Future Generation Computing Systems (2002)

The Java based cellular automata simulation system – JCASim

Uwe Freiwald and Jörg R. Weimar *

Institute of Scientific Computing, Technical University Braunschweig, D-38092 Braunschweig, Germany

Abstract

The software system JCASim is a system for simulating cellular automata. It can simulate cellular automata (CA) with different geometries, different state sets (including state sets structured by the use of state components with different types), boundary conditions and initial conditions. The state transition function can be specified using a graphical user-interface, using Java (by programming only four methods), or in CDL, a special-purpose CA language. Cells can be represented by text, colors, or icons. JCASim also provides support for block-CA and for asynchronous models. The system is written entirely in Java to ensure portability. Here we describe the simulation system and show a number of examples, which demonstrate the ease of programming CA with JCASim. Finally, some issues surrounding execution speed are discussed.

Key words: cellular automata, simulation, Java

1 Introduction

The concept of cellular automata is about fifty years old. In this period of time, a large number of people have written programs to simulate cellular automata (CA). Most of these programs were written to simulate one specific CA, but a significant number of simulation systems have been created for the simulation of "any" cellular automaton. An overview is given in [1]. Two systems we would like to single out developed around specially created languages for the description of cellular automata: *cellang* and *CDL*. The language *cellang* and

* to whom correspondence should be addressed.
Email address: J.Weimar@tu-bs.de (Jörg R. Weimar).
URL: http://www.jcasim.de (Jörg R. Weimar).

the system *cellsim*, developed by J. Dana Eckart [2] was originally strongly Unix-based, but is now available also for Windows. In the language *cellang* the cellular automata are restricted to cubic lattices (any dimensions), finite state sets, and otherwise traditional CA. *cellang* extends the CA concept by the concept of "agents", which are objects that move around the lattice.

The language *CDL* [3] was developed first for translation into hardware simulation systems (of the CEPRA family [4]), but a software simulation system was also developed around the language. *CDL* allows a number of extensions to the strict CA approach, namely the use of floating point and integer variables in the state set, uses only cubic lattices, simplifies coding by structured data types in the state, and introduces (in *CDL++*) moving objects as an extension (which is translated into standards CA constructs) [5].

2 Distinguishing features of JCASim

The features distinguishing the new system JCASim [6] from other cellular automata simulation systems are the following:

Platform independence through Java: The JCASim system is completely coded in Java, which means that it runs on all modern operating systems. Other CA simulation systems do include the possibility to generate a Java applet from the description of the CA (e.g., in *CDL*), but there the translation system itself is not universally portable. In JCASim, the simulation as well as all translation tools are coded in Java and therefore portable.

Coding of the CA in Java or *CDL*: In JCASim, the description of the state and the state transition function are coded in Java, *CDL*, or *cellang*. The initial conditions are also specified in the Java or *CDL* code (using an extension to *CDL*). The other choices, such as boundary conditions, lattice geometry, neighborhood, etc., can be specified interactively in the simulation system.

Support for block-CA: Block-CA are a class of cellular automata where the updating rule specifies how a block of cells change their state together, instead of specifying how one cell changes depending on the neighbors [7]. They are formally equivalent to regular CA, but are much more convenient to formulate in many cases, especially where conservation laws must be observed [8]. Block-CA are directly supported in JCASim.

Use of icons for representing cell states: The state of a cell can be represented using descriptive text, colors, or icons. The easy-to-use support of icons is a new feature of JCASim. The user prepares an image that contains an array of the icons used for representing the state. In (a new extension to) the *CDL* code it is then specified which icon in this image is to be used to represent the state of a given cell (see Figures 4 and 5 below).

State of a cell is object: The JCASim system encapsulates data access such that the basic object is the state of a cell. Accesses to neighboring cells occur through special neighborhood access functions, which ensures the uniformity of the lattice of cells.

3 Choices in constructing CA

A large number of different possibilities can be selected in using CAs for simulation[9]. Here we discuss the most important choices and note which options are supported by JCASim.

Geometry: Uniform regular tilings can be 1-dimensional, 2-dimensional square, hexagonal, or triangular, 3-dimensional, or higher-dimensional. In JCASim cubic lattices in 1 to 3 dimensions are supported, as well as hexagonal and triangular 2-D grids.

