Chapter 15
Parallel Cellular Programming for Emergent Computation

Domenico Talia and Lev Naumov

15.1 Introduction

In complex systems, global and collective properties cannot be deduced from its simpler components. In fact, global or collective behavior in a complex system emerges from evolution and interaction of many elements. Therefore programming emergent systems needs models, paradigms, and operations that allow for expressing the behavior and interaction of a very large number of single elements.

Because of their inherent parallelism, cellular automata (CA) can be exploited to model large scale emergent systems on parallel computers. In this scenario parallel cellular models and languages provide useful tools for programming emergent computations that model complex phenomena in many application domains from science and engineering to economics and social sciences.

The programming of emergent phenomena and systems based on traditional programming tools and languages is hard and it results in long and complex code. This occurs because these programming approaches are based on the design of a system as a whole. Design and programming do not start from basic elements or system components, but represent a system by modeling its general features. On the contrary, it is better to design emergent and complex systems by means of paradigms that allow for expressing the behavior of the single basic elements and their interactions. The global behavior of these systems then emerges from the evolution and interaction of a massive number of simple elements; hence it does not need to be explicitly coded.

Parallel architectures such as multicore, clusters, and multicomputers are well suited for implementing inherently parallel computing abstract models such as cellular automata, neural networks, and genetic algorithms that represent new mathematical models for describing complex scientific phenomena and systems with emergent properties. All cells of a cellular automaton are updated in parallel. Thus the state of the entire automaton advances in discrete time-steps and the global behavior of

D. Talia (✉)
DEIS, University of Calabria, Rende, Italy
e-mail: talia@deis.unical.it

the system is determined by the evolution of the states of each cell as a result of multiple local interactions. Cellular automata provide a global framework for the implementation of parallel applications that represent natural solvers of dynamic complex phenomena and systems based on the use of discrete time and discrete space.

CA are intrinsically parallel and they can be efficiently mapped onto parallel machines because the communication flow between processors can be kept low, communication patterns are regular and involve only neighbor cells. Inherent parallelism and restricted communication are two key points for the efficient use of CA for high performance simulation [36].

The cellular automata theory was invented half a century ago. The exact author of this area cannot be named definitely. In 1948 John von Neumann [1] gave a lecture entitled "The General and Logical Theory of Automata", where he presented his ideas of universal and self-reproducing machines. According to his own statement, his work was inspired by Stanislaw Ulam [2]. Konrad Zuse [3] also suggested that the universe could be a cellular automaton. Zuse used this idea for developing computing machines. At the same time, some members of the scientific society regard the paper by Wiener and Rosenblueth [4], or the mathematical work that was done in early 1930s in Russia as the start of the field [5].

However more recently CA emerged as a significant tool for modeling and simulation of complex systems. This occurred thanks to the implementation of cellular automata on high-performance parallel computers. Parallel cellular automata models are successfully used in fluid dynamics, molecular dynamics, biology, genetics, chemistry, road traffic flow, cryptography, image processing, environment modeling, and finance. To explain this approach, we discuss the main features of cellular automata parallel software environments and how those features can support the solution of large-scale problems.

To describe in detail how the marriage of the cellular automata theory with parallel computing is very fruitful, we will discuss some leading examples of high-performance cellular programming tools and environments such as CAMELot and CAME$_{\&}$L. Moreover we will discuss programming of complex systems in CA languages such as CARPET. Those parallel cellular automata environments have been used to solve complex problems in several areas of science, engineering, computer science, and economy. They offer a well structured way to facilitate the development of cellular automata applications, providing transparent parallelism and reducing duplication of effort by implementing a programming environment once and making it available to developers. We discuss the basic principles of parallel CA languages and describe some practical programming examples in different areas designed by means of those parallel CA systems.

15.2 Cellular Automata Systems

In the past decade, several cellular automata environments have been implemented on current desktop computers. For large size two or three dimensional cellular automata the computational load can be enormous. There are two main alternatives that allow to achieve high performance in the implementation of CA. The first one is the design of special hardware devoted to the execution of CA. The second alternative is based on the use of commercially-available parallel computers for developing parallel CA software tools and environments.

CA software and hardware systems belong to the class of problem-solving environments (PSE). The community has formulated the following common recommendations for a general PSE:

1. It should **reduce the difficulty** of the simulation [6].
2. It should **reduce costs and time** of complex solutions development [6].
3. It should allow to perform experiments **reliably** [6].
4. It should have a **long lifetime** without getting obsolete [6].
5. It should **support the plug-and-play paradigm** [6].
6. It should **exploit** the paradigm of the **multilevel abstractions** and complex properties of science [6].
7. User should be able to **use the environment without any specialized knowledge** of the underlying computer hardware or software [7].
8. It should be pointed at the **wide scope** of problems [7].
9. It should be able to coordinate **mighty computational power** to solve a problem [7].
10. It should be **complete**, containing and providing all computational facilities for solving a problem in a target domain [8].
11. **Extensibility** of the environment will provide the ability to enlarge the target problem domain, to enrich the set of supported tools and provided features. This can be achieved with the help of a component-based design. A component approach also complies with the trend that modern distributed problem-solving facilities should be based on web and grid services [6] or Common Object Request Broker Architecture (CORBA) objects [9].

Basing on common considerations, the software or hardware facility, which allows to perform experiments using CA should have the following attributes:

1. It should **hide the complexity** of used computational architecture, operating system or networking mechanism. The language which a researcher should use to control the environment has to be related to basic cellular automata concepts and to the target problem domain.
2. It should allow to **setup and tune** a cellular automaton for the computational experiment. The degree of freedom which is granted to a user here may play the key role.
3. It should give an opportunity to **run and control** a computational experiment. A good solver should use all the benefits provided by the computational architecture and utilize as much parallelizable aspects of the experiment's iteration as it is possible to improve the throughput.
4. It should support **visualization**, because this feature plays one of the key roles in understanding the phenomenon, especially, when modeling spatial-distributed systems.

5. It should provide a set of tools to **analyze** the computational experiment's intrinsic characteristics and their tendencies, current state of the automaton's grid or any other data, which is possible to obtain.
6. It should provide the **reproducibility** and allow to share the description of the way one have done the certain computational experiment. Donald Knuth have declared this feature as a required one for a scientific method [10].

The following list unites two previous ones and consists of concrete required features for a CA-based PSE. At the end of each statement there are references to attributes stated above, given in italics. References to the first list are preceded by "*PSE:*", whereas references to the second one are preceded by "*CA:*". Moreover, the "Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science" [7, 11] have produced "Findings" and "Recommendations" for PSEs. They will not be listed here, but will be referenced in the "*PSE:*" section as "F_n" or "R_n" respectively, where n is the number of distinct finding or recommendation stated in [7].

1. The environment should be as **universal and customizable** as possible. The support of miscellaneous grids, types of neighborhoods and boundary conditions is desirable or even necessary. Environment should allow to choose the type of the automaton to be modeled and the parameters of the experiment from the widest possible spectrum of variants *(PSE: 2, 8, 10; CA: 2)*.
2. **Extensibility** is a contemporary and actively used property of software and hardware systems. The ability to incorporate novel functionalities and algorithms may be one of the most advantageous for the PSE *(PSE: 2, 4, 5, 6, 8, 9, 10, 11,* F_6, R_1*; CA: 1)*.
3. The environment should support **modern parallel or distributed computational technologies**. For software CA system this means that it should involve cluster or Grid computing, Message Passing Interface (MPI), Parallel Virtual Machine (PVM), OpenMP or other technologies. Hence the software has to emulate homogeneous parallel architecture of cellular automaton, but not counterfeit it. Nevertheless without the use of high-performance parallel hardware the CA model would be of no practical use for solving real world problems *(PSE: 1, 2, 3, 9, 10,* F_1, F_2, F_3*; CA: 1, 3)*.
4. The environment should provide a visually attractive, handy and clear **user interface**. It has to preserve the interactivity even when performing long experiments, preserving the reliable control *(PSE: 1, 2, 7,* F_2, F_4, F_7, R_7*; CA: 1, 2, 3)*.
5. The experiment's **description language** has to be close to the language of the target problem domain and as far from the implementation as possible. Description should be independent of the computational architecture, level of resources, and operating system. The ability to involve such dependencies is definitely considered as a powerful option *(PSE: 1, 2, 6, 7, 10,* F_1, F_4, R_7*; CA: 1, 2)*.
6. Grid state **visualization** is a most straightforward way of experiment's representation. Nowadays the scientific visualization seems to be a separate industry [12]. So there is no need for the CA-based PSE to be concurrent with top-level tools in

this area. Nevertheless the environment should contain the basic set of features and preferably be compatible with the specialized visualization software on the level of data-files *(PSE: 2, 10, R_7; CA: 4).*

