasil

Sociedade Brasileira de Matemática Aplicada e Computacional

2012

A Sociedade Brasileira de Matemática Aplicada e Computacional - SBMAC publica, desde as primeiras edições do evento, monografias dos cursos que são ministrados nos CNMAC.

Para a comemoração dos 25 anos da SBMAC, que ocorreu durante o XXVI CNMAC em 2003, foi criada a série **Notas em Matemática Aplicada** para publicar as monografias dos minicursos ministrados nos CNMAC, o que permaneceu até o XXXIII CNMAC em 2010.

A partir de 2011, a série passa a publicar, também, livros nas áreas de interesse da SBMAC. Os autores que submeterem textos à série Notas em Matemática Aplicada devem estar cientes de que poderão ser convidados a ministrarem minicursos nos eventos patrocinados pela SBMAC, em especial nos CNMAC, sobre assunto a que se refere o texto.

O livro deve ser preparado em **Latex (compatível com o Miktex versão 2.7), as figuras em eps** e deve ter entre **80 e 150 páginas**. O texto deve ser redigido de forma clara, acompanhado de uma excelente revisão bibliográfica e de **exercícios de verificação de aprendizagem** ao final de cada capítulo.

Veja todos os títulos publicados nesta série na página
http://www.sbmac.org.br/notas.php

SBMAC **Sociedade Brasileira de Matemática Aplicada e Computacional**

2012

TerraME : SUPORTE A MODELAGEM AMBIENTAL MULTI-ESCALAS INTEGRADA A BANCOS DE DADOS GEOGRÁFICOS

Tiago Garcia de Senna Carneiro
tiago@iceb.ufop.br

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Gilberto Camara
gilberto@dpi.inpe.br

Divisao de Processamento de Imagem
Coordenação de Observação da Terra
Instituto Nacional de Pesquisas Espaciais

Sociedade Brasileira de Matemática Aplicada e Computacional

São Carlos - SP, Brasil
2012

Coordenação Editorial: Sandra Mara Cardoso Malta

Coordenação Editorial da Série: Eliana Xavier Linhares de Andrade

Editora: SBMAC

Capa: Matheus Botossi Trindade

Patrocínio: SBMAC

Catalogação elaborada pela Biblioteca do IBILCE/UNESP
Bibliotecária: Maria Luiza Fernandes Jardim Froner

Carneiro, Tiago G. S.
TerraME : Suporte a Modelagem Ambiental Multi-Escalas Integrada a Bancos de Dados Geográficos - São Carlos, SP: SBMAC, 2012, 98 p.; 20,5cm - (Notas em Matemática Aplicada; v. 40)

e-ISBN 978-85-8215-001-6

1. TerraME. 2. Environmental Modeling. 3. Dynamic Modeling. 4. Nested-CA. 5. GIS. 6. Cellular Automata. 7. Agent. I. Carneiro, Tiago G. S. II. Câmara, Gilberto. III. Título. IV. Série.

CDD - 51

Esta é uma republicação em formato de e-book do livro original do mesmo título publicado em 2009 nesta mesma série pela SBMAC.

A meus pais Gabi Garcia e Alexandre Carneiro.
A minha esposa Tenesca Scofield.
A minha filha Cecília Carneiro.
Dedico

Contents

Chapter 1

Introduction

TerraME is a development environment for spatial dynamical modeling that supports the concepts of Nested Cellular Automata (Nested-CA) [1]. TerraME uses a spatial database for data storage and retrieval. A *spatial dynamic model* is a model whose time and locations are independent variables. The outcomes of these models are maps that depict the spatial distribution of a pattern or of a continuous variable. TerraME enables simulation in two-dimensional cellular spaces. Among the typical applications of TerraME are land change and hydrological models.

This tutorial provides an introduction to the basic features of TerraME. For a full description, see [1]. The tutorial has twelve parts. In chapter 2, we present the theoretical foundations of the TerraME software. In chapter 3, we present the TerraME architecture. In chapter 4, we show how to install TerraME and its development environment. In chapter 5, we present the several basic commands of the TerraME programming language. In chapters 6 and 7, we show several examples of hydrological and land change modeling, respectively, using TerraME. In chapter 8, we present the function trajectory and an example of land change model. In chapter 9, we discuss how to nest cellular spaces for coupling layers of cells with different spatial resolutions. In chapter 10, we show how to use TerraME Hybrid Automaton to simulate systems which behavior has discrete and continuous components. In chapter 11, we develop a rain drainage model to exemplify how to build discrete and continuous models based on the Cellular Automata Theory. Chapter 12 shows how one can develop models with multiple temporal resolutions. At the end of each chapter, we suggest many exercises to pratice and to test reader knowledge about TerraME concepts and language. Readers interested in an introduction to the principles

of modeling and simulation should refer to [2]-[4]. Useful discussions on spatial dynamic modeling applications include [5]-[10].

Chapter 2

Theoretical Foundations

According to [3], **modeling** is the cognitive process in which the principles of one or more theories are applied to produce a model of a real phenomenon. A **phenomenon** is any concrete fact or situation of scientific interest, which can be described or explained. Any model is an outcome from the creativity of the modeler and from the knowledge she/he has about the observed phenomenon. A **model** can be defined as a simplified and abstract representation of a phenomenon, based on a formal description of entities, their relations, and processes. Model **simulation** is the act of reproducing the behavior of some phenomenon in a computer environment [11] [22] [7].

During the modeling activity, the modeler will need to specify the structure (syntax) and functioning (semantics) of the idealized model. This specification should be represented according to the syntax and the semantics of a model of computation, for example, the *Multiagent* model of computation or the *Cellular Automata* model of computation. The term **model of computation** can be roughly defined as a formal and abstract definition of a computer.

In this section, we will briefly introduce the models of computation which serve as foundation for the TerraME software.

2.1 Finite automata

A **finite automata** or **finite state machine** is a abstract model for a real phenomenon or system and may be defined as a directed graph $G_g = (V, E_g)$, called *transition diagram*, where V is a finite set of vertices and E_g is a set of ordered vertices pairs named arcs [12]. Each graph vertex

corresponds to one automaton state. If there is a transition from the state q to the state p, as a response to one input a, them in the transition diagram G_g there is an arc from the vertex q to the vertex p with label a. Each arc is associated to a transition rule which determines if the transition described by the arc will be executed.

The finite automata model uses a discrete time base [13]. The variable t which represents time is assigned to discrete values $0, \pm 1, \pm 2, \ldots$. The behavior of the automata is a linear sequence of events in time. Since the set of possible states is finite, a finite automaton is not appropriate to simulate behavior where the set of system states is potentially infinite.

Figure 2.1 shows a transition diagram for a finite automaton capable to store a binary digit that was provided as input at the instant $t - 1$. The symbol that triggers a transition is presented at the origin of the arcs. The symbol at the middle of an arc represents the response of the machine at the transition time.

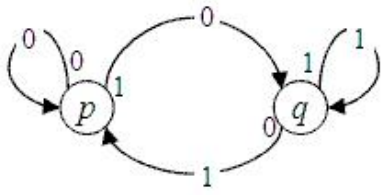


Figure 2.1: Transition diagram for the memory machine.

Due to its simplicity, existence of an underling formal theory, and event-driven properties, the *finite automata* model [13] is widely used for modeling dynamical systems where the flow control is neither sequential nor predetermined because it depends on external events.

2.2 Hybrid automata

A **hybrid automaton** is an abstract model for a system which behavior has discrete and continuous components, that is, a hybrid system. A hybrid automaton consists of a finite automaton equipped with continuous variables and continuous operations over them [14]. A hybrid automaton extends the idea of finite automata to allow continuous change to take place between transitions. Inside each discrete state, the automaton continuous variables are allowed to change. We have adapted Henzinger's hybrid automata model as a basis for environmental models development. As used in this work, a

hybrid automaton H is defined by the structure (X, G, *init*, $flow$, $jump$, *method*) where:

- **(a) Variables:** a finite set $X = \{x_1, \ldots, x_n\}$ of real variables, modeled as set of points in the R^n space. The notation $X' = \{x'_1, \ldots, x'_n\}$ is used to denote the set of first derivatives. The notation $X* = \{x_1*, \ldots, x_n*\}$ is used to denote the values of the set X at the moment of a transition between states.
- **(b) Control graph:** a finite directed graph $G = (V, S)$. The vertices in V represent the discrete states of the system and are named *control modes*. The edges in S model the system discrete dynamics and are called *control switches*.
- **(c) Initial condition:** The automaton H has an associated function *init*, which is the starting point of the system. It determines the initial control mode and the values of set X of model variables.
- **(d) Flow conditions:** Each control mode $v \in V$ has an associated function *flow*. The flow condition $flow(v)$ defines the behavior of the system inside each control mode and is generally specified as a differential equation.
- **(e) Jump condition:** Each control switch $s \in S$ has an edge labeling function *jump*. The jump condition $jump(s)$ is a predicate over $X \cup X*$ and determines if a control switch will be trigged;
- **(f) Method:** $\{m_1, \ldots, m_n\}$ is a set of methods, called to obtain information about the automaton internal state, or to update the value of any variable $x \in X$.

We define a *configuration* of a hybrid automaton as a pair (v, x), where $v \in V$ is the current control mode and $X^+ = \{x_1^+, \ldots, x_n^+\}$ is the current value of its variables.

Communication between automata uses remote method invocation. Each automaton provides a set of methods that can be called by other automata. By calling methods of other automata, an automaton can obtain information about their configuration. The behavior of the automaton depends on the current control mode. This determines the flow condition that will be executed and the subset of jump conditions that may cause a transition between control modes.

The hybrid automaton on the figure 2.2 models a climate variation system. The x variable represents the temperature. In the control mode *cooling*, the climate is becoming cooler and the temperature is declining

according to the flux condition $\frac{dx}{dt} = -0,1x$. In the control mode *warming*, the climate is becoming warmer and is temperature is rising according to the flux condition $\frac{dx}{dt} = 5 - 0,1x$. Initially, the temperature is $20°C$. The jump condition $x < 19$ indicates that the climate system will shift to the 'warming' mode as soon as the temperature falls below $19°C$. The jump condition $x > 21$ indicates that the climate system will shift to the 'cooling' mode as soon as the temperature is higher than $21°C$.

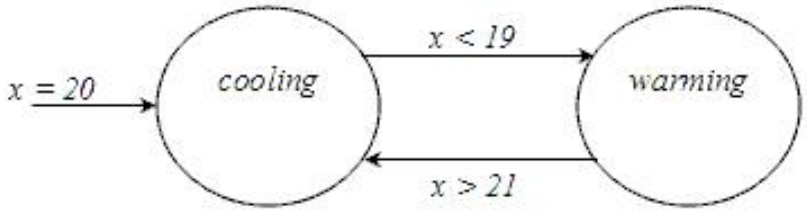


Figure 2.2: Hybrid automata model for a climate variation system (Source: adapted from [14])

2.3 Cellular automata

A **cellular automata** (CA) as conceived by [15] is comprised of a finite two-dimensional lattice of squared cells, a finite automaton, and a neighborhood relationship. Each cell is occupied by a copy of the finite automaton which is connected to its four adjacent automata. As the same set of rules is present on each cell, the cellular structure is said **functionally homogeneous**. As each automaton has the same neighborhood relationship in all directions, the von Neumann CA is said **isotropic**. As all automata have the same configuration of neighbors, it is also said **stationary**.

The finite automaton on each cell may be on a different internal state. Hence, one cellular space region can act on a given way and receives information from a determined direction while another can behave on a different manner and receives information from other direction. The CA model is useful due to its capacity to reproduce spatial changing trough diffusion processes [16][17] and since it can simulate emergent phenomena [18].

The information flow in a CA is unidirectional. When an finite automaton is being executed, it requests information from its neighbors. This information is combined with the internal state of the automaton to define the action it will take. Figure 2.3(a) presents the view of a portion of a CA

lattice, showing the CA finite automaton on different states on each cell. Figure 2.3(b) shows the CA finite automata neighborhood relationship.

Figure 2.3: Cellular Automata: (a) same finite automaton on each cell - the cellular structure is functionally homogeneous, and (b) same neighborhood relationship on each cell - the cellular structure is isotropic and stationary.

2.4 Situated agents

In an attempt to capture the dynamic of phenomena whose are outcomes of several individual interactive systems acting over the space, researchers have proposed the use of agent-based models immersed in a cellular space [7]. There are different and sometimes conflicting definitions of the concept of an 'agent' [19]. This work adopts the definition provided by [20]. An agent is an abstract model for an entity that is embedded in an environment. The agent is capable of sensing the environments and of acting on it. We consider that an agent has three properties: autonomy, social ability, and reactivity. To be autonomous, an agent has to control its actions and its internal state. Granting social ability to an agent requires that agents communicate. The agent should be able to perceive its environment and react accordingly.

To combine the theory of agents to that of cellular automaton, each automaton has to perform as an agent. In this section, we consider an agent model (situated agents) that allows embedding agents in CAs [21] A situated agent is defined by the structure $M = (S, \sum, A, \delta, \lambda, s_0)$, where:

- **(a)** S is a set of finite internal states.
- **(b)** $\sum$ is a set of inputs (stimulus).

- **(c)** A is the set of outputs (actions).
- **(d)** $\delta : S \times \sum \rightarrow S$ is a function that determines the agent's next internal state.
- **(e)** $\lambda : S \rightarrow A$is the function that determines the agent's next action.
- **(f)** s_0 is the agent initial state.

An environment state ϕ be distinguished if the modeler develops a transition function δ in such way that the agent will be in internal state s for any sequence of inputs $\theta*$ that leads the environment to a condition ϕ from an initial condition ϕ_0. This establishes a correlation between the agent's internal state and the environment's state, and one can say that the agent is capable of recognizing the environment state.

In this model, agents are purely reactive. The environment E generates inputs to the agent M. The agent receives this input and performs some actions. These actions result in the agent reaching an internal state. One can then say that the situated agent is capable of taking decisions based on the state of the environment. The important aspect of situated automata theory is modeling systems such that, for each state of the environment E, there will be a corresponding state of the automaton M. The Figure 2.4 shows the coupling between a situated agent and its environment.

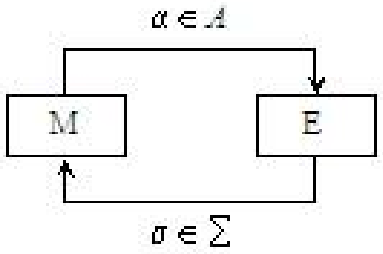


Figure 2.4: A situated agent M coupled to its environment E (source: [14])

Chapter 3

The TerraME Environment

The key part of the TerraME development environment is the *TerraME interpreter*, as shown in Figure 3.1. It reads a program written in the TerraME modelling language (a LUA language extension), interprets the source code, and calls functions in the TerraME framework. This framework is a set of modules written in C++ that provides functions and classes for spatial dynamical modelling. It also links to a *TerraLib* spatial database. The modeling results can be accessed by the TerraView application [http://www.dpi.inpe.br/terraview].

The TerraME environment consists of the following parts:

- The TerraME interpreter, which executes the model source code.