Boundary conditions: JCASim supports periodic, reflective, and constant boundaries, which can be selected separately for each boundary. An extension CAComb [10,11] also supports a special kind of boundary for coupling different CA. Note that *CDL* uses a different concept for handling boundaries, which is not implemented or translated in JCASim: In *CDL* the special key word "`border`" takes on a value that gives the distance from the boundary for cells where the boundary is in the neighborhood. We prefer to use special constant boundary values for this purpose, since this concept is more universal and easier to understand.

State set: The normal definition of CA requires the state set to be finite. In CDL, the cell state can contain (theoretically unlimited) integers and floating point variables. The state in *CDL* can be structured by the use of enumeration variables, records, and unions. All of these possibilities are supported by the Java system as well (unions and records are translated to inner classes).

Transition function: Any language constructs provided by CDL and Java can be used in describing the transition function. This also allows constructs that would be forbidden by a strict definition of cellular automata (such as recursion or loops with an undetermined number of iterations) and should therefore be avoided in this context.

Initial conditions: The specification of the state class in Java can contain a method "`initialize`" that allows the user to set the cells of the lattice to any initial configuration required by the problem. The following extension to CDL makes it possible to also formulate many initial conditions without resorting to Java programming: In *CDL*, a new section is introduced with the keyword "`initial`". Following this keyword are pairs of values and conditions of the form

value $\sim$ *condition*

where *value* is a constant record that assigns a value to each field of the cell state, and *condition* is a boolean expression which may contain the special variables `x`, `y`, `z` and `lx`, `ly`, `lz`, where `x`, `y`, `z` specify the position of the cell in the lattice of size `lx*ly*lz`. The record *value* may also contain these variables, and the first matching pair is used to assign the initial state.

Global variables: In Java, variables may be declared "static", which means that to there is only one copy of the variable shared by all instances of a class. In the context of cellular automata, this construct allows global communication between cells, which should be avoided for normal CA. A special case is the use of read-only global variables to distinguish different phases of the CA. To support this, JCASim has a special transition rule "global transition" which is called only once per time step and can be used to update such global phase variables, but cannot read the state of any cell. A corresponding extension in CDL is the introduction of a "`global`" section to declare global variables, and a named rule "`rule global`" to change them.

Moving objects / agents: Moving objects or agents are not currently supported in JCASim. A *CDL++* description pre-processed into a *CDL* description can be used in JCASim to simulate problems requiring moving objects.

4 Structure of JCASim

The system JCASim uses an object-oriented structure. An overview over the structure is shown in the UML diagram in Figure 1. The design was guided

by the principle that each class should only contain the data and provide the methods for correct coding of a cellular automaton. For example, a cell in the CA should not have access to its absolute position in the lattice, but only to the state of its neighbors. The second guideline was that the user should only have to create one class for the specification of a complete CA including initialization and visualization. This aim contradicts the strict object-oriented structure to some extend, as can be seen in the method `initialize` of `State`, but the advantages outweigh the disadvantages in this case.

The main interface to start a simulation is `CellularAutomaton`, which contains a `Lattice` and has methods to set the lattice type and cell state type, initialize the lattice, and execute a number of steps. The class `CALocal` is a simple implementation of this interface for CA which reside completely on one computer (unlike the CA described in the extension CAComb [10,11]). The `Lattice` can be one of several different subclasses, depending on the dimension and geometry of the lattice. The different subclasses have methods to access a cell with given coordinates, or to collect the neighbors of a given cell according to one of several definitions of neighborhood (vonNeumann, Moore, user-defined). Specialized lattice classes exist for asynchronous cellular automata (e.g., all those described in [12]), for multi-threaded simulation on parallel shared-memory computers, or for other user-defined purposes.