7. The environment should support **analysis** functionality to monitor the quality of the experiment, study the progress and the final results for making conclusions and producing new scientific knowledge *(PSE: 2, 6, 10, R_7; CA: 5).*
8. The environment should allow to **reproduce** experiment made once on the same or another computational system. This will give an opportunity to share the knowledge and the experience between researchers, eliminate ambiguous computations, postpone the simulation, reanalyze, and revisualize or generally reuse the data. This can be achieved by providing the ability to store/restore full computational experiment setup and automaton's grid state to/from a file *(PSE: 1, 2, 4, 10, F_1, F_4, R_6; CA: 6).*

Such tight connection of properties listed above with general recommendations for the PSEs design and features list for CA modeling facilities allows to conclude that these eight properties are close to be common requirements for cellular automata based modeling environments.

The number of the software created for cellular automata modeling is impressive [13, 14]. The apogee of this boom was at the 1990s. Many projects have been already outdated, but some new successful prototypes appeared.

The comparative survey of the existing software and hardware facilities is summarized in Table 15.1. The first column contains the name of the project with references. The second one presents the target platform and the third – the year of the latest known release or of the last publication devoted to the instrument. Further eight columns contain pluses if project satisfies the requirement with corresponding number (see the previous list) and minuses otherwise.

It is impossible and useless to overview all the existing projects, which were created ever. Those of them which last known version had been released in the

Table 15.1 The comparative survey of existing cellular automata modeling environments. The first raw contains project's name with references, the second is used for the information about the target platform, the third stores the year of current release. The rest raws contain pluses or minuses depending on the conformance to the corresponding requirements (see the previous list)

Name	Platform	Release	1	2	3	4	5	6	7	8
CAGE [15]	Windows	2004	+	−	−	+	+	+	−	+
CAM [16, 17]	iX86 or Sun	1994	+	+	+	+	+	+	−	+
CAMEL [18–20]	UNIX/Linux	1996	+	−	+	+	+	+	−	+
CAME$_{\&}$L [14, 21, 23]	Windows	2010	+	+	+	+	+	+	+	+
CAMELot [24]	UNIX/Linux	2001	+	−	+	+	+	+	+	+
Cellular [25]	UNIX/Linux, CygWin	1999	+	+	−	+	+	+	−	+
JCASim [26]	Java	2001	+	+	+	+	+	+	+	+
MCell [27]	Windows	2001	+	+	−	+	+	+	−	+
ParCeL-6 [28, 29]	UNIX/Linux, Windows	2004	+	+	+	−	+	−	+	+
SIMP/STEP [30, 31]	UNIX/Linux, Windows	2003	+	+	+	+	+	+	+	+
Trend [32]	UNIX/Linux, Java	2002	+	−	−	+	+	+	−	+

twentieth century were mostly excluded from the study. Only several of them are listed, because of their significant historical value. Also some relatively new projects are deliberately not presented if they are deemed to be unsuitable for the research and scientific modeling.

Projects are listed in the alphabetical order. In the following there are short, one-paragraph reviews, which briefly describe each project in more detail. The name of the project which is subjected to the review is shown with bold when it appears for the first time.

The name **CAGE** stands for “Cellular Automata General Environment”. The tool does not support any parallel or distributed computational technology, but this is compensated by the universality. Authors have generalized the notion of the “cellular automaton” and used their vision of it for the software design. The environment supports multilayered grids, rich means for the neighborhoods’ formation (including the query-based one) and an ability to use irregular grids. The spectrum of the functionality is extremely rich. Transition rules are to be defined using the C-like language with the help of built-in visual programming means. Written rules are being translated to C++ sources and compiled into the executable code for better computational throughput. Despite of the functional richness, all grid’s layers seem to be 2D only.

CAM means “Cellular Automata Machine” and represents a single instruction multiple data (SIMD) hardware implementation of the modeling environment. CAM-6 [16] is a PCI-device which should be plugged into iX86 workstation governed by PC-DOS operating system. It supports 256×256 2D grids with Moore, von Neumann and Margolus neighborhoods. CAM-8 [17] is a device which works in tandem with Sun workstations via CBus and should be controlled by the accompanying STEP software (a predecessor of SIMP/STEP project which is also present in this survey). Eighth version of the machine supports 3D grids. There is an ability to extend the grid by using multiple device specimens. Visualization is performed by the XCAM utility. Transition rules are to be programmed using a dialect of Forth language supplemented with necessary routines. For each transitions function the machine compiles the full lookup table. However, multiple instruction multiple data (MIMD) architectures are more flexible than SIMD machines for implementing CA, as they allow to deal with irregularities on a microscopic level of the algorithm asynchronously and to efficiently simulate also heterogeneous systems. On a higher level of abstraction it is possible to synchronize the parallel components of a MIMD system explicitly as this is the only way to maintain global invariance of CA.

The project’s name **CAMEL** stands for “Cellular Automata environMent for systEms modeLing”. This software was designed to perform computations on the net of transputers or using MPI. It supports grids of up to three dimensions and complex neighborhoods. The cell’s state can be represented by the instance of a data structure composed of basic types. For the CA definition it uses a specialized language CARPET (“CellulAR Programming EnvironmenT”) [19] which will be discussed in Sects. 15.3 and 15.4. The program written using this language traditionally consists of the declarative part and the statements. The language is clear and successfully hides implementation issues coming from a parallel computer’s architecture complexity, allowing to describe automata and rules in general terms. The additional program IVT (comes from "Interactive Visualization Tool") has been added to CAMEL software to improve the data visualization. In twenty-first century the same group has switched to the development of CAMELot project [24].

By coincidence the name of the project **CAME$_{\&}$L** is very similar to the previous one. Nevertheless in this case it stands for "Cellular Automata Modeling Environment & Library". The ampersand in the abbreviation appeared exactly for it to be distinguishable from the CAMEL. This project will be discussed in more detail in Sect. 15.5, but will be reviewed briefly. The key idea was to create a universal and extensible facility, which supports parallel and distributed computing without any target problem domain specialization. This was achieved by usage of the CA based computational experiment decomposition (see Sect. 15.5.1). The software allows synchronous, asynchronous, probabilistic, inhomogeneous, and any other kind of CA with arbitrary grids, neighborhoods or type of cells' state. Even if the particular functionality or distinct automata type implementation is not included into the standard software package, one may add it and make new solution immediately available for the community. So CAME$_{\&}$L users can be divided into two interconnected and mixed groups: researchers who are just building solutions from the bricks they have, and developers who enhance the set of bricks for themselves and everyone. Ideal situation will be reached when anyone will get the ability to perform arbitrary cellular automata based experiments without the need to create new bricks.

As a descendant of CAMEL, **CAMELot** ("CAMEL Open Technology") also uses CARPET language [19] for the experiment description and MPI for the simulation execution. This project will be discussed in Sect. 15.4 and here it will be overviewed briefly. The software represents the environment for programming and seamlessly parallel execution of cellular automata. It has a graphical user interface for experiment setup, control and visualization. It also includes the customizable tool to produce traces of the simulation in a specified format thus allowing to post-process the output of the experiment by means of the external utilities. Moreover it supports profiling capabilities. The simulator is flexible with regard to cellular space size and dimension (form 1D to 3D), cell's state structure, neighborhood and rules. The program, written using CARPET is translated and compiled into UNIX/Linux executable file. The experiment setup preparation consists of editing of a text file.

Cellular software consists of the programming language (Cellang 2.0), associated compiler (`cellc`), virtual machine for the execution (`pe-scam`) and the viewer (`cellview`). A program written with Cellang 2.0 consists of two parts: the description and the set of statements. The description determines dimensionality of a grid, data-fields, which are contained in each cell, and ranges of acceptable values for each field. There are two possible statements: an assignment and a conditional test. The only possible data type is integer. The viewer is independent of the Cellang 2.0 language and the compiler. The input format for the viewer is identical to the output format of Cellang 2.0 programs. The software supports different grids of arbitrary dimensionality, non-trivial neighborhoods, several kinds of boundary conditions.