- *TerraView*, a *Geographic Information System* (GIS) application developed over the *TerraLib* C++ library for spatial database management [22]. It is used for vector and raster spatio-temporal data acquisition, visualization, and analysis.

- A text editor, as the Crimson [http://www.crimsoneditor.com/] or Notepad++[http://notepad-plus.sourceforge.net], or a Integrated Development Environment (IDE), like Eclipse [http://www.eclipse.org/], which provide highlight syntax for the LUA programming language [23] and, therefore, for the TerraME model source code .

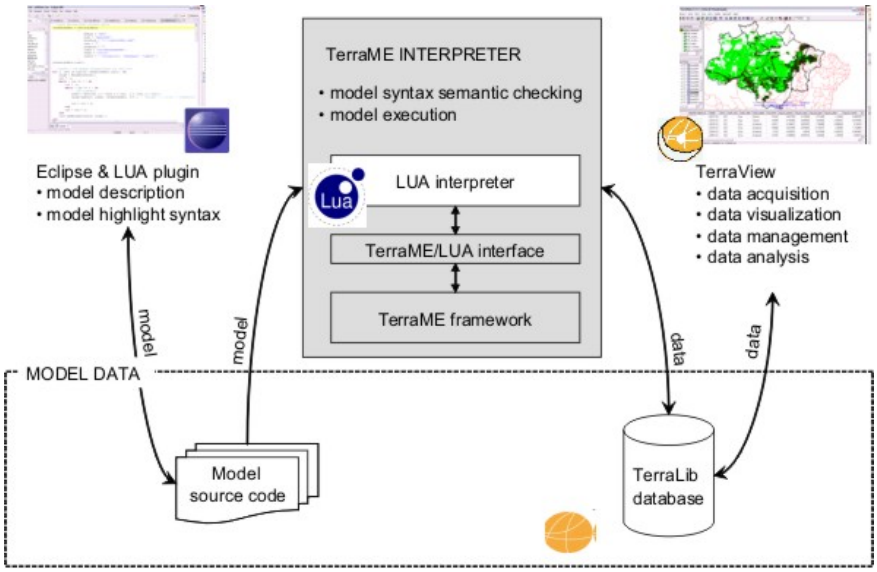


Figure 3.1: The TerraME development environment.

Figure 3.2 shows the *TerraME* architecture. Lower layers provide basic

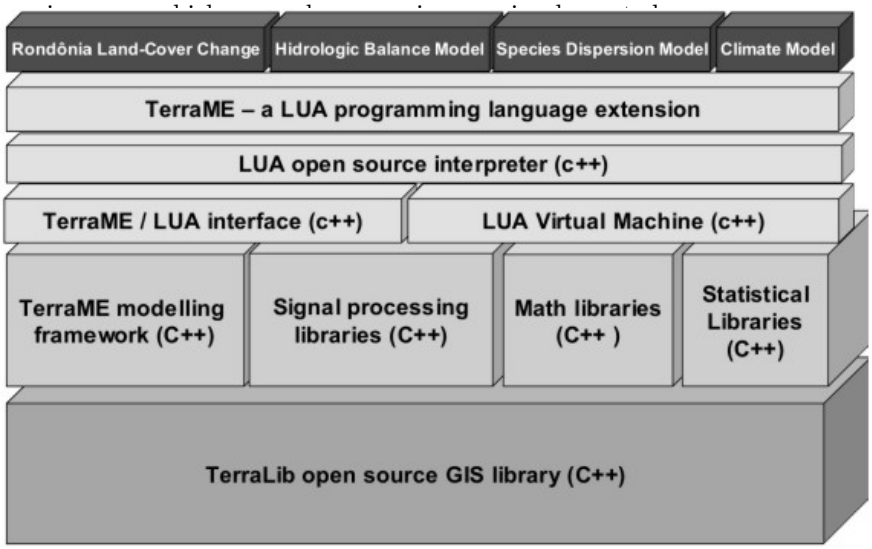


Figure 3.2: TerraME architecture.

In the first layer, *TerraLib* offers typical GIS spatial data management and analysis services, and extra functions for temporal data handling. The *TerraME framework* provides the simulation engine and the calibration and validation services. It is an open source ANSI C++ implementation of the Nested-CA model[1], portable for Windows and Unix-like operating systems. This framework can be used directly for model development. Since developing models in C++ can be a challenge for non-programmers, *TerraME* provides a high-level modelling language. The third layer of the architecture implements the *TerraME modelling language* interpreter and runtime environment. The *TerraME/LUA* interface extends LUA with new data types for spatial dynamic modelling and services for model simulation and evaluation. Using the LUA library API, it exports the *TerraME* framework API(Application Program Interface) to the LUA interpreter, so it recognizes the TerraME types. If needed, other C or C++ applications (such as statistical libraries) can have their APIs exported to the LUA interpreter and integrated in the architecture. The last layer, called application layer, includes the end-user models.

Chapter 4

TerraME installation

This section shows how to install, in the Microsoft Windows platform, the TerraME architecture and development environment, Crimson Editor, as well its configuration.

4.1 Installing the TerraME Development Environment

Initially, it's necessary to do download TerraME RC4.zip, Crimson Editor and Database Examples, from the course web page http://lucc.ess.inpe.br/doku.php?id=software. Decompress the TerraME RC4.zip file to the C:\TerraME directory, execute Crimson Editor and copy databases in the C:\TerraME\Database directory.

4.2 Configuring a TerraME Project in Crimson Editor

It's necessary start Crimson Editor. In Tools menu select Conf. user tolls. Into Preferences dialogue box select User tolls (in left side). In an empty slot, fill with the following arguments:

- Menu text: any name, example: TerraME;
- Command: search TerraME installation directory as the workspace directory (C:\TerraME\TerraME_RC4\TerraME.exe);
- Argument: $(FileDir)\$(FileName);

- With Windows use "$(FileDir)\$(FileName)";
- Initial Dir: search TerraME directory (C:\TerraME\TerraME_RC4);
- Hot Key: create a short-cut to your run the model.

Type "Ok" to finish configuration.

Chapter 5

The TerraME Modeling Language: Basic Commands

This section presents the basic *TerraME Modeling Language* mechanisms for multiple scale spatial dynamic model representation and simulation.

5.1 TerraME as a LUA Extension

Lua is an extension programming language designed to support general procedural programming with data description facilities [23],[24]. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Being an extension language, Lua has no notion of a "main" program: it only works embedded in a host client. This host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code. By using C functions, Lua can be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework [25]. The *TerraME Modeling Language* is a LUA *Programming Language* extension. It uses the LUA extensibility mechanisms to include new data types and functions.

5.2 An little introduction of LUA language

LUA is a dynamically typed language: variables do not have types; only values do. There are no type definitions. The basic value types are *number*(double) and string. The value nil is different from any other value in the language and has the type nil. Functions in *LUA* are first-class values . That is, a function definition creates a value of type *function* that can be stored in variables, passed as arguments to other functions and returned as results. The only structured data type is *table*. It implements associative arrays, that is, arrays that can be indexed not only with integers, but with *string, double, table,* or *function* values. For *table* indexing, both *table.name* and *table["name"]* are acceptable. *Tables* can be used to implement records, arrays, and recursive data types. They also provide some object oriented facilities, such as methods with dynamic dispatching [23].

```
loc = { cover = 'forest', distRoad = 0.3, distUrban = 2 };
loc.desfPot = loc.distRoad + loc.distUrban;
...
loc.reset = function( self )
        self.cover = "";
        self.distRoad = 0.0;
        self.distUrban = 0.0;
end
```

Listing 5.1: The use of associative table and function values in LUA.

Program 5.1 shows the use of table and function values. The code creates a *table* with three attributes (land cover, road distance, and urban center distance) and stores it the variable *loc*. It calculates a new attribute and adds it to *loc* (deforestation potential is the sum of the road and urban center distances). Finally, it creates a second attribute called reset and adds it to table *loc*. It is as a function that receives the *table* as parameter. This is indicated by the keyword *self*.

LUA has a powerful syntactical tool, called *constructor*. When the modeller writes *name*{...}, the *LUA interpreter* replaces it by *name*({...}), passing the *table*{...} as a parameter to the function *name( )*. This function typically initializes, checks properties values and adds auxiliary data structure or methods [23]. In Program 5.2, it constructs the type *MyLoc*. When the table *L* is instantiated, the constructor initializes the attribute *desfPot*.

```
function MyLoc( loc )
  loc.desfPot = loc.distRoad + loc.distUrban;
  return loc;
end
l = MyLoc{cover = "forest", distRoad = 0.3, distUrban = 2 };
```

Listing 5.2: The use of the *constructor* in LUA.

To build spatial dynamic models, TerraME includes new value types in LUA using the constructor mechanism. These values are: *CellularSpace, Cell, Neighborhood, Environment, Trajectorty, Automaton, Agent, State, Jump, Flow, Timer, Event and Message.*

5.3 The CellularSpace

A *CellularSpace* is a multivalued set of *Cells*. Nowadays, most spatially-explicit modeling platforms supports the concept of **regular cellular space** (RCS) for space representation, i. e., a regular two-dimensional grid of multi-valued cells grouped into neighborhoods, where the dynamic model rules operate and possibly change cells attribute values. Nevertheless, to minimize border effects and cell attributes aggregation problems which are dependent on the model chosen grid resolution [26][27], Figure 5.1, TerraME implements the concept of (Irregular Cellular Space) (ICS) purposed in [28].

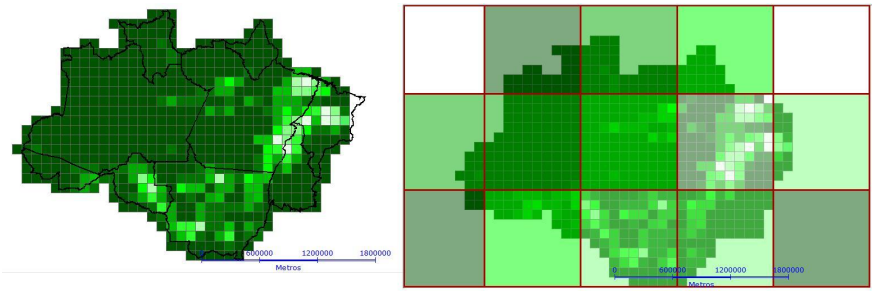


Figure 5.1: Problems due to the choice of a raster structure for space representation: aggregation of cell attribute values and border effects. Maps color: light means "deforested" and dark means "forest".

The ICS extends the spatial structure from the RCS to support the development of GIS integrated spatial dynamic models which uses many space representations for supporting multiple scale modeling. The cellular space is any irregular arrange of cells which geometrical representation may vary from a regular grid of same size squared cells to a irregular set of points, lines, polygons, nodes and arcs, pixels, or even voxels. Figure 5.2 shows three ICS, (1) $25 \times 25\ km^2$ sparse squared cells which main attribute is landCover (white = "100% forest" and green = "0% forest"); (2) each polygon representing one Brazilian State is a cell which main attribute are

name = "MG" — "SP" — " RJ" — ... — "AM" the demanded area to be deforested; and (3) each roads is a cell which main attributes are status (red = "paved" and orange = "non-paved") and brazilianSate = "MG" — "SP" — " RJ" — ... — "AM".

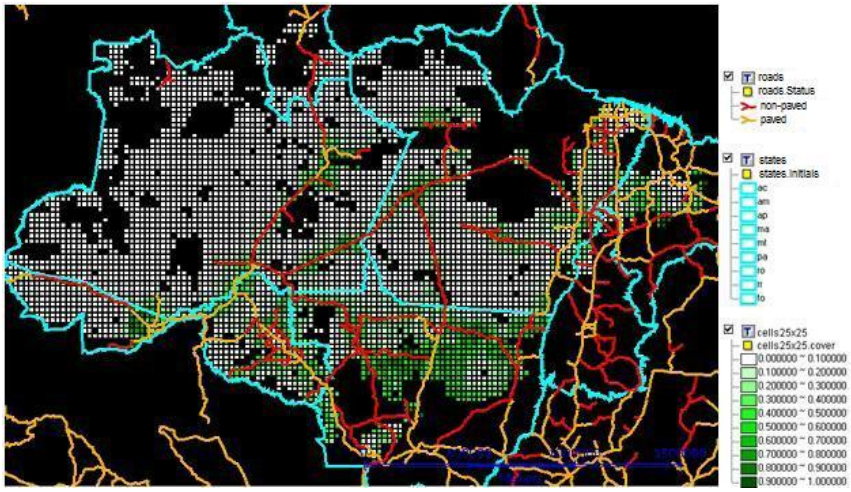


Figure 5.2: Three Irregular Cellular Spaces: (1) state polygons, (2) road lines and (3) regular squared cells.

CellularSpaces are stored and retrieved from a *TerraLib* database, so the modeller should specify the properties of the *CellularSpace* before using it, as shown in Program 5.3.

```
-- Loads a TerraLib cellular space
csCabecaDeBoi = CellularSpace {
  dbType = "ADO",
  host = "localhost",
  database = "C:\\TerraME\\Database\\CabecaDeBoi.mdb",
  user = "",
  password = "",
  layer = "cellsLobo90x90",
  theme = "cells",
  select = { "height", "capInf" },
  where = "mask <> 'noData'"
}
csCabecaDeBoi:load();

-- Dynamically creates a cellular space
cs = CellularSpace{ database = "", theme = ""}
for i = 1, 2, 1 do
```

```
  for j = 1, 2, 1 do
    c = Cell{ soilType = i*j }
    c.x = i;
    c.y = j;
    cs:add( c );
  end
end
```

Listing 5.3: An example of the definition and loading of a CellularSpace in TerraME.

In the Program 5.3 is used a database called "Cabea de Boi" available from the TerraME site. This database contains the terrain digital model (TDM) of a village in Minas Gerais state, Brazil. It is used in several models examples in this tutorial. The cellular space "csQ" has been dynamically created and filled. It is not possible to load or save it in a database.

The *host* and *database* values indicate where the input data is stored. The *dbType* value identifies the database management system (*MySQL*, *PostgreSQL*, etc). The *layer* and *theme* values are the names of the *TerraLib* database layer and theme used as input data. A *layer* is a container of data in *TerraLib*. A *theme* is a set of spatial objects from that *layer*, selected by a restriction. Selection uses a database query over attribute values, spatial relations, and temporal relations. The *select* property contains the names of the cell attributes loaded into the model from the input data set. The property *where* filters the data, as in SQL statements. The *select* and *where* properties are optional.

It's important to emphasize that TerraME currently version no exist tools for create a CellularSpace associated to geographical databases. Thus, is necessary to open a database in *TerraView* application and create a new *layer of cells* from a *layer* exists. For details about creation of a *layer of cells* see [http://www.dpi.inpe.br/ anapaula/plugin_celulas/].

In Program 5.3, the code loads the *CellularSpace* "`csCabecaDeBoi`" using csCabecaDeBoi:load() function from the "`cells`" *theme*, part of the "`cellsLobo90x90`" layer of the "`CabecaDeBoi`" database. For each cell, it loads two attributes: elevation (`height`) and infiltration capacity (`capInf`). It loads in the *CellularSpace* only cells whose "`mask`" attribute value is different from "`noData`".