The lattice contains a (computationally) rectangular array of `Cell`s. `Lattice` has the main methods `reset()` to initialize the lattice, `backup()` and `transition()` to execute one backup or transition step on each cell, and `getState()` to access a specific `State` (using the global coordinate). Each `Cell` represents one cell of the CA and depending on the type of CA, it contains one or two `State` objects which store the cell state. Two `State` objects are used to ensure the synchronous application of the transition function on a sequential computer. For this purpose, one step in the CA is split into two parts: First, a `backup()` phase, in which all the variables of the State are copied to the backup-State, then a `transition()` phase, in which the new states are calculated from the current state and the states of the neighbors. Accesses to the neighbor state use the backup-state of the neighbor, which remains constant during this phase, thus ensuring the synchronous updating semantics of cellular automata. The use of a backup state is not necessary for block-CA (where different blocks do not interact during the update phase, and the interaction within a block is under the control of the user) and for asynchronous variants of CA [12].

The `State` is the class written by the user and contains all the information specific to a special CA, apart from configuration options specified in the simulation system. The user writes a subclass of `State` which contains variables to be held in each cell and contains the method for changing the state in one time step (`transition(Cell)`). Access to the neighbors is through a method

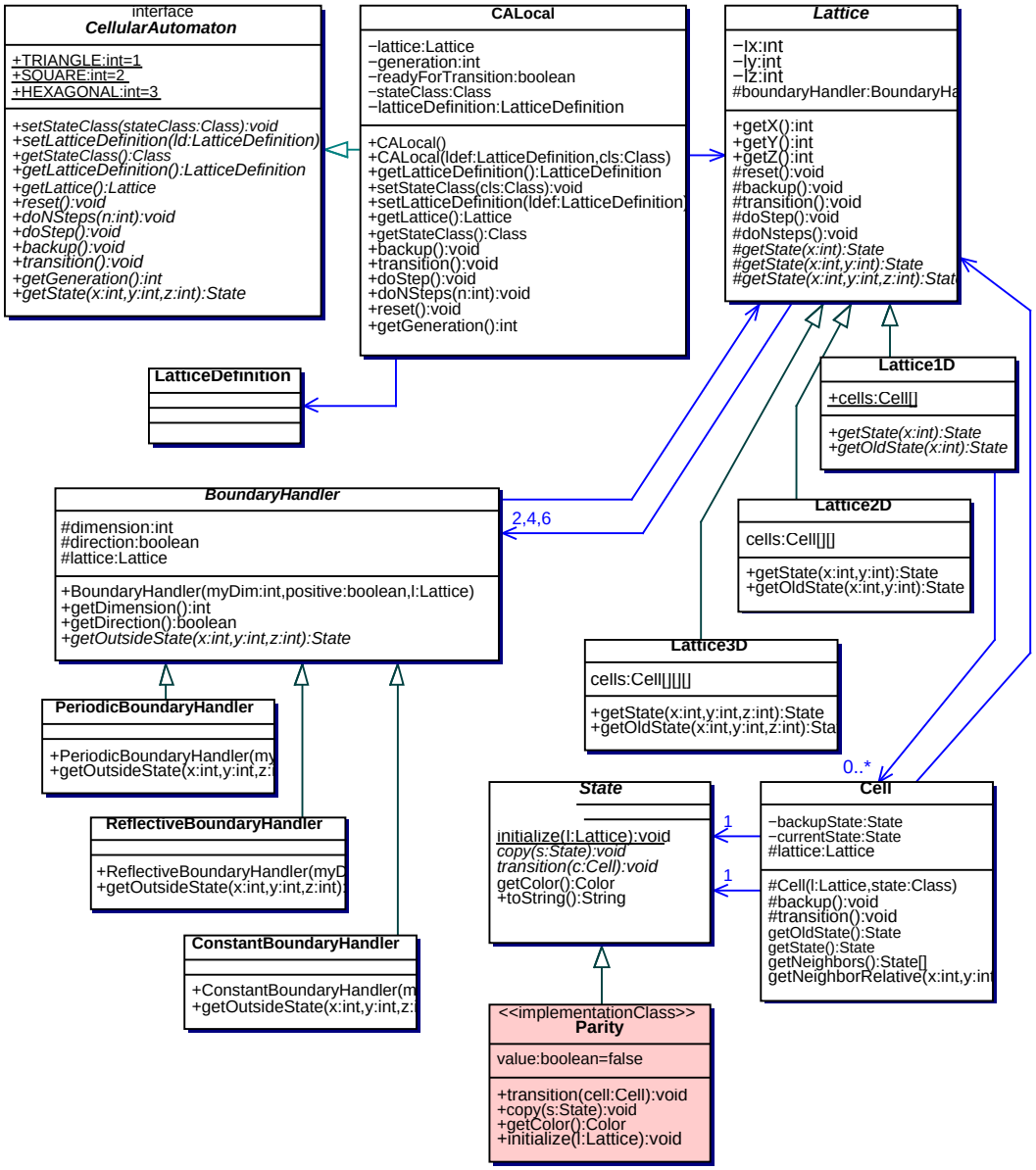

Fig. 1. Basic class structure of the JCASim package. The user-defined class (here `Parity`) can be found at the bottom.

of the `Cell`, `getNeighbors()`. The user only needs to implement four methods of `State`, namely

- `transition(Cell c)` to change the cell state depending on the current state and the state of the neighbors (accessible though `c.getNeighbors()`);
- `copy(State s)` to copy all non-constant member variables from the given Object to the current object. This operation is similar to `clone()`, but avoids the creation of a new object,which incurs a significant performance penalty.
- `getColor()` to return a color value to be used for displaying the cell in the simulation system. This color should depend on the current state of the

cell. Another option is to implement `getIcon()` and return an image to be shown for the given cell. The image should also depend on the cell state (`getColor()` should return `null` in the case when an icon is to be used). Icons- and color-representation can be mixed.