JCASim represents a general-purpose system for simulating CA on Java platform. It includes the standalone application and the applet for web presentations.

The cellular automaton can be specified in Java, CDL [33] or using the interactive dialogue. It supports 1D, 2D (square, hexagonal, or triangular) and 3D grids, different neighborhoods, boundary conditions (periodic, reflective or constant), and can display cells using colors, text, or icons. Initially CDL was designed to describe the hardware, which simulates homogeneous structures, but it can also be applied in software as a powerful and expressive tool. JCASim allows any constructions acceptable in CDL. For example, like in CDL, cell's state can be represented with theoretically unlimited amount of integer and floating-point variables. With the package `CAComb` the software allows to simulate CA on several machines in parallel. `CAAnalysis` package incorporates automatic analysis (the mean-fields and similar approximations will be calculated automatically).

MCell or "Mirek's Cellebration" is a very small and simple Windows application which supports 2D grids and no parallel or distributed computing. But despite of this its effort is great, because it can easily show the simplicity, beauty and power of a cellular automata to people who are far from this field of science. This is possible due to successful graphical user interface which is clear for non-specialists and a wide library of examples. Transition rules can be defined using the interface means or by creation of external dynamic-link library.

Project **ParCeL-6** represents the multi-layer cellular computing library for multiprocessor computers, clusters and Grids. The goal of its creation was to decrease the development time for the fine-grained applications. It is implemented in C language and can be linked to C and C++ programs. There are two subversions of the software: ParCeL-6.1 for architectures supporting the memory sharing paradigm and ParCeL-6.2 for architectures supporting the message passing approach. The cluster version of ParCeL-6 was developed in the framework of the Grid-eXplorer project. High level generic and parallel neural model of computations allows smart programming for numerous computing units. ParCeL offers the extended cellular programming model and maps "small" computing units on the "big" processors of parallel machines. When a cell is created, host processor is pointed out and unique registration number is associated with the cell. This number allows to identify it in a cellular network. Finally the cell is created directly on its host processor and executes the computing cycle on it. The software is also able to perform the automatic parallelization of the source code for the multiprocessor machines.

SIMP/STEP is a general-purpose software platform, which includes the language for cellular automata, lattice gases, and a "programmable matter" definitions. It is based on the Python programming language and suites for the wide range of problems. The software consists of two parts: SIMP is the user environment built on STEP, the applications programming interface, which separates conceptual components from implementation details, optimization routines etc. The software supports parallel computing technologies, has visualization and analysis capabilities. SIMP supports 2D rendering, but there are some experimental hooks for the 3D rendering, using VTK [12].

Trend is the 2D cellular automata programming environment with the integrated simulator and the compiler, which produces the virtual machine code for the evaluation module (under UNIX/Linux only) or Java machine. It has several interesting features: the simulation backtracking, conflicts catching, flexible template design and others. The Trend language allows user-defined terms, symmetrically rotatable statements and other constructions specific for the cellular automata programming. The software supports arbitrary neighborhoods within 11×11 region around the center cell. Each cell can be in the state which is coded by unsigned integer variable. The project does not support any parallel computing technologies.

15.3 Parallel CA Languages

For developing cellular automata on parallel computers two main approaches can be used. One is to write programs that encode the CA rules in a general-purpose parallel programming language such as HPF, Erlang, Java, Linda or CILK or still using a high-level sequential language like C++, Fortran or Phyton with one of the low-level toolkits/libraries currently used to implement parallel applications such as MPI, PVM, or OpenMP. This approach does not require a parallel programmer to learn a new language syntax and programming techniques for cellular programming. However, it is not simple to be used by programmers that are not experts in parallel programming and code consists of a large number of instructions even if simple cellular models must be implemented.

The other possibility is to use a high-level language specifically designed for CA, in which it is possible to directly express the features and the rules of CA, and then use a compiler to translate the CA code into a program executable on parallel computers. This second approach has the advantage that it offers a programming paradigm that is very close to the CA abstract model and that the same CA description could possibly also be compiled into different code for various parallel machines. Furthermore, in this approach parallelism is transparent from the user, so programmers can concentrate on the specification of the model without worrying about architecture related issues. In summary, it leads to the writing of software that does express in a natural manner the cellular paradigm, and thus programs are simpler to read, change, and maintain. On the other hand, the regularity of computation and locality of communication allow CA programs to achieve good performance and scalability on parallel architectures.

In recent years, several cellular automata environments have been implemented on current desktop computers as well (see Sect. 15.2). Sequential CA-based systems can be used for educational purposes and very simple simulations, but real world phenomena simulations generally take very long time, or in some cases cannot be executed, on this class of systems because of memory or computing power limits. Therefore, massively parallel computers are the appropriate computing platform for the execution of CA models when real life problems must be solved. In fact, for two and three dimensional cellular automata of large size the computational load can be enormous. Thus, if CA are to be used for investigating large complex phenomena, their implementation on high performance computers composed of several processors is a must.

In particular, general-purpose distributed-memory parallel computers offer a very useful architecture for a scalable CA machine both in terms speed-up, programmability, and portability. These systems are based on a large number of interconnected processing elements (PE) which perform a task in parallel. According to this approach, in the recent years several parallel cellular software environments have been developed.

The main issues that influence the way in which CA languages support the design of applications on high performance architectures are

- The *programming approach*: the unit of programming is the single cell of the automaton.
- The *cellular lattice declaration*: it is based on definition of the lattice dimension and the lattice size.
- The *cell state definition and operations*: cell state is defined as single variable or a record of typed variables; cell state access and update operations are needed.
- The *neighborhood declaration and use*: neighborhood concept is used to define interaction among cells in the lattice.
- The *parallelism exploitation*: the unit of parallelism is the cell and parallelism, like communication, is implicit.
- The *cellular automata mapping*: data partitioning and process-to-processor mapping is implicit at the language level.
- The *output visualization*: automaton global state, as the collection of the cell states, is showed as it evolves.

By addressing these issues we illustrate how this class of languages can be effectively used to implement high-performance applications in science and engineering using the massively parallel cellular approach.

15.3.1 Programming Approach

When a programmer starts to design a parallel cellular program she/he must define the structure of the lattice that represents the abstract model of a computation in terms of cell-to-cell interaction patterns. Then she/he must concentrate on the unit of computation that is a single cell of the automaton. The computation that is to be performed must be specified as the transition function of the cells that compose the lattice. Therefore, differently form other approaches, a user does not specify a global algorithm that contains the program structure in an explicit form.

The global algorithm consists of all the transition functions of all cells that are executed in parallel for a certain number of iterations (steps). It is worth to notice that in some CA languages it is possible to define transition functions that change in time and space to implement inhomogeneous CA computations. Thus, after defining the dimension (e.g., 1D, 2D, 3D) and the size of the CA lattice, she/he needs to specify, by the conventional and the CA statements, the transition function of the CA that will be executed by all the cells. Then the global execution of the cellular program is performed as a massively parallel computation in which implicit communication occurs only among neighbor cells that access each other state.

15.3.2 Cellular Lattice Declaration

As was mentioned above, the lattice declaration defines the lattice dimension and the lattice size. Most languages support two-dimensional rectangular lattices only (e.g., CANL and CDL). However, some of them, such as CARPET and Cellang, allow the definition of 1D, 2D, and 3D lattices. Some languages allow also the explicit definition of boundary conditions such as CANL that allows *adiabatic* boundary conditions where absent neighbor cells are assumed to have the same state as the center cell. Others implement *reflecting* conditions that are based on mirroring the lattice at its borders. Most languages use standard boundary conditions such as *fixed* and *toroidal* conditions.

15.3.3 Cell State

The cell state contains the values of data on which the cellular program works. Thus the global state of an automaton is defined by the collection of the state values of all the cells. While low-level implementations of CA allow to define the cell state as a small number of bits (typically 8 or 16 bits), cellular languages such as CARPET, CANL, DEVS-C++ and CDL allows a user to define cell states as a record of typed variables as follows:

```
cell = (direction :int ;
        mass      : float;
        speed     : float);
```

where three substates are declared for the cell state. According to this approach, the cell state can be composed of a set of sub-states that are of *integer*, *real*, *char* or *boolean* type and in some case (e.g., CARPET) arrays of those basic types can also be used. Together with the constructs for cell state definition, CA languages define statements for state addressing and updating that address the sub-states by using their identifiers, e.g. `cell.speed`.