5.3.1 Referencing cells

A *CellularSpace* has a special attribute called *cells*. It is a one-dimensional table of references for each *Cell* in the *CellularSpace*. The first cell index is 1. Program 5.4 shows how to refer to the i-th *Cell* from a *CellularSpace*.

```
-- c is the seventh cell in the cellular space
c = csCabecaDeBoi.cells[7];
-- Updating the attribute ''infcap'' from the seventh cell
c.infcap = 0;
csCabecaDeBoi.cells[7].infCap = 0
```

Listing 5.4: Examples of references to cells.

5.4 Database management for cell spaces

A *TerraME CellularSpace* provides three functions for database management:

- `load()` - loads the cell attributes from the spatial database;
- `loadNeighborhood()` - loads a neighborhood structure;
- `save()` - stores the cell attribute values in the associated *TerraLib* database

The Program 5.5 shows how these functions are invoked for the `csCabecaDeBoi` *CellularSpace*.

```
--Defines a TerraLib cellular space
csCabecaDeBoi = CellularSpace {
  ...}
--Loads a TerraLib cellular space
csCabecaDeBoi:load();
--Loads a Moore Neighborhood
CreateMooreNeighborhood(csCabecaDeBoi);

for time = 1, 10,1 do
  ...
  csCabecaDeBoi:save( time, "sim", {"water"});
end
```

Listing 5.5: An example about loading and saving cellular spaces in *TerraME*

The `load()` function simply loads a previously defined cellular space in memory (see in Program 5.3 an example). The CreateMooreNeighborhood function creates Moore neighborhood that comprises the eight cells surrounding a central cell. The user can create your own neighborhood structures using the TerraView application, including a generalized proximity matrix (GPM) where each cell has a different neighborhood [29]. For this, it is necessary to use other functions explained in section 5.7.

The syntax of the `save` function is `save (time, themeName, attrNameTable)`. The function uses the value `time` as the data timestamp. It stores data in the new *TerraLib theme* that received as name the union of `themeName` and `time`. It also saves the cell attributes in the *theme* table (themeName_time) with name `attrName1`, `attrName1`, ..., `attrNameN`. If the third value is empty or a *nil* value, all cell attributes will be saved.

The `save(...)` function also creates a *view* named `Result` in the *TerraLib* database. It inserts in this view a theme containing the saved data. In code shown in Program 5.5, at each simulation step, it adds a new *theme* to this *view* to store the current "`water`" attribute value. This attributes are saved in the *themes*: "`sim_1`", "`sim_2`", "`sim_3`", and so on.

5.5 The Cell Type

A *Cell* represents a spatial location, its properties, and its nearness relationships. A *Cell* is a *table* that includes persistent and runtime attributes. The persistent attributes are loaded from and saved to the database. The runtime attributes exist only in memory during the model execution. A *Cell* value has two special attributes: `latency` and `past`. The `latency` attribute registers the period of time since the last change in a cell attribute value. It is useful for rules that depend on how long the cell remains in a state. The `past` attribute is a copy of all cell attribute values in the instant of the last change.

For example, Program 5.6 shows the command "if the cell cover attribute is *abandoned* land during 10 year then the cover value transit to *secondary forest*". Program 5.6 also shows a rule for simulating rain in a cell, which adds $2mm$ of water to the past amount of water. In general, dynamic models read values in the past and write values in the present.

```
if( cell.cover == "abandoned" and cell.latency >= 10 ) then
    cell.cover = "secFor"
end
-- Rule for simulating rain
cell.water = cell.past.water + 2;
```

Listing 5.6: An example the of use of the *latency* and *past* Cell attributes.

5.6 Traversing a cell space

TerraME provides a ways for traversing a cellular space. A second-order function (a function that has a function as an argument): `ForEachCell(cs,`

`function())` applies the chosen function to each cell of the cellular space. This function enables using different rules in a cellular space.

Program 5.7 show an example of the use of the `ForEachCell` function in the cellular space `csQ` where there is constant rain (2 mm/hour) during 10 hours. At the end of each iteration, the cell space must be synchronized (this is explained in section 5.8).

```
for time = 1, 10, 1 do
  ForEachCell(
        csQ,
        function(i,cell)
        cell.soilWater = cell.past.soilWater + 2
        return true;
        end
        );
  csQ:synchronize();
end
```

Listing 5.7: An example of the traversal of a cell space

5.7 The Neighborhood type

Each cell has one or more *Neighborhoods* to represent proximity relations. A *Neighborhood* is a set of pairs (*weight, cell*), where *cell* is a neighbor *Cell* and *weight* is the strength of this relationship. There are two ways of creating a neighborhood in TerraME:

- By creating Moore (3x3) neighborhood, using `CreateMooreNeighborhood()` function;
- By loading an existing neighborhood what can be done:
  - Using the `loadGALNeighborhood()` function and a ".gal" extension file, which contains a GPM previously created and saved by the TerraView application;
  - Using the `loadTerraLibGPM()` function to load, directly from a TerraLib database, a GPM previously created and saved by TerraView application.

As seen in section 5.4, the CreateMooreNeighborhood function creates Moore neighborhood that comprises the eight cells surrounding a central cell. By default, TerraME provides a Moore neighborhood (3x3). This function has two parameters, `CreateMooreNeighborhood(cs,index)`, the name of cellular space and the neighborhood identifier or index. The cs

parameter is the name of the cellular space. The index parameter can be a string or a number. If this index is nil value, the TerraME admits by default the number one. The index parameter becomes important if the user intends to use more than one neighborhood. The code in Program 5.5 shows an example of the use this function without the index parameter.

In the latter case, the neighborhood is created from a generalized proximity matrix or GPM [29] generated by the TerraView application. TerraView has facilities for creating these types of flexible neighborhoods. Please refer to the TerraLib documentation for more details on GPMs. The parameter of `loadGALNeighborhood` function is the directory that contains the file (for example, "c:\\gpm_file.gal"). The parameter of `loadTerraLibGPM` function is a name or a number that identify the neighborhood in the TerraLib database. Thus as in the `CreateMooreNeighborhood` function, this parameter becomes important if exist more than one neighborhood. Program 5.8 show an example of the use theses functions.

```
--Defines a TerraLib cellular space
csCabecaDeBoi = CellularSpace {...}
--Loads a TerraLib cellular space
csCabecaDeBoi:load();

--Loads a neighborhood GPM of the file
cs:loadGALNeighborhood("c:\\neighborhood.gal")

--Loads a neighborhood GPM of the TerraLib database
cs:loadTerraLibGPM("2")
```

Listing 5.8: An example of neighborhood loading.

We can operate on the neighbors of each cell using the function `ForEachNeighbor(cell, function(), index)`, as shown in Program 5.10. `ForEachNeighbor` receives a function as parameter and traverses the i-th *Neighborhood* of a `Cell` applying this function to all cells in it. The index parameter is the same to index parameter of the function that create a neighborhood. Therefore, this parameter is defined as "1" by default of the TerraME.

In Program 5.9, the index parameter has been defined as "v1". The variable weight received as parameter registers the intensity of the neighborhood relationship between the cell and its current neighbor. In this simple example, the weight of all neighbors of each cell is printed.

```
--Defines a TerraLib cellular space
csCabecaDeBoi = CellularSpace {...}
--Loads a TerraLib cellular space
csCabecaDeBoi:load();
```

```
--Loads a Moore Neighborhood
CreateMooreNeighborhood(cs,"v1")

ForEachCell(csQ,
  function (cell)
    ForEachNeighbor(cell,
          function(cell, neigh, weight)
            print(weight)
          end,
          "v1");
    return true
    end
);
```

Listing 5.9: - An example of traversing a neighborhood.

5.8 Synchronizing a cell space

TerraME keeps two copies of a cellular space in memory: one stores the past values of the cell attributes, and another stores the current (present) values of the cell attributes. The model equations must read (the right side of the equation rules) the attribute values from the past copy, and must write (the left side of the equation rules) the attributes values to the present copy of the cellular space. At the appropriated moment, it will be necessary to synchronize the two copies of the cellular space, copying the current attribute values to the past copy of the cellular space. Figure 5.3 shows how synchronization works.

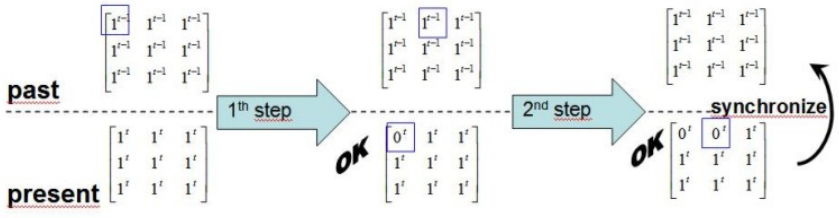


Figure 5.3: Synchronizing a cell space in TerraME.

Synchronization should occur after each iterationsimulation step which effects should influences the present and future behavior of the model. For example, in the "Game of Life" source code, Program 5.10, after traversal of all cells, we have a "present" cell space which is different from the "past"

cell space. Before the next iteration, it is necessary to synchronize the cell spaces. As a good modelling practice, in a neighborhood based rule, the modeller should only update the attributes of the central cell. The neighbor's attributes are read-only. *The flow of information is always from the neighbors to the central cell.*

5.8.1 Conway's Game of Life using TerraME

The following is shown TerraME code of the Conway's Game of Life. The game uses on a field of cells, each of which has eight neighbors. A cell is occupied or empty. The rules for deriving a generation from the previous one are these:

- If an occupied cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, the organism dies (0, 1: of loneliness; 4 to 8: of overcrowding).
- If an occupied cell has two or three neighbors, the organism survives to the next generation.
- If an unoccupied cell has three occupied neighbors, it becomes occupied.

The code shown in Program 5.10 starts by creating a Moore neighborhood. Then it iterates until a final time. At each iteration, it traverses the cell space. For each cell, it applies Conway's rules. Note that it uses the cell's *past* value as input. Then it updates the present value of the cell. Finally, the cell space is synchronized.

```
csQ = CellularSpace {
          ...}
csQ:load();
CreateMooreNeighborhood(csQ);

FINAL_TIME = 10

for time = 1, FINAL_TIME, 1 do
  ForEachCell(csQ,
       function(i, cell)
      count = 0;
      ForEachNeighbor(cell,
        function(cell, neigh)
            if (neigh.past.value == 1 and cell~=neigh)
          then count = count + 1;
           end
        end
      ); -- ForEachNeighbor
      -- apply Conway's rules
```

```
      if (cell.past.value == 1) and
        ((count < 2) or (count > 3)) then
        cell.value = 0 -- cell dies
      end
      if (cell.past.value == 0) and
        (count == 3) then
        cell.value = 1 -- cell lives
      end
       return true
      end
    ); -- ForEachCell
csQ:synchronize();
csQ:save(time,"Game",{"value"});
end -- for time
```

Listing 5.10: Conway's Game of Life

5.9 Exercises

5.9.1 Exercise 1

Add the code lines bellow at the end of the code shown in Program 5.1 and run the model. Explain the output printed on the screen.

```
print("Location:  "..loc.cover..",
"..loc.distRoad..", "..loc.distUrban, loc:reset());
print("Location:  "..loc.cover..", "..loc.distRoad..",
"..loc.distUrban);
```

5.9.2 Exercise 2

Type the following two lines at the end of the code shown in Program 5.2 and run the model. Please, explain the model output.

```
m = MyLoccover = "deforested", distRoad = 0.8, distUrban = 6 ;
print( l );
print( "Location:  "..l.cover..", "..l.distRoad..",
"..l.distUrban..", "..l.desfPot)
print( m );
print( "Location:  "..m.cover..", "..m.distRoad..",
"..m.distUrban..", "..m.desfPot)
```

5.9.3 Exercise 3

Using the TerraLib database "cabeaDeBoi.mdb" answer the questions bellow:

1. Comment the line where = "mask <> 'noData'". Type the following lines at the end of the code shown in Program 5.3 and run the resulting model.

```
print("Size:  "..#csCabecaDeBoi.cells);
print("Size:  "..#cs.cells);
```

2. Assign the complete path for the file "cabecaDeBoi.mdb" in your hard disk to the property database from the cellular space "csCabecaDeBoi" and run the model. Remember to use double bars "\\" for file paths specification.
   For instance, database = "c:\\TerraME\\cabecaDeBoi.mdb".

3. Uncomment the line where = "mask <> 'noData'" and run the model.

4. Type the lines bellow at the end of the model and run it.

```
c = csCabecaDeBoi.cells[7];
print("Cell:  "..c.x.."", "..c.y..", "..c.height);
c = cs.cells[4];
print("Cell:  "..c.x.."", "..c.y..", "..c.soilType);
cs.cells[4].soilType = 10;
print("Cell:  "..c.x.."", "..c.y..", "..c.soilType);
```

5. Explain the results from the above experiments.

5.9.4 Exercise 4

Looking the examples in Programs 5.5, 5.6, 5.7 and Figure 5.3 develop and execute a model which:

- Loads the cellular space "csQ" from the Theme "cells" in "cabecaDeBoi.mdb" TerraLib geographical database.
- Runs from time 1 to time 10 and, at each time step, accumulates 2 mm of water in the attribute "soilWater" of each cells.
- At ech time step, saves the complete cellular space in the database.

Use the application TerraView to see the model outcomes.

5.9.5 Exercise 5

As shown in Program 5.9, loads the cellular space "csCabecaDeBoi" from the spatial database "cabecaDeBoi.mdb", creates a Moore neighborhood to it, and traverse the neighborhood of each cell printing the strength of their vicinity relations.

5.9.6 Exercise 6

Load the "gameoflife.mdb" TerraLib database as shown bellow. Each cell will have just one attribute called "state".

```
DATABASEDIR = "C:\\TerraME\\Database\\"
csQ = CellularSpace{
dbType = "ADO";
database = DATABASEDIR.."gameoflife.mdb",
theme = "gameoflife",
}
csQ:load();
```

Implement the Conway's Game of Life as in Figure 15 changing in the code the variable "value" by the variable "state" loaded from the database. For instance, the line cs:save(...) will become csQ:save(time,"Game","state"). Run the model. Use the TerraView application to open the database "gameoflife.mdb" and explore the view "Results". It should have 10 themes: "Game1", "Game2",..., "Game10". Edit the legend for one of these themes. The legends should be based on the attribute "state" from the table "Game#.state".

5.9.7 Exercise 7

Let "csQ" be a 10 by 10 cellular space defined as in code bellow. It has been dynamically created. Can it be saved in the geographical database?

```
math.randomseed( os.clock() );
-- Dynamically creates a cellular space
SIZE = 10;
ALIVE_PERCENTAGE = 0.2;
csQ = CellularSpace{ database = "", theme = ""}
for i = 1, SIZE, 1 do
for j = 1, SIZE, 1 do
local c = Cell{ state = 0 }
if( math.random( ) < ALIVE_PERCENTAGE ) then c.state = 1;
 end
c.x = i;
c.y = j;
csQ:add( c );
end
end
```

Chapter 6

Several examples of rain drainage models in TerraME

This section is divided in four different rain drainage models. They vary from a simple non-spatial model to a spatial model integrated in a geographic database.