- `static initialize(Lattice l)` is a static method used to assign initial values to the different cells in a simulation. This method gets access to the `Lattice` in order to be able to set the initial values of cells depending on the absolute positions of the cells. This method also violates somewhat the encapsulation spirit, but avoids the necessity for the user to write two classes for one simulation. It is also logical for the initialization to have direct access to the member variables of the `State` objects, which is the case for this static method.

The acccess to the neighbors is managed through appropriate methods of `Cell`. The neighborhood can be selected in the simulation system (in which case the rules should be able to cope with neighborhoods with varying number of neighbors) or specified during the initialization phase.

If a cell accesses a neighbor outside the simulation region, a special object of type `BoundaryHandler` is called to handle this request. There are three basic boundary handlers in JCASim: `PeriodicBoundaryHandler`, `ReflectiveBoundaryHandler`, and `ConstantBoundaryHandler`. They return the appropriate state of the lattice at the periodic or reflected position or a special constant state. The `ConstantBoundaryHandler` can be used to implement all kinds of special treatments at the border, like special CA rules for the border cells, by returning a special state that does not appear in the inside of a simulation. The CA rule (implemented in `transition`) can then operate differently depending on the number and position of these boundary cells in the neighborhood.

The JCASim system includes a number of other classes for block-CA, for controlling the simulation, and for the graphical user-interface.

5 Examples

5.1 Parity CA: Java, 1-D

A simple CA with two states and a transition rule that calculates the parity of all neighbors can be specified in Java in a just few lines, as shown in Figure 2.

The resulting CA can be simulated in all dimensions, with any lattice, and any neighborhood as selected in the simulation system. This is due

```
import de.tubs.cs.sc.casim.*;
public class Parity extends State{
    /** The state consists of this boolean variable */
    boolean value = false;
    /** This method is called to update the cell state */
    public void transition(Cell cell){
        State neighbors[] = cell.getNeighbors();
        for (int i=0; i<neighbors.length; i++){
            value ^= ((Parity)neighbors[i]).value;
        }
    }
    /** Helper method for copying the cell state */
    public void copy(State s){
        value = ((Parity)s).value;
    }
    /** Color representation for this cell */
    public java.awt.Color getColor(){
      return value?java.awt.Color.black:java.awt.Color.white;
    }
    /** Initialization */
    public void initialize(Lattice l){
        for (int x=0; x<l.getX(); x++){
        for (int y=0; y<l.getY(); y++){
        for (int z=0; z<l.getZ(); z++){
            ((Parity)l.getState(x,y,z)).value =
                    Functions.prob(0.2F);
        }}}
    }
}
```

Fig. 2. Java code for the Parity automaton.