15.3.4 Neighborhood

An important feature of CA languages that differentiate them from array-based languages and standard data-parallel languages is that they do not use explicit array indexing. Thus, cells are addressed with a name or the name of the cells belonging to the neighborhood. In fact, the neighborhood concept is used in the CA setting to define interaction among cells in the lattice.

In CA languages the neighborhood defines the set of cells whose state can be used in the evolution rules of the central one. For example, if we use a simple neighborhood composed of four cells we can declare it as follows

```
neigh cross = (up, down, left, right);
```

and address the neighbor cell states by the identifiers used in the above declaration (e.g., `down.speed, left.direction`). The neighborhood abstraction is used to define the communication pattern among cells. It means that at each time step, a cell send to and receive from the neighbor cells the state values. In this way implicit communication and synchronization are realized in cellular computing. The neighbor mechanism is a concept similar to the *region* construct that is used in the ZPL language [37] where regions replace explicit array indexing making the programming of vector- or matrix-based computations simpler and more concise. Furthermore, this way of addressing the lattice elements (cells) does not require compile-time sophisticated analysis and complex run-time checks to detect communication patterns among elements.

15.3.5 Parallelism Exploitation

CA languages do not provide statements to express parallelism at the language level. It turns out that a user does not need to specify what portion of code must be executed in parallel. In fact, in parallel CA languages the unit of parallelism is a single cell and parallelism, like communication and synchronization, is implicit. This means that in principle the transaction function of every cell is executed in parallel with the transaction functions of the other cells.

In practice, when coarse grained parallel machines, like clusters or multi-core, are used, the number of cells N is greater than the number of available processors P, so each processor executes a block of N/P cells that can be assigned to it using a domain decomposition approach.

15.3.6 CA Mapping

Like parallelism and communication, also data partitioning and process-to-processor mapping is implicit in CA languages. The mapping of cells (or blocks of them) onto the physical processors that compose a parallel machine is generally done by the run-time system of each particular language and the user usually intervenes in selecting the number of processors or some other simple parameter.

Some systems that run on multicomputers (MIMD machines) use load balancing techniques that assign at run-time the execution of cell transition functions to processors that are unloaded or use greedy mapping techniques that avoid some processor to become unloaded or free during the CA execution for a long period.

15.3.7 Output Visualization and Monitoring

A computational science application is not just an algorithm. Therefore it is not sufficient to have a programming paradigm for implementing a complete application. It is also as much significant to dispose of environments and tools that help a user in all the phases of the application development and execution. Most of the CA languages we are discussing here provide a development environment that allows a user not only to edit and compile the CA programs. They also allow to monitor the program behavior during its execution on a parallel machine, by visualizing the output as composed of the states of all cells. This is done by displaying the numerical values or by associating colors to those values. Examples of these parallel environments are CAMEL for CARPET, PECANS for CANL, and DEVS for DEVS-C++.

Some of these environments provide dynamical visualization of simulations together with monitoring and tuning facilities. Users can interact with the CA environment to change values of cell states, simulation parameters and output visualization features. These facilities are very helpful in the development of complex scientific applications and make possible to use those CA environments as real problem solving environments (PSEs).

Many of these issues are taken into account in parallel CA systems and similar or different solutions are provided by parallel CA languages. In Sect. 15.4 we outline some of the listed issues by discussing the main features of CAMELot, a general-purpose system that can be easily used for programming emergent systems using the CARPET cellular programming language according to a massively parallel paradigm and some related parallel CA environments and/or languages.

15.4 Cellular Automata Based Problem-Solving Environment Case Study: CAMELot and CARPET

CAMELot (CAMEL open technology) is a parallel software system designed to support the parallel execution of cellular algorithms, the visualization of the results, and the monitoring of cellular program execution [38]. CAMELot is an MPI-based portable version of the CAMEL system based on the CARPET language. CARPET offers a high-level cellular paradigm that offers to a user the main CA features to assist her/him in the design of parallel cellular algorithms without apparent parallelism [20].

A CARPET programmer can develop cellular programs describing the actions of many simple active elements (implemented by cells) interacting locally. Then, the CAMELot system executes in parallel cells evolution and allows a user to observe the global complex evolution that arises from all the local interactions. CARPET uses a C-based grammar with additional constructs to describe the rules of the transition function of a single cell. In a CARPET program, a user can define the basic rules of the system to be simulated (by the cell transition function), but she/he does not need to specify details about the parallel execution. The language includes

- a declaration part (cadef) that allows to specify:
- the dimension of the automaton (dimension);
- the radius of the neighborhood (radius);
- the type of the neighborhood (neighbor);
- the state of a cell as a record of substates (state);
- a set of global parameters to describe the global characteristics of the system (parameter).
- a set of constructs for addressing and updating the cell states (e.g., update, GetX, GetY, GetZ).

In a two-dimensional automaton, a very simple neighborhood composed of four cells can be defined as follows:

```
neighbor Stencil[2] ([-1,0]Left, [1,0]Right, [0,1]Up,
                     [0,-1]Down);
```

As mentioned before, the state (`state`) of a cell is defined as a set of typed substates that can be *shorts, integers, floats, char,* and *doubles* or *arrays* of these basic types. In the following example, the state consists of three substates.

```
state(float speedx, speedy, energy);
```

The *mass* substate of the current cell can be referenced by the predefined variable `cell_mass`. The neighbor declaration assigns a name to specified neighboring cells of the current cell and allows such to refer to the value of the substates of these identified cells by their name (e.g., `Left_mass`). Furthermore, the name of a vector that has as dimension the number of elements composing the logic neighborhood it must be associated to neighbor (e.g., `Stencil`). The name of the vector can be used as an alias in referring to the neighbor cell. Through the vector, a substate can be referred as `Stencil[i]_mass`.

To guarantee the semantics of cell updating in cellular automata the value of one substate of a cell can be modified only by the `update` operation, for example

```
update(cell_speedx, 12.9);.
```

After an `update` statement, the value of the substate, in the current iteration, is unchangeable. The new value takes effect at the beginning of the next iteration. Furthermore, a set of global parameters (`parameter`) describes the global characteristics of the system (e.g., the permeability of a soil). CARPET allows to define cells with different transition functions (inhomogeneous CA) by means of the `GetX`, `GetY`, `GetZ` functions that return the value of the coordinate X, Y, and Z of the cell in the automaton. Varying only a coordinate it is possible to associate the same transition function to all cells belonging to a plane in a three dimensional automaton.

The language does not provide statements to configure the automata, to visualize the cell values or to define data channels that can connect the cells according to different topologies. The configuration of a cellular automaton is defined by the graphical user interface (UI) of the CAMELot environment. The UI allows, by menu pops, to define the size of the cellular automata, the number of the processors onto which the automata must be executed, and to choose the colors to be assigned to the cell substates to support the graphical visualization of their values. The exclusion from the language of constructs for configuration and visualization of the data allows executing the same CARPET program with different configurations. Further, it is possible to change from time to time the size of the automaton and/or the number of the nodes onto which the automaton must be executed. Finally, this approach allows selecting the more suitable range of the colors for the visualization of data.

15.4.1 Examples of Cellular Programming

In this section we describe two examples of emergent systems expressed through cellular programming using the CARPET language. The first example is a typical CA application that simulates excitable systems. The second program is the classical Jacobi relaxation that shows how it is possible to use CA languages not only for simulate complex systems and artificial life models, but that they can be used to implement parallel programs in the area of fine grained applications such as finite elements methods, partial differential equations and systolic algorithms that are traditionally developed using array or data-parallel languages.

15.4.1.1 The Greenberg-Hastings Model

A classical model of excitable media was introduced 1978 by Greenberg and Hastings [39]. This model considers a two-dimensional square grid. The cells are in one of a *resting* (0), *refractory* (1), or *excited* (2) state. Neighbors are the eight nearest cells. A cell in the resting state with at least s excited neighbors (in the program we use $s = 1$) becomes excited itself, runs through all excited and resting states and returns finally to the resting state. A resting cell with less than s excited neighbors stays in the resting state.