6.1 The "Hello World" model

The simplest rain drainage model is a non-spatial model that considers all terrain as point in the space (see Figure 6.1). The water is a continuous variable Q that collects the rain input flow. The drainage is proportional to Q, where K is the flow coefficient constant. It follows that $\Delta Q_t = 2 - K * \Delta Q_{t-1}$, and $Q = \sum_0^t \Delta Q_t$. For a constant rain, Figure 6.2 shows the simulation results. For each time instant, it indicates the input rain flow (*rain*), the water in the system (Q), and the output flow (*drainage*). The model reaches a steady state after 10 minutes in the simulation clock. Program 6.1 presents the TerraME source code for this model.

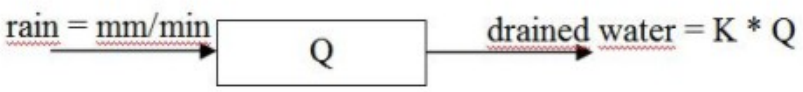
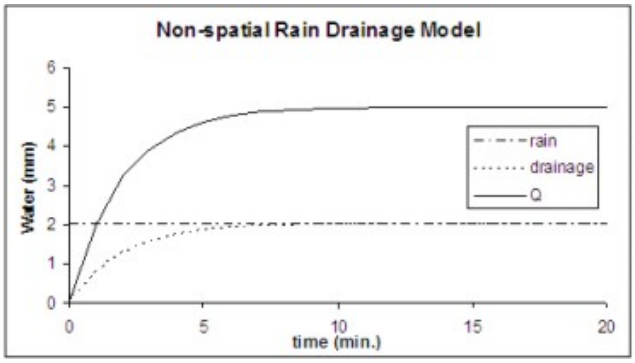


Figure 6.1: A non-spatial rain drainage model.



time	rain	drainage	Q
0	2	0.0	0.0
1	2	0.8	2.8
2	2	1.3	3.3
3	2	1.6	3.6
4	2	1.7	3.7
5	2	1.8	3.8
6	2	1.9	3.9
7	2	1.9	3.9
8	2	2.0	3.9
9	2	2.0	3.9
10	2	2.0	3.9
11	2	2.0	3.9
12	2	2.0	3.9
13	2	2.0	3.9
14	2	2.0	3.9
15	2	2.0	3.9
16	2	2.0	3.9
17	2	2.0	3.9
18	2	2.0	3.9
19	2	2.0	3.9
20	2	2.0	3.9

Figure 6.2: The amount of water in the system during the simulation.

```
C = 2; -- rain/t
K = 0.4; -- flow coefficient
-- GLOBAL VARIABLES
q = 0; input = 0; output = 0;
-- RULES
for time = 0, 75, 1 do
  -- rain
  input = C;
  -- soil water
  q = q + input - output;
  -- drainage
  output = K*q;
  -- report
  print(time.."\t"..input.."\t"..output.."\t"..q);
end
```

Listing 6.1: TerraME code for the non-spatial rain drainage model.

6.2 Discrete and Continuous Dynamic models

Program 6.1 presents a finite difference model which numerically simulates rain drainage as discrete process. The independent variable "time" is not explicit represented in the model rules. TerraME provides the constructor function "d" for the definition of differential equation based rules, allowing for continuous process simulation. Program 6.2 shows how to describe a continuous rain drainage process in TerraME.

```
C = 2; -- rain/t
K = 0.4; -- flow coefficient
dt = 0.01; -- time increment
-- GLOBAL VARIABLES
q = 0; input = 0; output = 0;
-- RULES
for time = 0, 75, 1 do
  -- rain
  input = d{ function() return C; end, 0, 0, 1, dt };
  -- soil water
  q = d{ function( ) return input - output; end, q, 0, 1, dt }
  -- drainage
  output = d{ function( ) return K*q; end, 0, 0, 1, dt };
  -- report
  print(time..."\t"..input..."\t"..output..."\t"..q);
end
```

Listing 6.2: TerraME code for the non-spatial continuous rain drainage model.

The model in Program 6.2 can be further simplified as shown in Figure 6.3. The constructor "d" has four obligatory parameters and a fifth optional parameter. The first parameter is a function which returns the real number and has the same formula of the differential equation that simulates the process. It has two optional parameters, the independent and the dependent variables of the differential equation, as in Program 6.3. The second parameter is the initial condition of the process been simulated. In Program 6.2, at each time step, the initial conditions are 0, q and 0 for the rain, soil water accumulation and drainage processes, respectively. The third and fourth parameters define the minimum and maximum values of the integration interval. In Program 6.2, each process is simulated from the time 0 to the time 1. In Program 6.3, the soil water accumulation process is simulated from the time 0 to 75. The fifth parameter is the independent variable increment value that should be used in the numeric integration methods used by the TerraME engine to simulate the models. The DELTA TerraME global variable defines the default value goal to 0.2. In Program

6.3 model, the time increment value is 0.1. Change the value of the INTEGRATION_METHOD TerraME global variable, the modeler will select the numeric integration method which will be used for continuous model simulation. The possible methods are: "Euler", "Heum" and "RugeKutta". The default method is "Euler".

```
C = 2; -- rain/t
K = 0.4; -- flow coefficient
dt = 0.01; -- time increment
-- GLOBAL VARIABLES
q = 0; input = 0; output = 0;
-- RULES
dt = DELTA/2;
-- INTEGRATION_METHOD = "Euler";
-- INTEGRATION_METHOD = "Heum";
-- INTEGRATION_METHOD = "RugeKutta";
q = d{ function (t, q) return C - K*q ; end, 0, 0, 75, dt };
print( q );
```

Listing 6.3: Writing differential equations for continuous process simulation in TerraME.

6.3 A simple spatial model

We now consider a 1D model. Space is modeled as a list of locations $Q = q_i | \forall i = 1..N$, where $N = 10$. The model executes the same rules with the same parameters in each space location. The temporal variation of water in each location is equal to the graphic shown in 6.2.

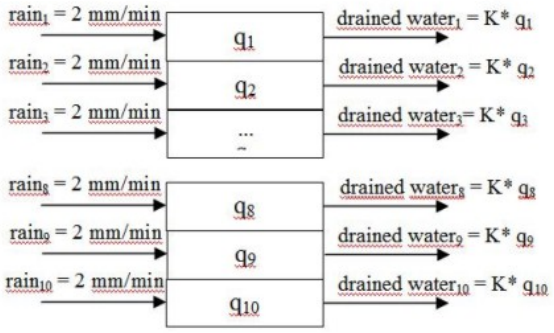


Figure 6.3: A 1D spatial discrete rain drainage model.

```
-- CONSTANTS (MODEL PARAMETERS)
C = 2; -- rain/t
K = 0.4; -- flow coeficient
-- GLOBAL VARIABLES
q = {}; -- a 1D table
-- RULES
for i = 1, 10, 1 do q[i] = 0; end
for time = 1, 20, 1 do
  -- rain and drainage
  for i = 1, 10, 1 do
        q[i] = q[i] + C;
        q[i] = q[i] - K*q[i];
    end
    -- report: soil water (Q)
  print("t: "..time );
  for i = 1, 10, 1 do print("["..i.."]: "..q[i]); end
end
```

Listing 6.4: TerraME source code for a 1D model.

Program 6.4 presents the TerraME source code for the unidimensional spatial drainage model. The variable q is a list, which locations q_i have been initialized with the value 0 (zero). The "`for ...end`" statement from TerraME *Modelling Language* has been used to traverse the list.

6.4 A 2D spatial model

We can now extend the model to a 2D grid. Figure 6.4 shows the conceptual model for a 2D spatial drainage model, using a grid $Q = q_{i,j}|i = 1..nand\forall j = 1..n$.

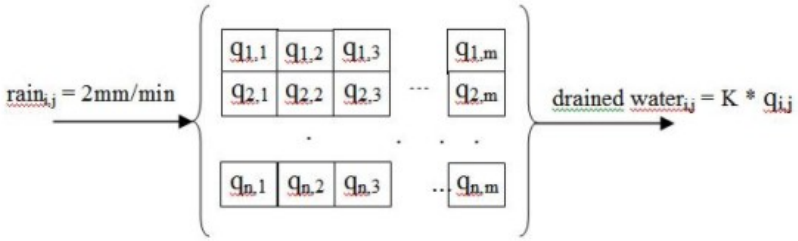


Figure 6.4: A 2D spatial rain drainage model.

The TerraME source code for this model is in Program 6.5. The variable q represents s a bidimensional grid. To traverse the space representation

the modeller has used a block of code containing two nested "`for ...end`" statements.

```
-- CONSTANTS (MODEL PARAMETERS)
C = 2; -- rain/t
K = 0.4; -- flow coefficient
-- GLOBAL VARIABLES
q = {};
-- RULES
for i = 1, 10, 1 do
  q[i] = {};
  for j = 1, 10,1 do
    q[i][j] = 0;
  end
end
for time = 0, 75, 1 do -- rain and drainage
  print("t: "..time );
  for i = 1, 10, 1 do
      for j = 1, 10,1 do
      q[i][j] = q[i][j] + C;
      q[i][j] = q[i][j] - K*q[i][j];
      -- report: soil water (Q)
      print ("i: "..i.."; j: "..j.."): "..q[i][j])
      end
  end
end
```

Listing 6.5: The TerraME source code for the 2D drainage model.

6.5 A spatial model integrated to a geographic database

In previous examples, we have not discussed how to read data. TerraME reads data from a TerraLib spatial database, as described in section 2. Program 6.6 presents the TerraME code of the rain drainage model integrated to a *TerraLib* database.

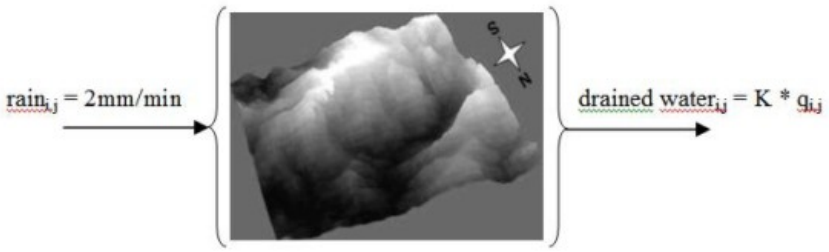


Figure 6.5: The spatial rain drainage model integrated to a geographical database.

The cell space `csQ` is retrieved from a layer in a *TerraLib* geographic database. For this example, we use the "`cabecaDeBoi.mdb`" database, available from the TerraME site. Each cell has an attribute called `soilWater`. The function `load`(), retrieves data and initializes the cells. We use the `ForEachCell` function to traverse the cellular space. The functions `save`() stores the soil water distribution at each simulation time step. A *view* called "`Result`" is created in the database. At each simulation step, it adds a new theme to this view to store the current values of the "`soilWater`" attribute. The reader may use the *TerraView* software to explore the data.

```
-- CONSTANTS (MODEL PARAMETERS)
C = 2; -- rain/t
K = 0.4; -- flow coefficient
FINAL_TIME = 24;

-- PART 1 - Retrieve the cell space from the database
csQ = CellularSpace{
  dbType = "ADO",
  database = "c:\\TerraME\\Database\\cabecaDeBoi.mdb",
  theme = "cells",
  select = { "height", "soilWater" }
}

-- RULES
csQ:load();
CreateMooreNeighborhood(csQ);
csQ:synchronize();

ForEachCell(csQ,
  function(i, cell)
    cell.soilWater = 0;
    return true;
  end);
```

```
for t = 1, FINAL_TIME, 1 do
  -- PART 2: It's raining in the high areas
  ForEachCell(csQ,
    function(i, cell)
      if(cell.height > 200) then
        cell.soilWater = cell.past.soilWater + C;
      end
      return true;
    end);
  csQ:synchronize();

  -- PART 3: create a temporary variable to store the flow
  ForEachCell(csQ,
    function(i, cell)
      cell.flow = 0;
      return true;
    end);

  -- Calculate the drainage and the flow
  ForEachCell(csQ,
    function(i, cell)
      -- PART 4: calculate the drainage
      cell.soilWater = cell.past.soilWater - K*cell.past.
          soilWater;
      -- count the lower neighbors
      countNeigh = 0;
      ForEachNeighbor(cell,
        function(cell, neigh)
         if (cell ~= neigh) and (cell.height >= neigh.height)
                then countNeigh = countNeigh + 1;
       end
      end);

      -- PART 5: calculates the flow to neighbors
      if (countNeigh > 0) then
        flow = cell.soilWater / countNeigh;
        -- send the water to neighbors
        ForEachNeighbor(cell,
          function(cell, neigh)
            if (cell ~= neigh)
            and (cell.height > neigh.height) then
              neigh.flow = neigh.flow + flow;
            end
          end);
      end
    end
  ); --ForEachCell

  ForEachCell(csQ,
    function(i, cell)
      cell.soilWater = cell.flow;
```

```
      return true;
    end);

  csQ:synchronize();
  -- report: soil water
  print("t: "..t);
  if (t == FINAL_TIME) then
    csQ:save(t, "water", {"soilWater"});
  end
end
```

Listing 6.6: The TerraME source code for a spatial model inside a geographical database.

6.6 Exercises

6.6.1 Exercise 1

Implement and run the rain drainage model shown in Program 6.1. When the model reaches the steady state? Run the model for several flow coefficient values = 0, 0.25, 0.5, 0.75, 1. What happens?

6.6.2 Exercise 2

Implement the rain drainage model shown in Program 6.2. Run the model for several flow coefficient values = 0, 0.25, 0.5, 0.75, 1. Compare the outcomes with these produced by the model in Figure 18. With the flow coefficient value = 0.25, run the model for several time increment values, dt = 0, 0.1, 0.001, 1. What can be concluded?

6.6.3 Exercise 3

Implement the model shown in Program 6.3. Run it using different numeric integration methods, INTEGRATION_METHOD = "Euler", "Heum", "RugeKutta" . Explain the model outcomes.

6.6.4 Exercise 4

Implement and run the model shown in Program 6.4. Compare the model results with the ones produced by the model in Program 6.1.

6.6.5 Exercise 5

Implement and run the model shown in Program 6.5. Compare the model outcomes with the ones produced by the model shown in Program 6.4.

6.6.6 Exercise 6

Implement and run the model shown in Program 6.6. Use the TerraView application to explore the model outcomes. Look for the theme "water24" in the view "Result" from the "cabecaDeBoi.mdb" database. Generate a legend for the attribute "water24.soilWater". Explain the model result.