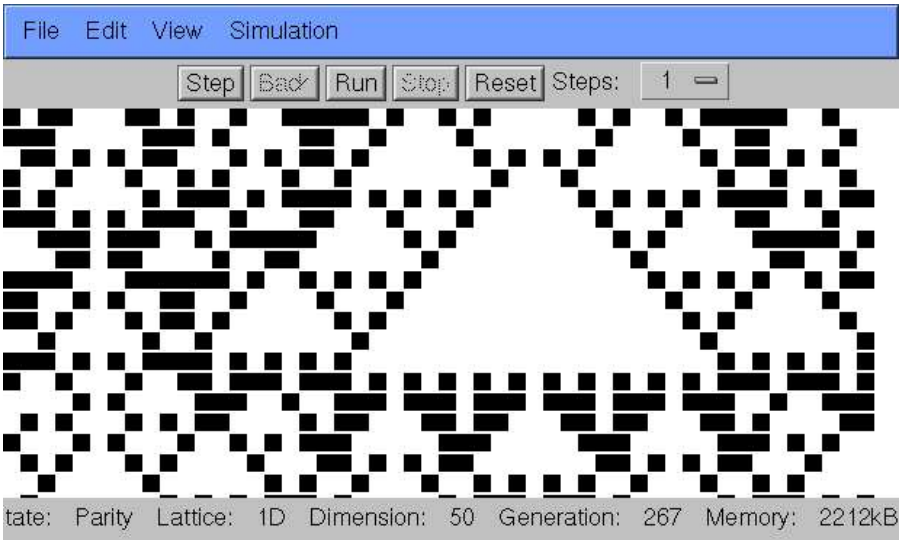

Fig. 3. 1-dimensional Parity automaton in the JCASim application.

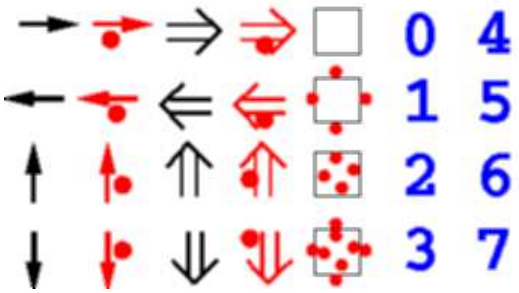

Fig. 4. Icons used for the 29 states of the von Neumann automaton (empty state is blank).