Excitable media appear in several different situations. One example is nerve or muscle tissue, which can be in a resting state or in an excited state followed by a refractory (or recovering) state. This sequence appears for example in the heart muscle, where a wave of excitation travels through the heart at each heartbeat. Another example is a forest fire or an epidemic model where one looks at the cells as infectious, immune, or susceptible.

Figure 15.1 shows the CARPET program that implements the two-dimensional Greenberg-Hastings model. It appears concise and simple because the programming level is very close to the model specification. If a Fortran+MPI or C+MPI solution is adopted the source code is extremely longer with respect to this one and, although it might be a little more efficient, it is very difficult to program, read and debug.

```
#define resting 0
#define refractory 1
#define excited 2

cadef
{
  dimension 2;
  radius 1;
  state (short value);
  neighbor Moore[8] ([0,-1]North, [1,-1]NorthEast,[1,0]East,
                     [1,1]SouthEast,[0,1]South,[-1,1]SouthWest,
                     [-1,0]West, [-1,-1]NorthWest);
}
 int i, exc_neigh=0;
{
 for (i=0; (i<8) && (exc_neigh==0); i++)
   if (Moore[i]_value == excited) exc_neigh = 1;
 switch (cell_value)
 {
   case excited    : update(cell_value, recovering); break;
   case recovering : update(cell_value, resting); break;
   default         : /* cell is in the resting state */
                     if (exc_neigh == 1)
                       update(cell_value, excited);
 }
}
```

Fig. 15.1 The Greenberg-Hastings model written in CARPET

15.4.1.2 The Jacobi Relaxation

As a second example, we describe the four-point Jacobi relaxation on a $n \times n$ lattice in which the value of each element is to be replaced by the average value of its four neighbor elements. The Jacobi relaxation is an iterative algorithm that is used to solve differential equation systems. It can be used, for example, to compute the heat transfer in a metallic plate on which boundaries there is a given temperature. At each step of the relaxation the heat of each plate point (cell) is updated by computing the average of its four nearest neighbor points. Figure 15.2 shows a CARPET implementation. The initial `if` statement is used to set the initial values of cells that are taken to be 0.0 except for the western edge where boundary values are 1.0.

The Jacobi program, although it is a simple algorithm, is another example of how a CA language can be effectively used to implement scientific programs that are not properly in the original area of cellular automata. This simple case illustrates the high-level features of the CA languages that can be also used for implement applications that are based on the manipulation of arrays such as systolic algorithms and finite elements methods.

For the Jacobi algorithm we present some performance benchmarks that have been obtained by executing the CARPET program using different grid sizes and processor numbers. Table 15.2 shows the execution times for 100 relaxation steps for three different grid sizes (100×200, 200×200 and 200×400) on 1, 2, 4, 8 and

```
cadef
{
 dimension 2;
 radius 1;
 state ( float elem );
 neighbor Neum[4]([0,-1]North,[-1,0]West,[0,1]South,[1,0]East);
}
   int sum;
{
 if (step == 1 )
   if (GetY == 1)
     update (cell_elem, 1.0);
   else
     update (cell_elem, 0.0);
 else
   {
    sum = North_elem+South_elem+East_elem+West_elem;
    update (cell_elem, sum/4);
   }
}
```

Fig. 15.2 The Jacobi iteration program written in CARPET

Table 15.2 Execution time (in) of 100 iterations for the Jacobi algorithm

Grid sizes	1 Proc	2 Procs	4 Procs	8 Procs	10 Procs
100×200	1.21	0.65	0.37		0.25
200×200	3.62	1.25	0.67	0.42	0.37
200×400	8.22	3.65	1.26	0.74	0.62

10 processors of a multicomputer. From the figure we can see that as the number of used processors increases, there is a corresponding decrease of the execution time. This trend is more evident when larger grids are used; while smaller CA do not use efficiently the processors. This means that, because of the algorithm simplicity, when we run an automaton with a small number of cells we do not need to use several processing elements. On the contrary, when the number of cells in the lattice is high, the algorithm benefits from the use of a higher number of computing resources. This can be also deduced from Table 15.3 that shows the relative speed up results for the three different grids. In particular, we can observe that when a 200×400 lattice of cells is used we obtain a superlinear speed up in comparison to the sequential execution mainly because of memory allocation and management problems that occur when all the 80,000 cells are allocated on one single processing element.

Table 15.3 Relative speed up of the Jacobi algorithm

Grid Sizes	1 Proc	2 Procs	4 Procs	8 Procs	10 Procs
100×200	1	1.86	3.27		4.84
200×200	1	2.89	5.40	8.62	9.78
200×400	1	2.25	6.52	11.10	13.25

15.5 Cellular Automata Based Problem-Solving Environment Case Study: CAME$_{\&}$L

The environment CAME$_{\&}$L [14, 21, 23] resulted from a collaboration between the Saint-Petersburg State University of Information Technologies, Mechanics and Optics (Russian Federation) and the Section Computational Science of the University of Amsterdam (The Netherlands).

15.5.1 Cellular Automata Based Computational Experiment Decomposition

The initial idea of "CAME$_{\&}$L" was to distribute the implementation of a computational experiment among the functional parts. Any simulation should be assembled as a set of interacting components of definite types. A researcher will be able to use them in miscellaneous combinations to add arbitrary functionality to the experiment. Components could be taken from the standard set or created by a user to fulfil the target problem requirements.

Consequently the CA based computational experiment decomposition [14, 21] was offered. It was decided to distinguish five types of components. Names of these types are shown with bold in the following list.

- The **grid** implements the visualization of automaton's state and the navigation among cells. It does not actually store cells states. This component's main task should be drawing and interacting with user.
- The **datum** provides cells states storage, exchange and some aspects of the data visualization. Namely it can define
  - the association of cells' states with colors, which will be used for their displaying;
  - the custom single cell drawing routine.
- The **metrics** provides the relationship of neighborhood, coordinates for each cell and distance measurement functions. Implementation of metrics as a separate component instead of entrusting its functions to the grid or the datum allows, for example, to use non-standard coordinate systems, like generalized coordinates [34].
- The **rules** describes computations and controls the iteration. In the introduced ideology terms "rules" and "transition function" are not synonyms. Components of this type define much more: the method of parallelization, methods of computations' optimization (if any are used), many other aspects and the transition function among the rest. This component also should allow
  - to handle experiment's start up (proceed the initialization);
  - to determine and check the criteria of experiment's completion;
  - to handle experiments finish (proceed the finalization);

- to define special tools for checking, changing, pre- and postprocessing;
- to define important experiment's properties for further studying with the help of analyzer components (see below).

- The **analyzer** allows to keep an eye on definite properties of the experiment, draw graphs, create reports, monitor values and all of this kind.

The union of compatible components of first three types totally define a "functionless" cellular automaton. Addition of a component of the fourth type will form a cellular automaton that can perform the computational experiment. Only single instances of the grid, the datum, the metrics and the rules are able to participate in the simulation, but it can involve arbitrary amount of analyzers (even none).

Components are continuously interacting during the whole computational experiment to do the work together. Obviously, each component cannot cooperate with arbitrary another component, but only with one, which is suitable for this. Such compatibility conditions for analyzers are trivially based on the examination of the analyzable parameter's variable data type. For the rest four types there should be a special language of logical expressions to describe their properties and requirements. In this case requirements should represent conditions imposed on properties.

Each component should have specific user interface: the declared set of available parameters, which allow to setup the component for the particular problem and for the accordance to user's needs and preferences.

15.5.2 Software Design

Taking everything, said in Sect. 15.5.1 into account it was decided to implement the software using C++ language. All basic statements, listed above, can be provided with the help of the object-oriented programming paradigm. Windows was chosen as a target operating system. Consequently each component should be represented as a dynamic-link library, developed in the framework of the predefined programming interface. The component's library have to contain the class, which implements the functionality corresponding to one of five types, listed in Sect. 15.5.1.

As a result, CAME$_{\&}$L software consists of three conceptually and functionally interconnected parts:

- **CADLib** or "Cellular Automata Development Library" is the C++ class library, which is designed to present an easy-to-use and rich set of instruments for implementing computational experiments according to given regulations and using definite abstractions. It provides basic classes for all types of components, parameters and for other concepts.
- **Standard components** are most common building blocks of computational experiments, which can be considered as both: ready-made solutions and examples for studying when one is going to create his own component. They also can be reused and extended to fulfil the needs of the researcher.