Chapter 7

Several examples of land change models in TerraME

This section presents TerraME examples for a different problem: modelling of land change. We will consider the database "amazonia.mdb", which contains a 100 x 100 km^2 cell space with data related to deforestation in Amazonia. Figure 7.1 shows a picture of the deforestation for each cell. This data is a simplified version of the database used in [30]. The attributes of the cell space are:

- `defor`: percentage of deforestation;
- `pop_dens_96`: population density from 1996 census;
- *pop_tx_urban_96*: urbanization rate from 1996 census;
- *pop_pc_migr_91_96*: migration rate from 1991 to 1996;
- *agr_area_small*: percentage of cultivated area for small farms;
- *agr_area_medium*: percentage of cultivated area for medium farms;
- *agr_area_large*: percentage of cultivated area for large farms;
- *dist_urban_areas*: average distance to urban areas;
- *dist_roads*: average distance to roads;

- *conn_markets_inv_p*: strength of connection to markets
- *clima_humi_min_3_ave*: humidity in the three driest months;
- *clima_precip_min_3_a*: precipitation in three driest months;
- *soils_fert_B1*: average soil fertility
- *prot_all1*: percentage of protected areas in 1996
- *prot_all2*: proposed percentage of protected areas in 2006

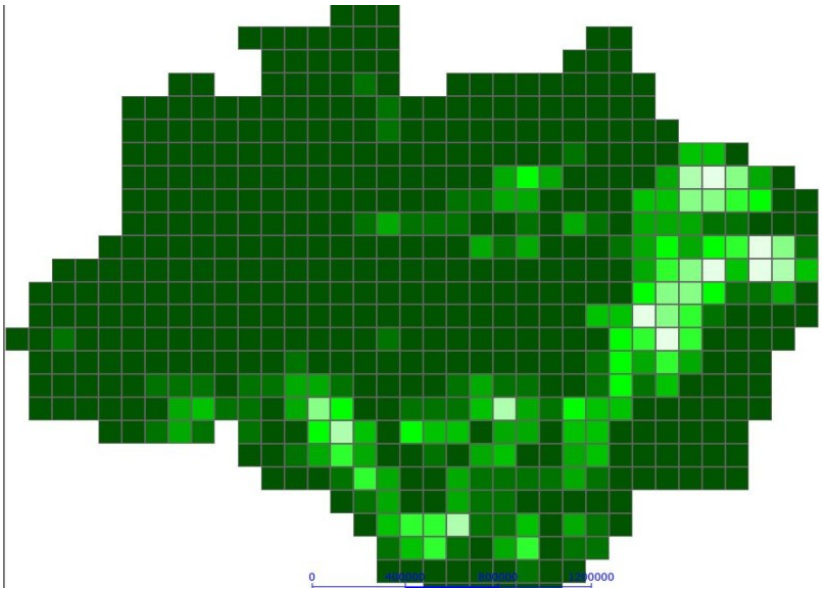


Figure 7.1: A cell space of deforestation in Amazonia.

The model considers a fixed demand for change, which will be allocated spatially. It calculates the potential for change at each cell. Then, it divides the demand as a proportion of the potential of change. We will consider three models: a simple diffusive model, a simple regression model, and a spatial regression model.

7.1 A spatial diffusive model for land change

Consider a spatial model that allocates 30.000 km^2 of deforestation in Amazonia for 10 years. The potential of change for each cell is the average

of neighbour's deforestation. The allocation function uses is proportional to the cell's potential, divided by the total potential for change. The result is shown in Figure 7.2 and the model is shown in Program 7.1. It works as follows:

1. Reads the data from the database (command `csQ` = `CellularSpace`{...);
2. Creates a 3x3 neighbourhood (`CreateMooreNeighbourhood` (`csQ`));
3. Defines a new attribute for potential for change (using the command `ForEachCell`(.. `cell.pot` = 0 ...);
4. Calculate the change potential for each cell. This requires a traversal of the cell space (`for...`). The potential for change for a cell is the average of its neighbour's deforestation.
5. Assign the demand based on the potential for each cell. This needs a second for loop. This loop is inside an allocation loop that considers the case where the change potential for a cell may exceed 100% of deforestation.
6. Syncronize the cell space after each time step and save the last time step.

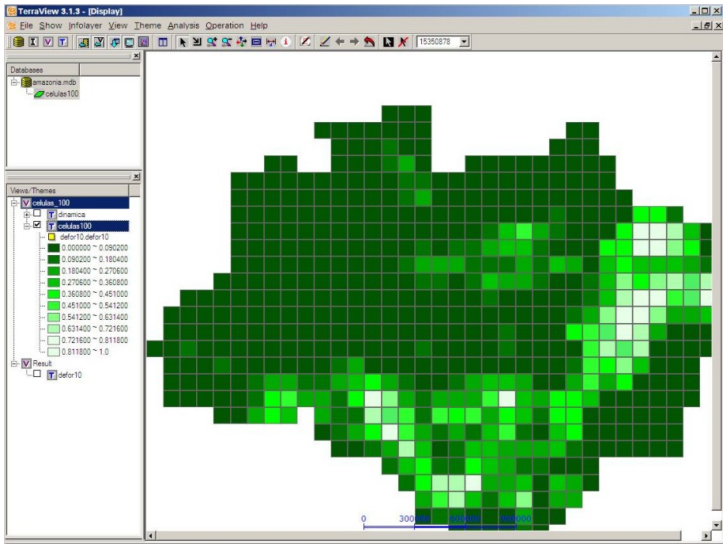


Figure 7.2: Result of the diffusive model after 10 years.

```
-- CONSTANTS (MODEL PARAMETERS)
CELL_AREA = 10000;
FINAL_TIME = 10;
ALLOCATION = 30000;
LIMIT      =    30;
-- GLOBAL VARIABLES
csQ = CellularSpace{
  dbType = "ADO",
  host = "localhost",
  database = "c:\\TerraME\\Database\\amazonia.mdb",
  user = "",
  password = "",
  layer = "celulas100",
  theme = "dinamica",
  select = {"defor"}
}
-- RULES
csQ:load();
CreateMooreNeighborhood(csQ);
csQ:synchronize();

for time = 1, FINAL_TIME, 1 do
  print("t: "..time );

  -- initialize the potential
  ForEachCell(csQ,
    function (i, cell)
      cell.pot = 0;
    end
  );

  total_pot = 0;
  ForEachCell (csQ,
    -- Calculate the change potential for each cell
    function (i, cell)
      countNeigh = 0;
      ForEachNeighbor( cell,
        function(cell, neigh)
        --The potential of change for each cell is
        --The average of neighbor's deforestation
        --Fully deforested cells have no potential
          if (cell.defor < 1.0 ) then
             cell.pot = cell.pot + neigh.defor
             countNeigh = countNeigh + 1
          end
        end
      ); -- ForEachNeighbor
      if(cell.pot > 0 ) then
         -- increment the total potential
         cell.pot = cell.pot / countNeigh;
         total_pot = total_pot + cell.pot;
```

```
      end
    end
  ); --ForEachCell

  -- ajust the demand for each cell so that
  -- the maximum demand for change is 100%
  -- adjust the demand so that excess demand is
  -- allocated to the remaining cells
  -- there is an error limit (30 km2 or 0.1%)
  total_demand = ALLOCATION;
  while (total_demand > LIMIT ) do
    print("total_demand: "..total_demand );
    ForEachCell (csQ,
      function (i, cell)
        if (cell.pot > 0) then
           prop_cell = cell.pot/total_pot
           newarea = prop_cell* total_demand
           cell.defor = cell.past.defor +
                newarea/CELL_AREA
          if (cell.defor >= 1) then
             total_pot= total_pot - cell.pot
             cell.pot  = 0;
             excess= (cell.defor -1)*CELL_AREA
             cell.defor = 1
          else
             excess = 0;
          end
           -- adjust the total demand
           total_demand = total_demand - (newarea - excess)
        end
      end
    ); -- ForEachCell
  csQ:synchronize();
  end

  if (time == FINAL_TIME) then
    csQ:save( time, "defor1", {"defor"} );
  end
end
```

Listing 7.1: The TerraME source code for a simple diffusive land change model.

7.2 A regression model for land change

We will now consider a regression model based on three driving forces: distance to urban centers, connection to markets, and protected areas. The potential for change is based a linear regression between the cell's current deforestation and the expected deforestation, as follows:

- Calculate the expected deforestation as

$$expected = -0.45 \times \log(D) + 0.26 \times (C) - 0.14 \times (P) + 2.313$$

  where:

  $D = distancetourbanareas$, $C = connectiontomarkets$ and $P = protectedareas$

- Calculate the potential for change as

$$cell.pot = expected - cell.defor$$

- Normalize the potentials (since there may be negative potentials) and allocate 30.000 km^2 for 10 years.

This model is a simplified version of the detailed deforestation model developed by [26]. Please see that document for details on the model. The model code is shown in Program 7.2 and the result in Figure 7.3.

```
-- CONSTANTS (MODEL PARAMETERS)
CELL_AREA = 10000;
FINAL_TIME = 10;
ALLOCATION = 30000;
LIMIT      =    30;
-- GLOBAL VARIABLES
csQ = CellularSpace{
  dbType = "ADO",
  host = "localhost",
  database = "c:\\TerraME\\Database\\amazonia.mdb",
  user = "",
  password = "",
  layer = "celulas100",
  theme = "dinamica",
  select= {"defor", "dist_urban_areas",
           "conn_markets_inv_p", "prot_all2" }
}
-- RULES
csQ:load();
CreateMooreNeighborhood(csQ);
csQ:synchronize();

for time = 1, FINAL_TIME, 1 do
  print("t: "..time );
  -- initialize the potential
  ForEachCell (csQ,
    function (i, cell)
      cell.pot = 0
    end
```

```
);

total_pot = 0;
ForEachCell (csQ,
  function (i, cell)
    -- The potential for change is the residue of a
    -- linear regression between the cell's
    -- current and expected deforestation
    -- according to the following model:
    if (cell.defor < 1.0) then
      expected =
      - 0.45*math.log10 (cell.dist_urban_areas)
      + 0.26*cell.conn_markets_inv_p
      - 0.14*cell.prot_all2
      + 2.313;
      if (expected > cell.defor) then
        cell.pot = expected - cell.defor
        total_pot = total_pot + cell.pot
      end
    end
  end
); -- ForEachCell

--Adjust the demand so that excess demand is
--Allocated to the remaining cells in an error limit (0.1%)
total_demand = ALLOCATION
while (total_demand > LIMIT ) do
  ForEachCell (csQ,
    function (i, cell)
      if (cell.pot > 0) then
        prop_cell = cell.pot/total_pot
        newarea = prop_cell* total_demand
        cell.defor = cell.past.defor +
            newarea/CELL_AREA
         if (cell.defor >= 1) then
           total_pot = total_pot - cell.pot
           cell.pot  = 0
           excess = (cell.defor - 1)*CELL_AREA
           cell.defor = 1
         else
           excess = 0;
         end
       -- adjust the total demand
       total_demand=total_demand -newarea +excess
       end
    end
  ); --ForEachCell
  csQ:synchronize();
end

if (time == FINAL_TIME) then
  csQ:save( time, "defor2_", {"defor"} );
```

```
  end
end
```

Listing 7.2: The TerraME source code for a linear regression land change model.

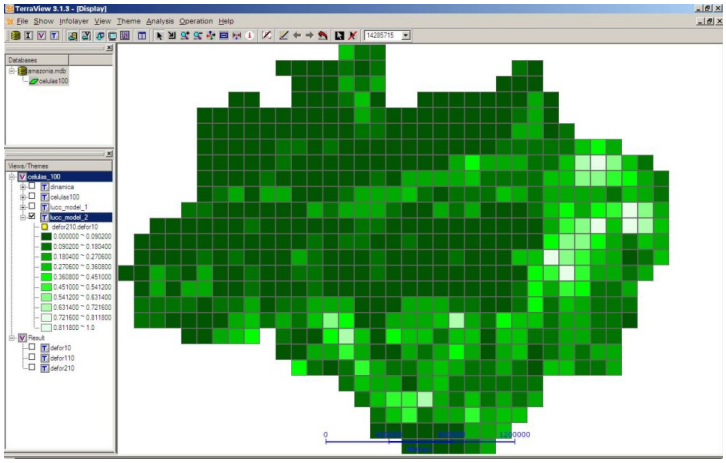


Figure 7.3: Result of land change model based on linear regression.

7.3 A combined diffusive/regression model

We will now consider a regressive and difusive model based on four driving forces: the deforestation on the neighbours, distance to urban centres, connection to markets, and protected areas. For a detailed discussion of the impact of neighbours on deforestation, see [26]. The potential for change is based on the residues of a spatial regression between the cell's current deforestation and the expected deforestation according to the following model:

- Calculate the expected deforestation as

  $$expected = 0.73 \times \log_{10}(AVERAGE(N)) - 0.15 \times \log_{10}(D) + 0.05 \times (C) - 0.07 \times (P) + 0.7734$$

  where $N = neighbordeforestation$, $D = distancetourbancentres$, $C = connectiontomarkets$ and $P = protectedareas$

- Calculate the potential for change for each cell as

$$potential = expected - deforestation$$

- Allocate 30.000 km2 for 10 years for all cells with positive potentials. Note there may be negative potentials, which are cells with more deforestation than expected. In this case, there is no change for the cell.

This model is a simplified version of the detailed deforestation model developed by [26]. The model code is shown in Program 7.3 and the result in Figure 7.4.

```
-- CONSTANTS (MODEL PARAMETERS)
CELL_AREA = 10000
FINAL_TIME = 10
ALLOCATION = 30000
LIMIT      = 30

-- GLOBAL VARIABLES
csQ = CellularSpace{
  dbType = "ADO",
  host = "localhost",
  database = "c:\\TerraME\\Database\\amazonia.mdb",
  user = "",
  password = "",
  layer = "celulas100",
  theme = "dinamica",
  select= {"defor", "dist_urban_areas",
           "conn_markets_inv_p", "prot_all2" }
}

-- RULES
csQ:load();
CreateMooreNeighborhood(csQ);
csQ:synchronize();

for time = 1, FINAL_TIME, 1 do
  print ("t: "..time);
  -- initialize the potential
  ForEachCell (csQ,
    function (i, cell)
      cell.pot = 0
      cell.ave_neigh = 0
    end
  );

  ForEachCell (csQ,
    function (i, cell)
      -- Calculate the average deforestation
```

```
    countNeigh = 0;
    ForEachNeighbor( cell,
      function(cell, neigh)
      --The potential of change for each cell is
      --the average of neighbors' deforestation.
        if (cell.defor < 1.0 ) then
          cell.ave_neigh = cell.ave_neigh + neigh.defor
          countNeigh = countNeigh + 1
        end
      end
    ); -- ForEachNeighbor
    -- Find the average deforestation
    if(cell.defor < 1.0 ) then
      cell.ave_neigh = cell.ave_neigh / countNeigh
    end
  end
); -- ForEachCell

total_pot = 0;
ForEachCell (csQ,
  function (i, cell)
    -- Potential for change
    if (cell.defor < 1.0) then
      expected =  0.73*cell.ave_neigh
        - 0.15*math.log10(cell.dist_urban_areas)
        + 0.05*cell.conn_markets_inv_p
        - 0.07*cell.prot_all2 + 0.7734;
      if (expected > cell.defor) then
        cell.pot = expected - cell.defor
        total_pot = total_pot + cell.pot
      end
    end
  end
); -- ForEachCell

-- adjust the demand for each cell
total_demand = ALLOCATION
while (total_demand > LIMIT ) do
  print("total_demand: "..total_demand );
  ForEachCell (csQ,
    function (i, cell)
      if (cell.pot > 0) then
        prop_cell = cell.pot/total_pot
        newarea = prop_cell* total_demand
        cell.defor = cell.past.defor +
                                        newarea/CELL_AREA;
          if (cell.defor >= 1) then
            total_pot= total_pot -cell.pot
            cell.pot  = 0
            excess = (cell.defor - 1)*CELL_AREA
            cell.defor = 1
          else
```

```
            excess = 0;
          end
        -- adjust the total demand
        total_demand = total_demand - newarea + excess;
      end
    end
  ); --ForEachCell
  csQ:synchronize();
end

if(time == FINAL_TIME) then
  csQ:save( time, "defor3_", {"defor"} );
end
end
```

Listing 7.3: Code for land change model based on spatial regression.