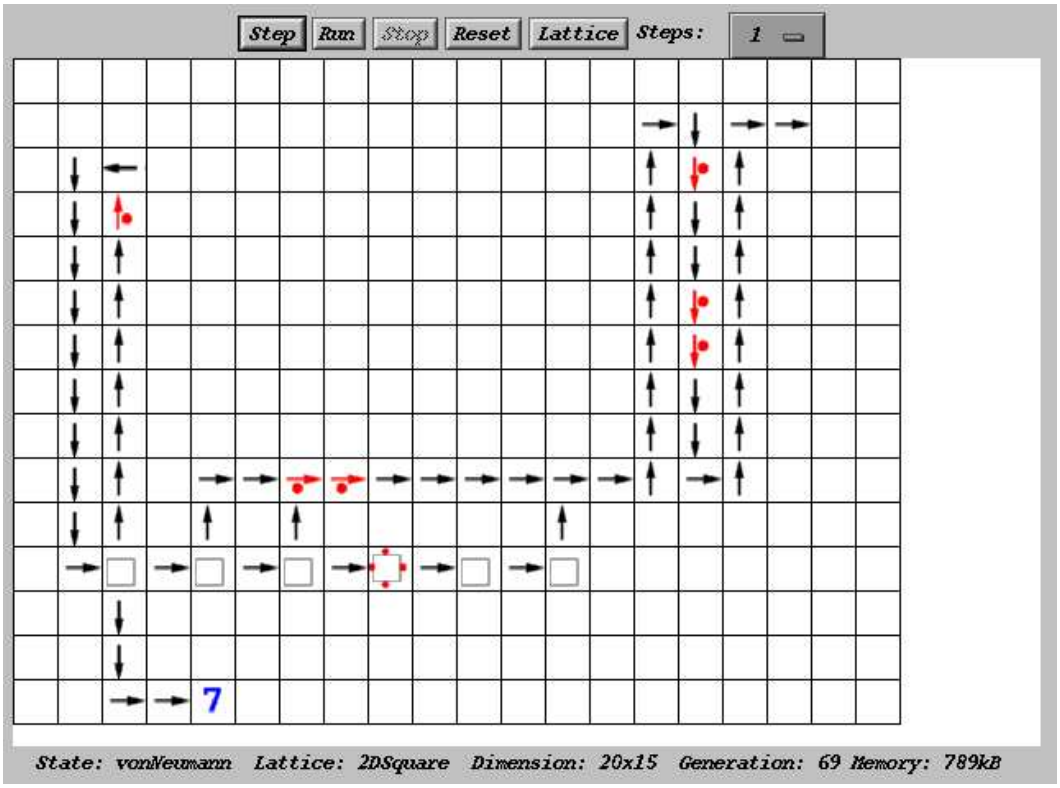

Fig. 5. Simulation of the von Neumann 29 state-CA (CDL code unchanged from [5]) as it appears in the applet in a web-browser.

to the use of `cell.getNeighbors()` and general references to the array of neighbors used in this code. An alternative is to use calls like `cell.getNeighborRelative(0,1)`, which access specific neighbors and are not portable to other dimensions and neighborhoods. A one-dimensional simulation is shown in Figure 3, where the older states are shown below the top line (time runs up).

The 29-state self-reproducing automaton developed by John von Neumann [13] is shown as a *CDL* example in [5]. Here we use that code directly, enhanced by the use of icons for the representation of cells. The icons are collected in one image (see Figure 4) and an appropriate icon is selected for each state. A simulation with a coding machine and a pulser ring is shown in Figure 5.

5.3 Block-CA

Block cellular automata are a special kind of CA, in which the state transition function operates on a block of cells, changing all of them without reference to any cell outside of the block. The blocks do not overlap, and can be updated in parallel or in any order, since no interaction between blocks takes place. Interaction of cells across the lattice is achieved by selecting a different partition of the lattice into blocks for each time step. In JCASim, block CA are supported directly (by using `BlockState` instead of `State`). Here we show an example CA which simulates a catalytic surface reaction using blocks of two cells.

We model the catalytic oxidation of carbon monoxide on a platinum surface as $A + \frac{1}{2}B_2 \to 0$ with the sub-steps of adsorption of A-molecules on an empty site of the surface, adsorption and dissociation of B_2 molecules on two adjacent empty sites of the surface, reaction of adjacent A and B molecules on the surface to a product which desorbs immediately, and diffusion of A particles on the surface (more details in [14,15]).

In a block CA, these sub-steps can be translated directly into changes of the configuration of a block of two cells. Each cell can now have one of three states: A, B, or 0 (empty). The transformations are:

$$
\begin{aligned}
[0,.] &\to [A,.] && \text{with probability } p_A \\
[0,0] &\to [B,B] && \text{with probability } p_B \\
[A,B],[B,A] &\to [0,0] && \text{with probability } p_R \\
[A,0] &\leftrightarrow [0,A] && \text{with probability } p_D
\end{aligned}
$$

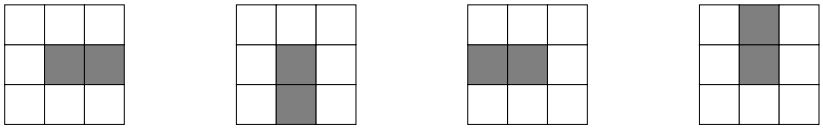

Fig. 6. Different orientation of blocks of two cells on a square lattice.