- The **environment** is the application with rich user interface for simulations and research with the help of cellular automata. It provides the access to tools for the simulation control, studying and analysis, cluster arrangement, workstations management and many other purposes. Important note is that the environment itself contains no computational functionality, but allows to execute components' libraries in the definite software surrounding.

One may say that the ability to use C++ is a too complicated skill to demand it from the researcher. This is true, but at the same time this is totally in the ideology of the extensible environment: the scope of the rules basic class is much wider than just the transition function definition. So one can create a rules component, which represents the parser for the automaton's iteration description from the specific language. This means that one rules component is able to implement not just the single transition function, but the class of such functions. This ideology allows to incorporate arbitrary amount of specific computations description languages into one software and provide specialists from distinct field of the research with the component, which supports necessary abstractions from the given subject field. The code snippets, the rich set examples, scripts and the CADLib itself are provided to make the components creation simpler.

15.5.3 Usage Example. Tumor Growth Modeling

Now CAME$_{\&}$L is intensively used for the 3D tumor growth modeling. In this section a very schematic example will illustrate the common approach to using this software for a simulation. The example is related to the tumor growth simulation, but is free of plunging into the biological background. Computational Oncology is an active area of research with many promosing results. For instance, Sottoriva et al. [22] report on extensive simulations to reveal Cancer's stem cell driven tumor growth using such models.

To implement the cellular automaton, which will perform modeling, one should select the set of at least four components (grid, datum, metrics and rules), which will arrange the experiment. If particular component is not presented in the set of standard components then it should be created.

Usually, there is no need to create user analyzers, because they are much less problem domain dependant than any others. That is why only grid, datum, metrics and rules components are considered in the list below. The component type's name is shown with bold.

- For performing the computational experiment of 3D tumor growth, the standard **grid** component "Basic 3D Grid" will be suitable. It supports many functions, which are extremely useful for the model of such solid clot: drawing sections, stubs and slices to take a look inside the tumor.
- In the experiment, each automaton's cell is to represent single biological cell, which should be described with distinct user developed data structure. Let's

assume that it is called `BioCell` (there is no need to discuss what it consists of). There is no standard **datum** component implementing 3D storage for cells which contains instances of the `BioCell` structure. This component should be created with the help of CADLib as a descendant of the `CADatum` class. Library makes it extremely easy, providing `CABasicCrts3DDatum` class template, which automatically implements the majority of needed functions. Primitive, but functional class declaration should look like shown of Fig. 15.3. Numbers, given in brackets at the left, are used for further referring to appropriate lines or sections (sets of lines from one number to another) of the code and have no attitude to the source.

On line (1) and following one the parent class template is used with the specific values of parameters: first one is the data type to be stored in each cell, second – the class of the user interface dialog (may be none), used to edit the values of a stored data type, third – the resource identifier of the dialog template. Section, started from line (2) contains constructor and destructor declarations. The main task, which is entrusted to the constructor, is the initialization of component's parameters. Section, started from line (3) presents component's self-introduction functions, treating macrodefinitions, provided by CADLib. These declarations contains (in the same order) components short name, longer description, resource identifier of the corresponding icon, requirements for the properties of other components to be compliant to this one and properties, implemented by this components.

Last two statements worth special discussion. Attributes and requirements specifications are formulated using the trivial language of consequently adjustable properties. The self-characteristics, given on the last line of section (3) should be understood as the declaration of the fact that the component implements the property "Data". Then it is refined: data is "composite". Moreover, composite data is attributed as "biocell". In the same manner requirements represent

```
    class CACrtsCell3DDatum:public
(1)     CABasicCrts3DDatum
        <BioCell, CBioCellDlg, IDD_BIOCELL> {
    public:
(2)     CACrtsCell3DDatum();
        virtual ~CACrtsCell3DDatum();

(3)     COMPONENT_NAME(BioCells for Cartesians 3D)
        COMPONENT_INFO(3D storage for cellular (biological)
          data for cartesian metrics)
        COMPONENT_ICON(IDI_ICON)
        COMPONENT_REQUIRES(Metrics.3D.cartesian.*)
        COMPONENT_REALIZES(Data.composite.biocell)

(4)     virtual inline COLORREF GetCellColor(CACell c); (5)
(5)     virtual inline void SetDefValue(CACell c);
    };
```

Fig. 15.3 The declaration of the datum component for the tumor growth modeling in CAME$_{\&}$L

the conditions over properties, allowing wildcards and logical operations. This component needs to collaborate with another one, which should implement the property “Metrics”. The property should be attributed as “3D” and, moreover, “cartesian”. The asterisk means that any amount of deeper refining subproperties will fit. There is no strict rule for properties naming. The properties conformance checkup is case-insensitive. The union of components will not form a proper cellular automaton if at least one component has unsatisfied requirements.

There is no need to overload any additional members of the `CADatum` class, because all the required functionality is basically implemented by the `CABasicCrts3DDatum` class template. Nevertheless most likely one will decide to overload two functions, shown on lines (4) and (5).

First of all, note that `CACell` class represents the universal cell identifier, which allows to refer any given cell in the arbitrary metrics. From the technical point of view, it represents the 64 bits integer value. For example, when dealing with standard 2D cartesian metrics the universal cell identifier stores cell’s absciss in first 32 bits and the ordinate in rest 32 bits. For standard 3D cartesian metrics the universal cell identifier is divided into three unequal parts: 22 bits for absciss, 21 for ordinate and 21 for applicate. When using generalized coordinates [34] as, for example, Peano-curve-based metrics [35], the universal cell identifier is interpretted as a solid unsigned integer number. So, CAME$_{\&}$L can govern the cellular automaton of up to 2^{64} cells.

The function, overloaded on line (4), is to return the color, which should be used to visualize the value, stored in the cell `c`. The function on line (5) should put the “default value” to the cell `c`. This value will be used for the grid initialization and as the out-of-bounds value for constant boundary conditions.

Finally, the component’s library should contain the class declaration, the implementation (in the case of this component, four functions should be implemented: constructor, destructor and two, declared on lines (4) and (5)) and component’s library access functions, which can be easily created with following two lines of code:

```
COMPATIBLE_DATUM(1.1)
DATUM_COMPONENT(CACrtsCell3DDatum)
```

First one implements the authentication function for the library, which says that the component was built to be compatible with CADLib version 1.1. Second one adds the creation and the destruction functions for the component, implemented by the `CACrtsCell3DDatum` class.

- It is logical to perform modeling in Cartesian **metrics**, which is implemented by one of the standard components. The name of this component is “Cartesians 3D”.
- Each **rules** component should be implemented by the descendant of the `CARules` class. This type of components was designed to allow the full control over the simulation and to support a lot of features. Nevertheless for the plain implementation of algorithm in most cases it’s enough to overload its `SubCompute` function only. This function represents the transitions’ laws, which are applied to some zone of the grid. The description of the zone is given by the object of

```
    bool CATG3DRules::SubCompute(Zone& z)
    {
(1)     *** Prestep ***

(2)     int i,j,k;
        for(i=(int)z.a1; i<=(int)z.b1; i++) {
(3)         pEnv->SetProgress(((double)i-z.a1)/(z.b1-z.a1+1))
            for(j=(int)z.a2; j<=(int)z.b2; j++)
                for(k=(int)z.a3; k<=(int)z.b3; k++) {
(4)                 *** Transition for the cell (i;j;k) ***
                    *** Compute the analyzable values ***
                }
        }

(5)     *** Poststep ***

(6)     *** Compute the analyzable values ***
        *** Assign the values to the analyzable parameters ***

(7)     pEnv->SetProgress(1.0);

(8)     return (*** Criteria of the completion ***); }
```

Fig. 15.4 The schematic representation of the tumor growth modeling algorithm implemented in CAME$_{\&}$L

CADLib's `Zone` class passed as the parameter to the function. If a variable `z` describes the zone, then `z.a1` and `z.b1` are the lower and the higher boundaries of the zone along a first axis, `z.a2` and `z.b2` – along a second one and `z.a3` and `z.b3` – along a third one. All boundaries should be included. Axes are just enumerated, but not named here as the "absciss", the "ordinate" and the "applicate", because zonal mechanism is to be metrics independent and in general situation the meaning of the particular axis is unknown. So, it would be wrong to conclude that the object of the `Zone` class always describes the parallelepiped.
The environment will call the `SubCompute` function with the correct value of the zone description. In most cases, and in the case of tumor growth modeling also, the main structure of the implementation of this function should look like shown in Fig. 15.4 (verbal descriptions of the functionality which replace the code are given between three asterisks, bracketed numbers are used for referencing, as above).