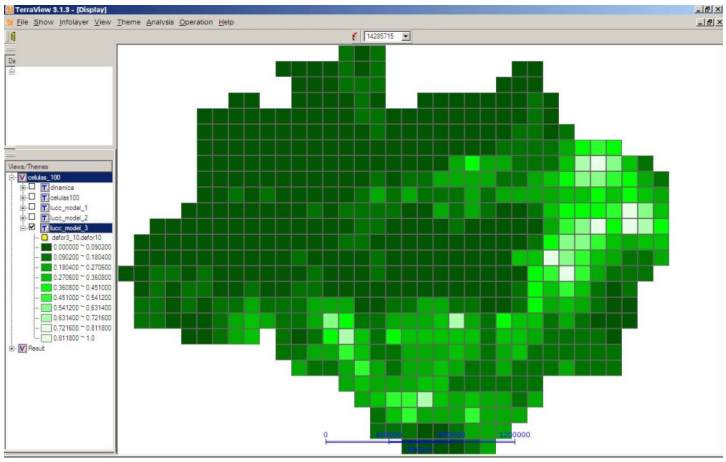


Figure 7.4: Result of land change model based on spatial regression.

7.4 Exercises

7.4.1 Exercise 1

Implement and run the model shown in Program 7.1. Use the TerraView application to explore the model outcomes in the "amazonia.mdb" geographical database. Look for the themes "defor#" in the view "Result". Explain the observed results.

7.4.2 Exercise 2

Implement and run the model shown in Program 7.2. Use the TerraView application to explore the model outcomes in the "amazonia.mdb" geographical database. Look for the themes "defor2_#" in the view "Result". Explain the observed results.

7.4.3 Exercise 3

Implement and run the model shown in Program 7.3 . Use the TerraView application to explore the model outcomes in the "amazonia.mdb" geographical database. Look for the themes "defor3_#" in the view "Result". Explain the observed results.

7.4.4 Exercise 4

Compare the outcomes produced by the models shown in Programs 7.1, 7.2 and 7.3.

Chapter 8

The Trajectory type

The *trajectory* function is useful to reproduce spatial patterns or represent process preferential directions (anisotropy). This function is also useful to define change suitability surfaces, which associate each Cell to a real number that indicates how prone the *Cell* is to specifics types of change (forest to pasture, pasture to abandonment, pasture to urban, etc).

The trajectory function is defined by three parameters:

- A *CellularSpace* over which the trajectory will take place;
- A *function* that includes cells in the trajectory;
- A *function* that orders the cells included in the trajectory.

The first *function* receives a *Cell* as parameter and returns a Boolean value (*true or false*). It is used to filter the *Cells*. If this *function* returns *true*, the cell is included in the trajectory. The second *function* receives two *Cell* values as parameters and returns true if the first one is greater than the second. If the second *function* is not defined, the *Cells* are traversed from North to the South and from West to the East. If both f*unctions* are not defined, all *Cells* are included in the trajectory.

Program 8.1 shows an example of *trajectory* function useful to simulate the deforestation process. A trajectory over the *CellularSpace* (`csQ`) is defined by two functions. The first function select only cells whose land cover is "forest". The second orders the *Cells* according to their distance to the nearest road, making *Cells* closer to roads more suitable to change. Figure 8.1 illustrate two different trajectories, the former is defined as in Program 8.1, and the latter has a different ordering function: "function(

c1, c2) return c1.distUrbanCenter < c2.distUrbanCenter; end". Light gray cells are traversed before dark gray cells.

```
-- Define a CellularSpace
csQ = CellularSpace{
...}

csQ:load()

--Define a trajectory
it = Trajectory{
    csQ,
    function(cell)
return cell.cover == "forest"
end,
    function( c1, c2 )
return c1.dist_roads > c2.dist_roads
end
}
```

Listing 8.1: Defining a Trajectory in TerraME Modeling Language.

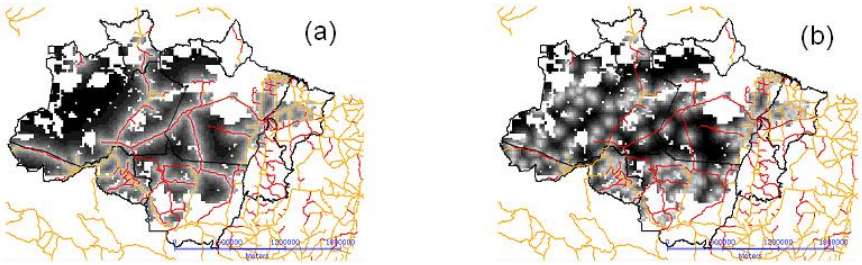


Figure 8.1: Change potential surfaces defined through Trajectory functions based on (a) distance to road or (b) distance to urban centers. The gray scale surface reflects the potential for change of each cell: dark gray means low potential for change and light gray means high change potential.

Program 8.2 shows how to use *Trajectories* values. Instead of pass a *CellularSpace* value as the first parameter in a *ForEachCell* function call, one may pass a Trajectory. In this example, the trajectory "it" defined in Program 8.1 is used to simulate deforestation around roads.

```
demandByYear = { 3, 5, 6, 7, 8, 4, 3, 3, 3, 3}

for time = 1, 10, 1 do
  demand = (demandByYear[ time ]/100) * #csQ.cells;
```

```
  while (demand >= 1) do
      ForEachCell(
        it,
        function(i, cell)
          if( demand >= 1) then
            cell.cover = "deforested";
            demand = demand - 1;
          else
            return false;
          end
          return true;
        end
      );
  end
  csQ:synchronize();
end
```

Listing 8.2: An example of spatial pattern simulation using the Trajectory type.

8.1 Exercises

8.1.1 Exercise 1

Based on Programs 8.1 and 8.2 develop a deforestation model over the "amazonia.mdb" geographical database. The deforestation rate by year is given by demandByYear = 3, 5, 6, 7, 8, 4, 3, 3, 3, 3. Use a Trajectory to simulate the deforestation cause only by urban centers expansion. Run the model and use the TerraView application to explore the model results.

8.1.2 Exercise 2

Based on Figures 8.1 and 8.2 develop a deforestation model over the "amazonia.mdb" geographical database. The deforestation rate by year is given by demandByYear = 3, 5, 6, 7, 8, 4, 3, 3, 3, 3. Use a Trajectory to simulate the deforestation around the roads. Run the model and use the TerraView application to explore the model results.

Chapter 9

Nested cellular spaces

TerraME do not have nested cells concept but only scales are nested (scale or Environment), and all its contents: cell spaces, automata and schedulers. TerraME provides necessary tools to create any spatial structure using the neighbor concept: the cells inside a cellular environment could have neighbor in any other cellular environment

Suppose the cellular spaces "cs1" and "cs2" as shown in Figure 9.1. Each cellular space "cs1" has 4 cells resolution finest in cellular space "cs2". A one-way (cs1 to cs2) was used to implement the concept of nesting of cells and allow downscaling. If "cs2" cells needed to know your "parent" cell to allow "upscaling", then a two-way vicinity should be used, including the cells "cs1" as neighboring of "CS2".

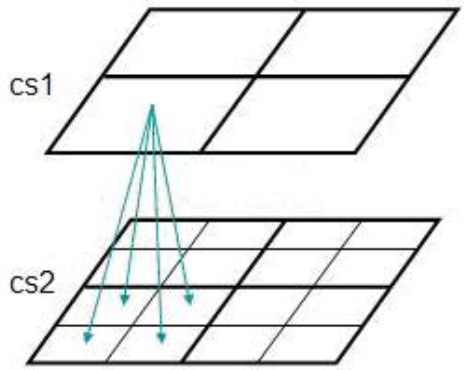


Figure 9.1: Nested cells by neighborhood relationships.

Functionality nested cellular spaces can easily be used in the models through the combination of functions "ForEachCell" and "ForEachNeighbor, as in Program 9.1.

```
ForEachCell( cs1,
  function(i, cell)
    ForEachNeighbor( cell,
         function( cell, neigh, weight )
        -- neigh is a nested_cell
        -- insert the code
      end,
      índex
      -- refers to vicinity index used
         -- to implement the structure
  end
);
```

Listing 9.1: How to traverse nested cellular spaces.

The source code in Program 9.2 shows how neighborhoods can be created respecting the Cartesian (structure) coordinate system allocated by the TerraLib GIS Library. In the code, a neighborhood Moore is created for the cell space received as parameter.

```
-- Creates a Moore neighborhood for each cell
function CreateMooreNeighborhood( cs, name )
  for i, cell in ipairs( cs.cells ) do
    local neigh = Neighborhood();
    local lin = -1;
    while ( lin <= 1 ) do
      local col = -1;
      while ( col <= 1 ) do
        -- add neighbor
        local index = TeCoord{ x = (cell.x + col), y =          (
            cell.y + lin)};
          neigh:addCell( index, cs, 1/9 );
        -- weight = 0.111111 (9 neighbors)
        col = col + 1;
      end
      lin = lin + 1;
    end
  cell:addNeighborhood( neigh, name );
  end
end
```

Listing 9.2: How to define neighborhoods around geographical locations.

TerraME provides some other functions for neighborhood definition and for cellular space coupling through neighborhood relationships:

- Create3x3StationaryNeighborhood(cs,filterF, weightF, name) creates a 3x3 stationary neighborhood for each "cs" cellular space cell.

- CreateMxNStationaryNeighborhood(M, N, cs,filterF, weightF, name) creates a M (collumns) by N (rows) stationary neighborhood for each cell in the "cs" cellular space.

- SpatialCouppling(M, N, cs1,cs2, filterF, weightF, name) creates a M (collumns) by N (rows) stationary neighborhood between cells from two different cellular spaces, the "cs1" and "cs2", one-way neighborhood relations are defined from "cs1" to "cs2".

The parameters "cs", "filterF(cell, neigh) → Boolean", "wheighF(cell, neigh) → Real" and "name" are, respectively, the cellular space over which the neighborhood will be created, a Boolean function for filtering cells that will take place in the neighborhood based on their properties and on the properties each one of their neighbors, a numeric function that assigns a weight to each one-way neighborhood relationship and a identifier (or index) for the neighborhood been created. Program 9.3 illustrate the use of the function "Create3x3StationaryNeighborhood" for building a neighborhood useful for drainage simulation: only lower neighbors are included in the cell neighborhood and the weight these relationships are calculate based on a slope metric. This weigh could be used to direct flow of water among the lower neighbors.

```
-- Creates a 3x3 Neighborhood based on the cell "slope"
-- only lower neighbors are considered
-- Creates a 3x3 Neighborhood based on the cell "slope"
Create3x3StationaryNeighborhood(
  csQ,
  function(cell,neigh)
    return neigh.altimetria < cell.altimetria;
  end,
  function(cell, neigh)
    return (cell.altimetria - neigh.altimetria)/
        (cell.altimetria + neigh.altimetria);
  end,
  "slope"
);
```

Listing 9.3: How to define neighborhoods around geographical locations.

9.1 Exercises

9.1.1 Exercise 1

Using the "cabecaDeBoi.mdb" geographical database load a cellular spaces from the theme "cells" and build a 4 by 4 neighborhood in which the

weights of the vicinity relations depends on the terrain slope, as in Program 9.3. Then, traversse the cellular space printing the weight calculated for each neighbor relation.

9.1.2 Exercise 2

Create a new layer of cells in the "cabeaDeBoi.mdb". The cell resolution should be 180 by 180 meters. Use the TerraView plugin called "fill cell" to generate the new attribute "height" for each cell. Use the attribute "height" from the "cells" theme as input data and the operation "average value" to calculate the value the value of the new attribute in each space location. Create the theme "myCells" to visualize the terrain represented by the new layer of cells. Then, load two cellular space from the themes "cells" and "myCells". Couple both cellular space building a nested cellular space as shown in Figure 9.1, where "cs1" is "cells" and "cs2" is "myCells". Finally, traverse the cellular space "cells" printing the height of each neighbor as cell from "cells" has in "myCells".

Chapter 10

Hybrid Automata

A hybrid automaton is a dynamical system whose state has both a discrete, which is updated in a sequence of steps, and a continuous component, which evolves over time, so it can be viewed as an infinite-state transition system [31]. The concept of hybrid comes from the bivalence between discrete and also continuous [1]. According to [14], *a hybrid system is a dynamical system with both discrete and continuous components.* A modern automobile engine, for example, whose fuel injection (continuous) is regulated by a microprocessor (discrete), is a hybrid system.

A Hybrid Automaton H can be defined as follows:

- A finite set of real variables $X = x1, x2, \ldots xn$. n is the dimension of H.
- A finite directed graph $G = (V, E)$ with the set of vertices V called the States, and the set of edges E called the Jumps. Each edge Jump connects a source State to a target State. It also has a conditional rule associated to it. If this condition is evaluated as true, then the automaton internal discrete state will change from the source State to the target State.
- A set of Flow rules assigned to each State. When a Flow rule is evaluated it may change the automaton internal continuous state, i. e., the values of the real variables $x1, x2, \ldots xn$.

The most common example is the temperature control. Figure 10.1 models it. Variable x represents the temperature. In control mode OFF, the heater is off, and the temperature falls according to the flow condition $x = -0.1x$. In control mode ON, the heater is on, and the flow condition is

$x = 5 - 0.1x$. The initial state is OFF, and the temperature is 20 degrees. The jump condition is $x < 19$, in which the heater is turned ON as soon as the temperature falls below 19 degrees. The invariant condition $x >= 18$ says that at the latest the heater will go ON then the temperature falls to 18 degrees.

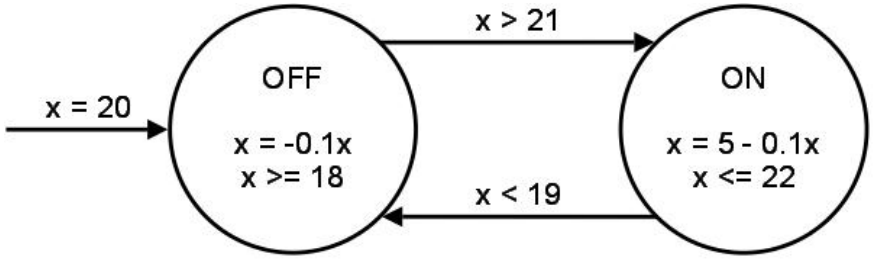


Figure 10.1: Nested cells by neighborhood relationships.

In TerraME an *Automaton* is defined as in Program 10.1. The automaton "at" has two possible internal discrete states: "first" and "second". Each State can have any number of Jump and Flow rules. *Automaton*, *State*, *Jump* and *Flow* values have an identifier, i. e., a obligatory property named "id" which identifies they univocally inside the whole model. This is very useful for model verification, debugging, logging and analysis. The first *State* defined inside an *Automaton* is considered its initial state. *Jump* and *Flow* should be defined inside the State they will run. Jump rules always run before *Flow* rules. *Jump* and *Flow* rules run in the order the have been defined.

```
-- Creates a Hybrid Automaton
at = Automaton{

  id = "MyAutomaton",

  State{
    id = "first",
    Jump{ id = "j1", ... },
    ...,
    Jump{ id = "jM", ... },
    Flow{ id = "f1", ... },
    ...,
    Flow{ id = "fN", ... }
  },

  State{
    id = "second"
```

```
    Jump{ id = "j1", ... },
    ...,
    Jump{ id = "jQ", ... },
    Flow{ id = "f1", ... },
    ...,
    Flow{ id = "fR", ... }
  }

}

print( "Automaton: ");
print( at1.id.., ".. at1:getLatency() );
print( at1:getStateName());

print( "Running the Automaton... ");
ev = Event{ time = 0 }
at:execute(ev);
```

Listing 10.1: Creating Hybrid Automata in TerraME

The output generate by this model is shown commented at the bottom of Figure 10.1. In TerraME, any *Automaton* value has a special attribute, called "latency", which registers the passed period of time since the last time the automaton has change its internal discrete state. Also, any *Automaton* is supposed to be located at one *CellularSpace* where it is possible to know its current state in each *Cell*. However, in this example the automaton "at" has not been located in any cellular space. Before running, the latency of an automaton is zero. The string "Where?" is returned if one try to know the current automaton internal discrete state.

To run an *Automaton* one should call its "execute(Event)" method which receives an *Event* value as parameter. *Events* are rather discussed in the Chapter 11, by now, should be enough to know that an *Event* represents the time instant when something should happens in the model, for instance, to run an automaton, to save or synchronize a cellular space, and so on. The obligatory attribute *time* of an *Event* is a numeric value that registers this instant.

In Program 10.1, although the method execute has been called, the automaton "at" will not run, because it should be located in a cellular space before run. In other words, before run an *Automaton* should be inserted in an *Environment* and at least a *Trajectory* should be defined for it. For that, one should define all cellular spaces the automaton will be inside and fill them with dynamically created cells or with cells loaded from a geographic database. So, these cellular spaces should be inserted inside an *Environment* and then, the automaton should be also inserted in this *Environment*. An automaton runs in all cellular space previously insert in the *Environment* in which it is embedded. The automaton will not run

in cellular spaces inserted afterwards in the *Environment*. These steps are illustrated in Program 10.2. An *Environment* is a virtual world which local space properties are modeled by cellular spaces and actors and processes are represented by automata or agents. Environments are further discussed in Chapter 12.

The automaton "at" defined in Program 10.2 will traverse the cellular space "cs" according to the trajectory "it". The automaton method "setTrajectoryStatus(Boolean)" can be used turn on or off the trajectories defined for it. By default the trajectories of an automaton are turned off. If it is turned on, when executed the automaton automatically will traverse all trajectories defined inside it. Otherwise, the automaton will not run at all.

In this example, the automaton real variable "cont" will be used to count how many times the automaton jumps among its internal discrete states inside each cell belonging to the trajectory "it". The automaton will do 10 jumps in the first cell before to go the next. Then, it will do more 10 jumps in the second cell before to go to the next one, and so on.

```
cs = CellularSpace{ ... }
cs:load();

at = Automaton{

  id = "MyAutomaton",

  it = Trajectory{
    cs,
    function( cell ) return true; end
  },

  cont  = 0,

  State{
    id = "first",
    Jump{
      function( event, agent, cell )
        if (agent.cont < 10) then
          agent.cont = agent.cont + 1;
          print(agent.id..": "..agent.cont.." -
              cell["..cell.x..", "..cell.y.."]");
          return true
        end
        if( agent.cont == 10 ) then agent.cont = 0 end
        return false
      end,
      target = "second"
    }
  },
```

```
  State{
    id = "second",
    Jump{
      function( event, agent, cell )
        if (agent.cont < 10) then
          agent.cont = agent.cont + 1;
          print(agent.id..": "..agent.cont.." -
                cell["..cell.x..", "..cell.y.."]");
          return true
        end
        if( agent.cont == 10 ) then agent.cont = 0 end
        return false
      end,
      target = "first"
    }
  }
}

print( "Automaton... ");
env = Environment{ id = "MyEnvironment" }
env:add( cs );
env:add( at );
ev = Event{ time = 0 }
at:setTrajectoryStatus( true );
at:execute(ev);
```

Listing 10.2: A jumping Hybrid Automaton in TerraME.

A *Jump* rule has two obligatory properties: a conditional function and the identifier of its target state. The first parameter is a conditional unnamed function "function(Event, Agent, Cell) → Boolean" which receives the *Event* that has triggered the automaton execution as its first parameter, the reference to *Automaton* or *Agent* who owns the Jump rule been executed as the second parameter, and the cell where the *Jump* is been evaluated as the third parameter. This way, the conditional function can be defined based on the actual simulation time, on the automaton or agent states (discrete and continuous), and on local properties of the space where the rule is embedded. If the conditional function returns "true" the current automaton discrete state will transit form the state it is to the state identified by the jump property "target". Otherwise, the automaton discrete state will remain the same. If an invalid identifier is assigned to the property "target" of a *Jump*, the model may crash in runtime.

When an automaton is executed, the *Jump* rules of its states are always evaluated before the *Flow* rules. This fact guarantee that the automaton or agent will execute its *Flow* rules only when its internal discrete state reflects the state of the whole model. Therefore, automata and agent should be

carefully designed to avoid the model to hang on in runtime. For example, a *Jump* could send an automaton to a state where another *Jump*sends it back to its first state.

10.1 Exercises

10.1.1 Exercise 1

Implement and run the model shown in Program 10.2. Describe how it works.

10.1.2 Exercise 2

Inside each Jump rule there are two statements return. Do two experiments: (a) Change the code doing all the return statements to return "false" and run the model; (b) Change the code again and do all return statements to return "true". How the model works? What could be concluded from these experiments?

Chapter 11

A Hybrid Cellular Automata based rain drainage model in TerraME

In Chapter, we will present two rain drainage models which have been implemented based on the Cellular Automata Theory [15]-[16]: one discrete and one continuous.

In both models, the rain is simulated as a Hybrid Automaton, called "rain" which has two possible states, see Figure 11.1. In the state "ON" it is turned on and increments the water stored in the soil by a constant value RAIN = 2. In the state "OFF", the rain automaton is turned off do nothing. The automaton rains only on top of the mountains, where cells are higher than 254 meters.

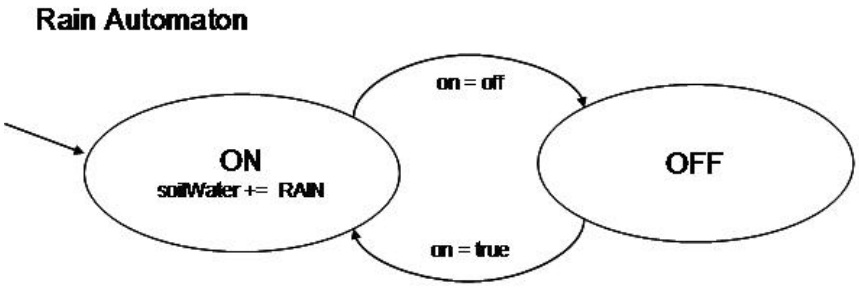


Figure 11.1: A Hybrid Automaton to simulate the rain.

The Hybrid Automaton "drainage" is used to simulate the rain drainage process and considers the vertical drainage of the rain water into the soil, known as infiltration process and the superficial flow of water from the higher cells to the lower ones, known as the runoff process. It has two possible internal discrete states, see Figure 11.2. In the state "DRY" it stores the runoff coming from the neighbor cells in the soil and drawn infiltrated amount of water from the soil. If the cell soil water is greater than the local soil infiltration capacity, the automaton transit from the "DRY" state to the "WET" state. In this latter state, the automaton calculates the local water surplus and sends it to the neighbor cells, causing the superficial water runoff.

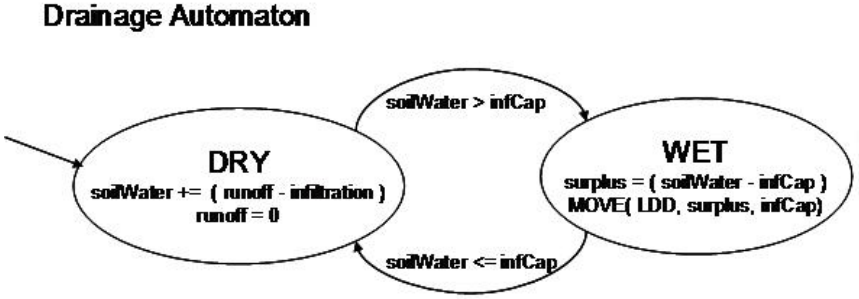


Figure 11.2: A Hybrid Automaton to simulate the drainage of rain water.

Programs 11.1 and 11.2 shows how the "rain" and "drainage" automata shown in Figures 11.1 e 11.1 can be written in TerraME. In this example, the both processes have been considered as discrete ones.

```
rain = Automaton{

  id = "Rain",

  it = Trajectory{
    csQ,
    function( cell ) return cell.altimetria > 254; end
  },

  on = true,

  State{
    id = "ON",
    Jump{
      function( event, agent, cell)
        return (not agent.on);
      end,
      target = "OFF"
    },
    Flow{
      function( event, agent, cell )
        cell.qtdeAgua = cell.past.qtdeAgua + RAIN;
      end,
      target = ""
    }
  },

  State{
      id = "OFF",
      Jump{
        function( event, agent, cell )
          return agent.on;
        end,
        target = "ON"
      }
  }
}
```

Listing 11.1: An automaton to simulate the rain in TerraME.

To consider the rain and the water drainage as continuous processes, the flow condition from the rain automaton "ON" state should be re-written as in Program 11.3. The flow condition from the drainage automaton "DRY" state should also be re-written as in Program 11.4.

```
drainage = Automaton{
  id = "Drainage",

  it = Trajectory{
    csQ,
    function( cell ) return true; end
  },
```

```
State{
  id = "DRY",

  Jump{
    function( event, agent, cell )
      return (cell.qtdeAgua > cell.capInf );
    end,
    target = "WET"
  },

  Flow{
    function( event, agent, cell)
      cell.qtdeAgua = cell.past.qtdeAgua + (cell.past.runoff
        - KInf*cell.past.qtdeAgua);
      cell.runoff = 0;
    end
  }
},

State{
  id = "WET",
  Jump{
    function( event, agent, cell )
      return ( cell.qtdeAgua <= cell.capInf );
    end,
    target = "DRY"
  },
  Flow{
    function( event, agent, cell)

      cell.surplus = cell.past.qtdeAgua - cell.capInf;
      if( cell.surplus < 0) then cell.surplus = 0; end

      -- PROCESS: send superficial flow to the lower
      -- neighbor cells
      if( cell.lowerNeigh > 0 ) then
        local runoff = cell.surplus/cell.lowerNeigh;
        ForEachNeighbor(
          cell,
          function(cell, neigh)
            if(cell.altimetria >
            neigh.altimetria) then
              neigh.runoff =
              neigh.runoff + runoff;
            end
            return true;
          end
        );
        cell.surplus = 0;
        cell.qtdeAgua = cell.capInf;
      end
```

```
      end
    }
  }
}
```

Listing 11.2: An automaton to simulate the drainage of rain water in TerraME.

```
State{
  id = "ON",
  ...
  Flow{
    function( event, agent, cell )
      cell.qtdeAgua = d{ function(t, q) return RAIN; end,
            cell.past.qtdeAgua, 0, 1}
    end
  }
}
```

Listing 11.3: A continuous flow rule to simulate the rain in TerraME.

```
State{
  id = "DRY",
  ...
  Flow{
    function( event, agent, cell)
      cell.qtdeAgua = d{ function( t, q) return cell.past.runoff
          - KInf*q;
           end, cell.past.qtdeAgua, 0,1, 0.0001 }
      cell.runoff = 0;
    end
  }
}
```

Listing 11.4: A continuous flow to simulate the drainage of rain water in TerraME.

The steps shown in Program 11.5 should be taken in order to execute the whole drain drainage model. First the cellular space "csQ" has been defined and loaded. So, both automata are defined. The environment "env" has been defined and the cellular space "cs" and the automata "rain" and "drainage" have been inserted in it. The automata trajectories have been turned on. Finally, the dynamic model has been executed. The "rain" automaton runs, the space is synchronized, then the "drainage" automaton runs and the space is synchronized again. The synchronization between the automata is used to mean that the processes are sequential in time. Without this synchronization step, the automata run over the same past value of the cellular space. The changes caused by the "rain" automaton in

the present value of the cellular space attributes would not be perceived by the "drainage" automaton, meaning theses processes are parallel in time.

```
csQ = CellularSpace{ ... }
csQ:load();

rain = Automaton{ ... }
drainage = Automaton{ ... }

env = Environment{ id = "MyEnvironment" }
env:add( csQ );
env:add( rain );
env:add( drainage );
rain:setTrajectoryStatus( true );
drainage:setTrajectoryStatus( true );

for t = 1, FINAL_TIME, 1 do
  ev = Event{ time = t }
  rain:execute(ev);
  csQ:synchronize();
  drainage:execute(ev);
  csQ:synchronize();
end
```

Listing 11.5: The Cellular Automata base rain drainage model overall structure.

11.1 Exercises

11.1.1 Exercise 1

Using the "cabecaDeBoi.mdb" geographical database and the codes from Program 11.1, 11.2 and 11.5 implement the whole model described in Chapter 11. Run the model from the instant 1 to the instant 20. Use the TerraView application to explore the model results. Explain the outcomes.

11.1.2 Exercise 2

Using the "cabecaDeBoi.mdb" geographical database and the codes from Program 11.1, 11.2 and 11.5 implement the whole model described in Chapter 11. Replace the Flow rules for the ones shown in Figures 11.3 and 11.4. Run the model from the instant 1 to the instant 20. Use the TerraView application to explore the model results. Explain the outcomes. Compare its results with the results observed in the exercise above.

11.1.3 Exercise 3

Using the "cabecaDeBoi.mdb" geographical database and the codes from Program 11.1, 11.2 and 11.5 implement the whole model described in Chapter 11. Run the model from the instant 1 to the instant 20 using different trajectories for the "rain" automaton. Use the TerraView application to explore the model results. Explain the outcomes.

Chapter 12

Timers, Events and Messages

Until now we have been working with discrete simulation. Only one temporal resolution ($\Delta t = 1$) has been considered. The time has not been explicitly represented in the model rules. Therefore for the model outcomes what matters is how many times the simulation loop has been execute. The same model output is produced if the simulation runs from time 0 to time 100 or from time 100 to time 200. However TerraME provides three temporal models for building dynamic models which take in consideration several process that act in different periodicity changing the space properties. These temporal models are: *Timer*, *Event* and *Message*.