This function will be called once for each timestep. So its beginning (line (1)) is the appropriate place for the prestep routines (initialization of variables, precomputing values, which will be used later, etc.). On the line (2) three variables, which will run over three Cartesian coordinated are declared. The running is provided by the following loop operators. Inside the loop (line (4) and the next one) the main part of an algorithm should be placed: for each cell its new state should be determined. Moreover, values of the analyzable parameters, which can be influenced by each single cell, should be updated. It is strongly recommended not to reassign the values to such parameters many times, but to deal with the temporary variables until step will be finished. Line (5) is the appropriate place for postprocessing step results. On line (6), after the analyzable values, which are influenced by the simulation step in general (not by any single cell) were calculated, the parameters can get their values for the current iteration (line after (6)).

Member variable `pEnv` allows the rules component to exchange the information with the environment. Its member function `SetProgress` is used to declare which part of the time-consuming process have been accomplished (from 0.0 (nothing have been done) to 1.0 (the process is finished)). The line (3) is the suitable place to report about the progress not excessively often, but adequately. Before finishing the iteration progress should be set to 100% (line (7)).

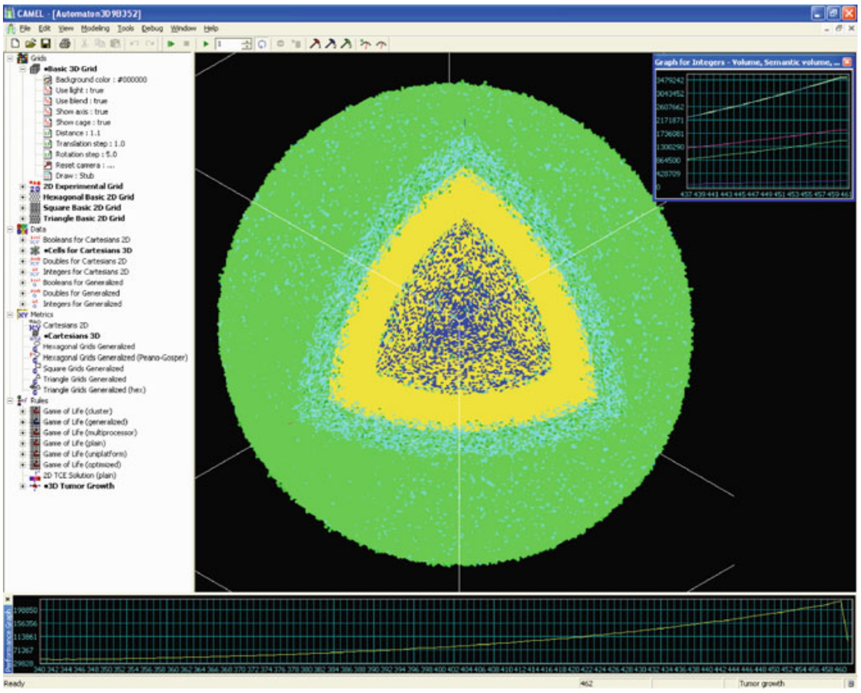
The value returned by `SubCompute` function (line (8)) plays the role of the computational experiment's completion criterium. Simulation will go on while function returns `true`.

In all the rest a rules component's library should contain the same principal parts as a datum component's library, considered above. Necessary component's library access functions can be also created by two lines of the code.

The situation, which has been considered is quite typical: in the overwhelming majority of cases, excluding purely educational purposes, a user has to create the rules component. In some cases, but not so often she/he has to implement the datum component also. The chance that one will need the non-standard metrics is very low and most likely attitudes to the special metrics-related research. A necessity of new grids or analyzers creation may rise even more rare. Standard analyzers can treat all basic types (boolean, integer and floating-point variables) and standard grids are suitable for 1D, 2D and 3D modeling. Moreover, the visualization can be slightly influenced or customized on the level of datum components (see the description of the `DrawCell`, `GetCellColor`, and `GetPlaceColor` member functions of the `CADatum` class [14, 23]).

From the opposite side, lets sort types of components in the order from the most simple to the most complicated one from the developers point of view. In this case the creation of new datum components will be the simplest. Then rules components follow. It looks not so simple, but in most cases the only thing, which researcher has to do is overloading the `SubCompute` member function. The rest three types are to be created form scratch and number of functions have to be overloaded. The next from the simplicity point of view are analyzer components. Their idea is quite clear and general, it contains less specifics and can be implemented easier than the next type – grid components. Metrics components are the most complex and hard to debug, because they make no visual output, but with the help of CADLib even this can be done without getting stuck.

To run the tumor growth simulation a researcher has to install two created components with the help of the "components manager" built into the environment ("Tools" | "Components Manager..." in the main menu). Then new document should be created ("File" | "New" in the main menu) and four components mentioned above


CAMEL - [Automaton3D9B352]
File Edit View Modeling Tools Debug Window Help
Graph for Integers - Volume, Semantic volume, ...
Performance Graph
Ready
462
Tumor growth

```
Grids
  •Basic 3D Grid
    Background color : #000000
    Use light : true
    Use blend : true
    Show axis : true
    Show cage : true
    Distance : 1.1
    Translation step : 1.0
    Rotation step : 5.0
    Reset camera : ...
    Draw : Stub
  2D Experimental Grid
  Hexagonal Basic 2D Grid
  Square Basic 2D Grid
  Triangle Basic 2D Grid
Data
  Booleans for Cartesians 2D
  •Cells for Cartesians 3D
  Doubles for Cartesians 2D
  Integers for Cartesians 2D
  Booleans for Generalized
  Doubles for Generalized
  Integers for Generalized
Metrics
  Cartesians 2D
  •Cartesians 3D
  Hexagonal Grids Generalized
  Hexagonal Grids Generalized (Peano-Gosper)
  Square Grids Generalized
  Triangle Grids Generalized
  Triangle Grids Generalized (hex)
Rules
  Game of Life (cluster)
  Game of Life (generalized)
  Game of Life (multiprocessor)
  Game of Life (plain)
  Game of Life (uniplatform)
  Game of Life (optimized)
  2D TCE Solution (plain)
  •3D Tumor Growth
```

Fig. 15.5 Screenshot of CAME$_{\&}$L, running tumor growth computational experiment, being studied with the help of two analyzers

(two standard and two created) should be chosen. After this the simulation can be executed with the help of "Go" button ("Modeling" | "Go" in the main menu). The screenshot of the environment, running the tumor growth computational experiment is shown on Fig. 15.5.

The experiment's window is divided into two parts. The left one displays the components tree. All components except analyzers are presented there and grouped by types. Tree's leaves of the first level are types' names, on the second level there are the components, and their parameters are on the third one. This tree is handy for fast switching between the components. Those of them, which were selected, are marked out with the small circle in the beginning of the name. Components, which are compatible with the currently chosen instances, are shown with bold font.

In the right part of the experiment's window the grid component is visualizing the simulation. In the shown case the multicellular tumor spheroid is represented as a "stub". This means that cells with positive values of all three coordinates are not drawn, to allow looking inside the formation.

At the bottom and at the upper-right corner there are two analyzer graphs: the performance one and the plot of key tumor growth characteristics (volume, amounts of proliferating, quiescent and dead cells).

15.6 Conclusions

The main goal of programming languages and tools has always been to make the programmer more productive and the programming task more effective. Appropriate programming languages and tools may drastically reduce the costs for building new applications as well as for maintaining existing ones.

It is well known that programming languages can greatly increase programmer's productivity by allowing the programmer to write high-scalable, generic, readable and maintainable code. Also, new domain specific languages, such as CA languages, can be used to enhance different aspects of software engineering.

The development of these languages is itself a significant software engineering task, requiring a considerable investment of time and resources. Domain-specific languages have been used in various domains and the outcomes have clearly illustrated the advantages of domain specific-languages over general purpose languages in areas such as productivity, reliability, and flexibility.

The main goal of the paper is answering the following question: How does one program emergent systems through cellular automata on parallel computers? We think that it is very important for an effective use of cellular automata for computational science on parallel machines to develop and use high-level programming languages and tools that are based on the cellular computation paradigm. These languages may provide a powerful tool for researchers and engineers that need to implement real-life applications on parallel machines using a fine-grain approach. This approach allows designers to concentrate on "how to model a problem" rather than on architectural details as occurs when people use low-level languages that have not been specifically designed to express fine-grained parallel cellular computations.

In a sense, parallel cellular languages provide a *high-level paradigm* for fine-grain computer modeling and simulation. While efforts in sequential computer languages design focused on *how* to express sequential data, objects and operations, here the focus is on finding out *what* parallel cellular objects and operations are the ones we should want to define. Parallel cellular programming emerged as a response to these needs.

References

1. J. von Neumann, *Theory of Self-Reproducing Automata*, ed. by A. W. Burks (University of Illinois Press, Urbana, IL, 1966)
2. S. Ulam, *Random Processes and Transformations / Proceedings of the International Congress of Mathematicians*, vol. 2 (American Mathematical Society Providence, RI 1952).
3. K. Zuse, *Calculating Space*. (Massachusetts Institute of Technology Technical Translation AZT-70-164-GEMIT (Project MAC)). (MIT Cambridge, MA, 1970)
4. N. Wiener, A. Rosenbleuth, The mathematical formulation of the problem of conduction of impulses in a network of connected excitable elements, specifically in cardiac muscle. Archi. Insti. Cardiol. Mex. **16**, 202–265 (1946)
5. P.M.A. Sloot, A.G. Hoekstra, *Modeling Dynamic Systems with Cellular Automata, Chapter 21*. ed. by P.A. Fishwick, Handbook of Dynamic System Modeling. (Chapman & Hall/CRC, London/Boca Raton, FL, 2007)

6. E. Houstis, E. Gallopoulos, J. Bramley, J.R. Rice, Problem-solving environments for computational science. IEEE Comput. Sci. Eng. **4**, 18–21 (1997)
7. E. Gallopoulos, E. Houstis, J.R. Rice, Computer as Thinker/Doer: Problem-solving environments for computational science. IEEE Comput. Sci. Eng. **1**, 11–23 (1994)
8. M. Abrams, D. Allison, D. Kafura, C. Ribbens, M.B. Rosson, C. Shaffer, L. Watson, *PSE Research at Virginia Tech: An Overview. Technical Report: TR-98-21*. (Virginia Polytechnic Institute & State University, Blacksburg, VA, 1998)
9. D.W. Walker, M. Li, O.F. Rana, M.S. Shields, Y. Huang, The software architecture of a distributed problem-solving environment. Concur. Pract. Exp. **12**, 1455–1480 (2000)
10. D.E. Knuth, *Literate programming. Center of the Study of Language and Information*. (Stanford, CA 1992)
11. E. Gallopoulos, E.N. Houstis, J.R. Rice, *Future Research Directions in Problem Solving Environments for Computational Science: Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science. Technical Report 1259. Center for Supercomputing Research and Development*. (University of Illinois, Urbana-Champaign, IL, 1992)
12. W. Schroeder, K. Martin, B. Lorensen, *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 4th edn. (Kitware, New York, NY, 2006)
13. T. Worsch, Programming Environments for Cellular Automata. *Proceedings of 2nd Conference on CA in Research and Industry (ACRI 96)* (Springer, Heidelberg, 1996)
14. L. Naumov, Generalized coordinates introduction method and a tool for computational experiments software design automation, based on cellular automata. PhD Thesis, SPbSU ITMO, Saint-Petersburg, 2007
15. I. Blecic, A. Cecchini, G. Trunfio, A generalized rapid development environment for cellular automata based simulations. *Cellular Automata: 6th International Conference on Cellular Automata for Research and Industry (ACRI-2004)*. (Springer, Heidelberg, 2004) pp. 851–860
16. T. Toffoli, N. Margolus, *Cellular Automata Machines: A New Environment For Modeling*. (MIT Press, Cambridge, MA, 1987)
17. N. Margolus, CAM-8: A Computer Architecture Based on Celluar Automata. *Physics of Computation Seminar* (MIT, Cambridge, MA, 1993)
18. M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, D. Talia, A parallel cellular automata environment on multicomputers for computational science. Parallel Comput. **21**, 803–823 (1995)
19. G. Spezzano, D. Talia, CARPET: a programming language for parallel cellular processing. *Proceedings 2nd European School on PPE for HPC*. (Alpe d'Huez, France, 1996) pp. 71–74
20. G. Spezzano, D. Talia, A high-level cellular programming model for massively parallel processing. *2nd International Workshop on High-Level Programming Models and Supportive Environments (HIPS97)*. IEEE Computer Society Press, LOS Alamitos, CA, pp. 55–63
21. L. Naumov, CAME$_{\&}$L – Cellular Automata Modeling Environment & Library. *Cellular Automata: 6th International Conference on Cellular Automata for Research and Industry (ACRI-2004)*. (Springer, Heidelberg, 2004) pp. 735–744
22. A. Sottoriva, J.J.C. Verhoeff, T. Borowski, S.K. McWeeney, P.M.A. Sloot, L. Vermeulen, Modelling cancer stem cell driven tumor growth reveals invasive morphology and increased phenotypical heterogeneity. Cancer Res. **70**, 46–56
23. CAMEL Laboratory – http://camellab.spb.ru/. Accessed date 23 Feb 2005
24. G. Spezzano, D. Talia CAMELot: A parallel cellular environment for modelling complexity. AI*IA Notizie **2**, 9–15 (2001)
25. J.D. Eckart, A cellular automata simulation system: Version 2.0. ACM SIGPLAN Notices **27**(8), 99–106 (1992)
26. U. Freiwald, J.R. Weimar, JCASim a Java system for simulating cellular automata. *Theoretical and Proctical Issues on Cellular Automata (ACRI 2000)*. (Springer, Heidelberg, 2001) pp. 47–54
27. Mirek's Cellebration – http://www.mirekw.com/ca/. Accessed date 27 Apr 2010
28. M. Ifrim, Contribution to ParCeL-6 Project: Design of Algorithms Mixing Memory Sharing and Message Passing Paradigms for DSM and Cluster Programming, 2005

29. O. Menard, S. Vialle, H. Frezza-Buet, Making cortically-inspired sensorimotor control realistic for robotics: Design of an extended parallel cellular programming model. *In International Conference on Advances in Intelligent Systems - Theory and Applications*. (IEEE Computer Society, Luxembourg, 2004)
30. T. Bach, T. Toffoli, SIMP, a laboratory for cellular automata and lattice gas experiments. *International Conference on Complex Systems*, (Boston, MA, 2004)
31. T. Toffoli, T. Bach, A common language for "Programmable Matter" (Cellular Automata and All That). Bull. Ital. Assoc. Artif. Intell., **2**, 23–31 (2001)
32. H. Chou, W. Huang, J.A. Reggia, The trend cellular automata programming environment. Simulation **78**(2), 59–75 (2002)
33. C. Hochberger, R. Hoffmann, CDL – a language for cellular processing. *Proceedings of the 2nd International Conference on Massively Parallel Computing Systems*, IEEE, Ischia. pp. 41–46 (1996)
34. L. Naumov, Generalized Coordinates for Cellular Automata Grids. Computational Science – ICCS 2003. Part 2. (Springer, Heidelberg, 2003) pp. 869–878
35. H. Sagan, *Space-Filling Curves*. (Springer, Heidelberg, 1994)
36. D. Talia, Cellular processing tools for high-performance simulation. Computer **33**(9), 44–52 (2000)
37. B.L. Chamberlain, S-E. Choi, S.J. Deitz, L. Snyder, The high-level parallel language ZPL improves productivity and performance. *In: Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing* (2004), Madrid
38. G. Spezzano, D. Talia, Programming cellular automata for computational science on parallel computers. Future Gen. Comput. Syst. **16**(2–3), 203–216 (1999)
39. J.M. Greenberg, S.P. Hastings, Spatial patterns for discrete models of diffusion in excitable media. SIAM J. Appl. Math. **34**, 515–523 (1978)