A *Timer* is a scheduler which maintains a queue of pairs (*Event, Message*) to control the simulation clock. The pairs are ordered by the *Event* time attribute. An *Event* represents a time instant when the simulation engine must execute some computation for the modeler, called Message. A *Message* is a modeler defined function from where, in general, simulation engine services are invoked. Among these services, there are services to load data from the database, to save data in the database, to execute a specific automaton, to synchronize a cellular space, etc.

In *TerraME*, a *Timer* is a container for pairs (*Event, Message*), Program 12.1. Any finite number of pairs (*Event, Message*) can be added to a *Timer*. Pairs (*Event, Message*) are inserted in *Timer* in the same order they have been defined. However, inside the *Timer* they will be re-ordered in a chronological sequence.

```
time = Timer{
  Pair{
```

```
    Event{ ... },
    Message{ ... }
  },
  Pair{
    Event{ ... },
    Message{ ... }
  },
  ...
  Pair{
    Event{ ... },
    Message{ ... }
  }
}
```

Listing 12.1: The general structure of a Timer in TerraME.

An Event is defined by two mandatory properties (*time* and *period*) and an optional one (*priority*). The *time* property defines the next instant of time (in the simulation clock) when the event must occur. The *period* property defines the periodicity in which the event must occur. The priority property is used to decide what event must occur first when two events have the same value for the *time* property. The default *priority* value is 0 (zero). Smaller values have higher priority. Program 12.2 presents an Event that must occur at the year 1985, repeat every year, and has priority equal to -1.

```
Event{ time = 1985, period = 1, priority = -1 }
```

Listing 12.2: Defining an Event in TerraME Modeling Language.

A Message is a user defined function whose parameter is the *Event* that has caused its execution. Program 12.3 shows a Message that prints the simulation time in the screen, executes the automaton "rain", and prints the word "Rained" in the screen.

```
Message{
  function( event )
    print(event:getTime());
    rain:execute( event );
    print("Rained");
    return false;
  end
}
```

Listing 12.3: Defining a Message in TerraME.

When a *Timer* is executed, it keeps running until there are no pairs (*Event, Message*) inside it. At each simulation step, a *Timer* pops the pair on the top of its internal queue, updates its internal clock to the instant

time registered in pair Event and, then, executes the pair Message. The "*Message(Event)* → Boolean" function always returns a Boolean value. If the returned value is true, the *Event.time* property will be calculated as *Event.time = Event.time + Event.period* and the pair (*Event, Message*) will be re-inserted in *Timer* queue. This way, the *Event* will happen again in the simulation future and the Message will be execute once more. If the returned pair is false, the pair (*Event, Message*) will discarded. The default return value is true. Program 12.4 shows an example of *Timer* which runs from the time 0 to the time 10 and prints the sequence 0, 0.5, 1, 1.5, 2, ..., 8.5, 9, 9.5, 10 on the screen. If one changes the *Message* function to return false, the *Timer* will run just once and print only the number 0 on the screen.

```
clock = Timer{
  Pair{
    Event{ time = 0, period = 0.5 },
    Message{ function(event) print(event:getTime()) end }
  }
}
clock:execute(10);
```

Listing 12.4: An example of a Timer with one internal pair (Event, Message).

When executed, the code shown in Program 12.5 prints the numeric sequence 0, 1, 2, 3, 3, 3.5, 4, 4, 4.5, 5, 5, 5.5, 6, 6, 6.5, 7, 7, 7.5, 8, 8, 8.5, 9, 9, 9.5, 10 on the screen. In the instants 0, 1 and 2 just the message from the first pair runs. From the instant 3 to 9 the first message runs once and the second message runs twice. In the instant 10, just the first message runs and the simulation finishes.

```
clock = Timer{
  Pair{
    Event{ time = 0, period = 1 },
    Message{ function(event) print(event:getTime()) end }
  },
  Pair{

    Event{ time = 3, period = 0.5 },
    Message{ function(event) print(event:getTime()) end }
  }
}
clock:execute(10);
```

Listing 12.5: An example of a Timer with two pairs (Event, Message).

12.1 Exercises

12.1.1 Exercise 1

Implement the model shown in Program 12.4. Run the model for several event periodicities. How the model works?

12.1.2 Exercise 2

Implement the model shown in Program 12.5. Run the model. Assign the value 1 to the "period" attribute of the second event. Run the model again. How the model works? What can be concluded?

12.1.3 Exercise 3

Implement the model shown in Program 7.3. Change the Message code of the second pair so that it returns "false". Run the model. How the model works?

Bibliography

[1] Carneiro, T., "Nested-CA: a foundation for multiscale modeling of land use and land change", in PhD Thesis in Computer Science. 2006, National Institute of Space Research: SÃ£o JosÃ© dos Campos, Brazil. p. 109.

[2] Odum, H.T., "Systems Ecology: An Introduction". 1983, March: John Wiley and Sons.

[3] Hestenes, D., "Toward a Modeling Theory of Physics Instruction". American Journal of Physics, 1987. 55(5): p. 440-454.

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

[5] Gibson, C.C., E. Ostrom, and T.K. Ahn, "The concept of scale and the human dimensions of global change: a survey". Ecological Economics, 2000. 32(2): p. 217-239.

[6] Kok, K. and A. Veldkamp, "Evaluating impact of spatial scales on land use pattern analysis in Central America". Agriculture Ecosystems e Environment, 2001. 85(1-3): p. 205-221.

[7] Parker, D.C., T. Berger, and S.M. Manson, "Agent-Based Models of Land-Use and Land-Cover Change, in Report and Review of an International Workshop", L.R. No.6, Editor. 2001: Irvine, California, USA.

[8] Verburg, P., P. Schot, and e. al., "Land use change modeling: current practices and research priorities". GeoJournal, 2004. 61(4): p. 309-324.

[9] Wesseling, C.G., D. Karssenberg, and e. al., "Integrating environmental models in GIS: the development of a Dynamic Modelling language". Transactions in GIS, 1996. 1: p. 40-48.

[10] Wainwright, J., Mulligan, M.. "Environmental Modelling: Finding Simplicity in Complexity", John Wiley and Sons. 2004.

[11] Briassoulis, H.. "Analysis of Land Use Change: Theoretical and Modeling Approaches, Lesvos, Greece". Doctors Thesis in Geography. Aegean University. Regional Research Institute, West Virginia University, 2000.

[12] Hopcroft, J. E., Ullman, J. D.. "Introduction to automanta theory, language and computation". Addison-Wesley Publishinh Company. 1979.

[13] Minsky, L. M.. "Computation: finite and infinite machines". Prentice-Hall, Inc. 1967

[14] Henzinger, T. A.. "The Theory of Hybrid Automata". In: Symposium on Logic in Computer Science (LICS'96), 11., 1996, Washington. Proceedings Washington: IEEE Computer Society, 1996.

[15] von Neumann, J.. "Theory of self-reproducing automata". Illinois: A.W. Burks, 1966.

[16] Couclelis, H.. "Cellular Worlds - a Framework for Modeling Micro-Macro Dynamics". 1985, Environment and Planning A. 17(5): p. 585-596.

[17] Batty, M.. "Modeling urban dynamics through GIS-based cellular automata". 1999, Computers, Environment and Urban Systems. 23: p.205-233.

[18] Wolfram, S.. "Cellular automata as models of complexity". 1984, Nature, 311: p. 419-424.

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

[20] Russel, S. J, Norvig P.. "Artificial Intelligence - A Modern Approach". Prentice Hall. Nova Jersey, 1995.

[21] Rosenschein, S. J., Kaelbling, L. P. "A situated view of representation and control". Artificial Intelligence 73:149-173, 1995

[22] Camara, G., et al. "TerraLib: Technology in Support of GIS Innovation. in II Brazilian Symposium in Geoinformatics", GeoInfo2000. 2000. São Paulo.

[23] Ierusalimschy, R., L.H. Figueiredo, and W. Celes, "Lua-an extensible extension language. Software: Practice e Experience". 26, 1996: p. 635-652.

[24] Figueiredo, L.H., R. Ierusalimschy, and W. Celes, "Lua: an extensible embedded language". Dr. Dobb's Journal 1996. 21(12): p. 26-33.

[25] Ierusalimschy, R., L.H. Figueiredo, and W. Celes, "Lua 5.0 Reference Manual". 2003, Computer Science Department - PUC-Rio: Rio de Janeiro. p. Report PUC-RioInf.MCC14/03 Abril 2003.

[26] Costanza, R., Maxwell, T.. "Resolution and Predictability: an Approach to the Scaling Problem". 1994, Landscape Ecology, 9(1): p 47-57.

[27] Kok, K., Veldkamp, T. A.. "Evaluating impact of spatial scales on land use pattern analysis in Central America". 2001, Agriculture Ecosystems & Environment. 85(1-3): p. 205-221.

[28] Carneiro, T., Camara, G., Maretto, R.. "Irregular Cellular Spaces: Supporting Realistic Spatial Dynamic Modeling using Geographical Databases". 2008, X Brazilian Symposium on Geoinformatics, GeoInfo 2008. SBC, Rio de Janeiro.

[29] Aguiar, A., G. Cmara, and R. Cartaxo. "Modeling Spatial Relations by Generalized Proximity Matrices". in V Brazilian Symposium in Geoinformatics - GeoInfo 2003. 2003. Campos do Jordo, SP, Brazil.

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

[31] Pedrosa, B. M. ; Cmara, G. ; Fonseca, F. ; Souza, R. C. M.. "TerraML: a Language to Support Spatial Dynamic Modeling". In: IV Simpsio Brasileiro de Geoinformtica - GeoInfo 2002, 2002, Caxambu - MG.

Index

Notas em Matemática Aplicada

Arquivos em pdf disponíveis em http://www.sbmac.org.br/notas.php

1. Restauração de Imagens com Aplicações em Biologia e Engenharia
   Geraldo Cidade, Antônio Silva Neto e Nilson Costa Roberty
2. Fundamentos, Potencialidades e Aplicações de Algoritmos Evolutivos
   Leandro dos Santos Coelho
3. Modelos Matemáticos e Métodos Numéricos em Águas Subterrâneas
   Edson Wendlander
4. Métodos Numéricos para Equações Diferenciais Parciais
   Maria Cristina de Castro Cunha e Maria Amélia Novais Schleicher
5. Modelagem em Biomatemática
   Joyce da Silva Bevilacqua, Marat Rafikov e Cláudia de Lello Courtouke Guedes
6. Métodos de Otimização Randômica: algoritmos genéticos e "simulated annealing"
   Sezimária F. Pereira Saramago
7. "Matemática Aplicada à Fisiologia e Epidemiologia"
   H.M. Yang, R. Sampaio e A. Sri Ranga
8. Uma Introdução à Computação Quântica
   Renato Portugal, Carlile Campos Lavor, Luiz Mariano Carvalho e Nelson Maculan
9. Aplicações de Análise Fatorial de Correspondências para Análise de Dados
   Homero Chaib Filho

10. Modelos Matemáticos baseados em autômatos celulares para Geoprocessamento

    Marilton Sanchotene de Aguiar, Fábia Amorim da Costa, Graçaliz Pereira Dimuro e Antônio Carlos da Rocha Costa

11. Computabilidade: os limites da Computação

    Regivan H. N. Santiago e Benjamín R. C. Bedregal

12. Modelagem Multiescala em Materiais e Estruturas

    Fernando Rochinha e Alexandre Madureira

13. Modelagem em Biomatemática (Coraci Malta ed.)

    1 - "Modelagem matemática do comportamento elétrico de neurônios e algumas aplicações"

    Reynaldo D. Pinto

    2 - "Redes complexas e aplicações nas Ciências"

    José Carlos M. Mombach

    3 - "Possíveis níveis de complexidade na modelagem de sistemas biológicos"

    Henrique L. Lenzi, Waldemiro de Souza Romanha e Marcelo Pelajo- Machado

14. A lógica na construção dos argumentos

    Angela Cruz e José Eduardo de Almeida Moura

15. Modelagem Matemática e Simulação Numérica em Dinâmica dos Fluidos

    Valdemir G. Ferreira, Hélio A. Navarro, Magda K. Kaibara

16. Introdução ao Tratamento da Informação nos Ensinos Fundamental e Médio

    Marcilia Andrade Campos, Paulo Figueiredo Lima

17. Teoria dos Conjuntos Fuzzy com Aplicações

    Rosana Sueli da Motta Jafelice, Laércio Carvalho de Barros, Rodney Carlos Bassanezi

18. Introdução à Construção de Modelos de Otimização Linear e Inteira

    Socorro Rangel

19. Observar e Pensar, antes de Modelar
    Flavio Shigeo Yamamoto, Sérgio Alves, Edson P. Marques Filho, Amauri P. de Oliveira
20. Frações Contínuas: Propriedades e Aplicações
    Eliana Xavier Linhares de Andrade, Cleonice Fátima Bracciali
21. Uma Introdução à Teoria de Códigos
    Carlile Campos Lavor, Marcelo Muniz Silva Alves, Rogério Monteiro de Siqueira, Sueli Irene Rodrigues Costa
22. Análise e Processamento de Sinais
    Rubens Sampaio, Edson Cataldo, Alexandre de Souza Brandão
23. Introdução aos Métodos Discretos de Análise Numérica de EDO e EDP
    David Soares Pinto Júnior
24. Representações Computacionais de Grafos
    Lílian Markenzon, Oswaldo Vernet
25. Ondas Oceânicas de Superfície
    Leandro Farina
26. Técnicas de Modelagem de Processos Epidêmicos e Evolucionários
    Domingos Alves, Henrique Fabrício Gagliardi
27. Introdução à teoria espectral de grafos com aplicações
    Nair Maria Maia de Abreu, Renata Raposo Del-Vecchio, Cybele Tavares Maia Vinagre e Dragan Stevanović
28. Modelagem e convexidade
    Eduardo Cursi e Rubens Sampaio
29. Modelagem matemática em finanças quantitativas em tempo discreto
    Max Oliveira de Souza e Jorge Zubelli
30. Programação não linear em dois níveis: aplicação em Engenharia Mecânica
    Ana Friedlander e Eduardo Fancello

31. Funções simétricas e aplicações em Combinatória
    José Plinio de Oliveira Santos e Robson da Silva
32. Semigrupos aplicados a sistemas dissipativos em EDP
    Carlos Raposo da Cunha
33. Introdução à Simulação Estocástica para Atuária e Finanças Usando R
    Hélio Côrtes Vieira, Alejandro C. Frery e Luciano Vereda
34. Modelos de Sustentabilidade nas Paisagens Amazônicas Alagáveis
    Maurício Vieira Kritz, Jaqueline Maria da Silva e Cláudia Mazza
35. Uma Introdução à Dinâmica Estocástica de Populações
    Leonardo Paulo Maia
36. Geometria de Algoritmos Numéricos
    Gregorio Malajovich
37. Equações Diferenciais, Teorema do Resíduo e as Transformadas Integrais
    Edmundo Capelas de Oliveira e Jayme Vaz Júnior