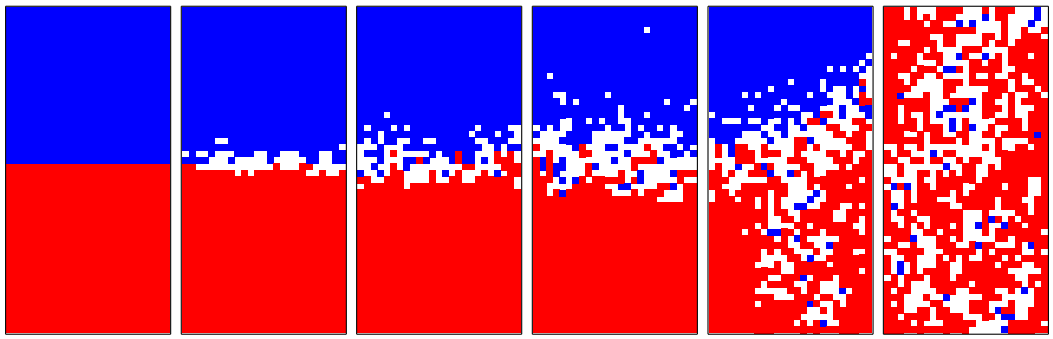

Fig. 7. Reactive front in the block CA (of size 25×50) at times 0, 10, 50, 200, 1000, 5000. The dark region shows A-particles, the grey region B-particles, and the white regions are empty sites.

Figure 7 shows such a simulation starting with two large areas filled with A and B particles. At the interface, reaction takes place, and the reactive front moves in both directions (but with different speeds).

5.4 Lattice dimension and geometries

As further examples we show the output of the "Print" command of the simulation system for two automata: the simple Greenberg Hastings excitable automaton can be simulated in any dimension and with any lattice geometry. Hexagonal and triangular examples are shown in Figure 8 on the left. A reaction-diffusion system using the Schlögl reaction is also shown in Figure 8 as a three-dimensional example. Further information on these examples can be found in [9] and in [15] (for reaction-diffusion systems).

6 Speed

6.1 Programming speed

The efficiency of a simulation ssystem depends not only on the execution speed of the completed simulation, but more significantly on the ease of programming the models to be simulated, and on the turnaround time for editing, compiling, starting a simulation, and observing the results. As the examples above have shown, programming a model for JCASim is fairly easy. The completed program can be compiled by the simulation environment and simulated immediately. Thus the turnaround time is very short, even if the simulation is less efficient than it could be for a simulation system written in C.

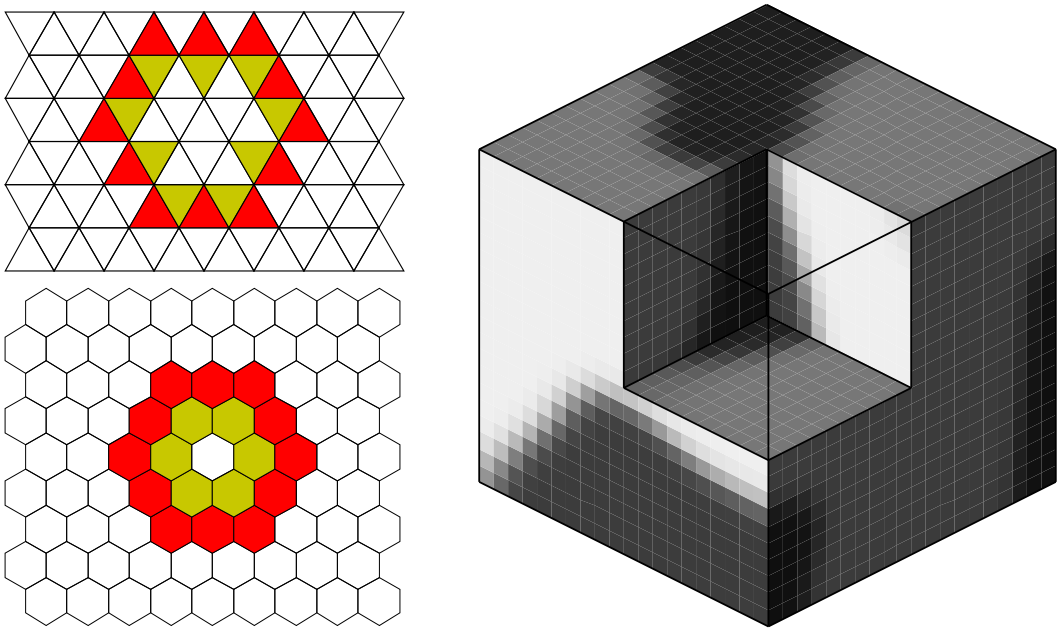

Fig. 8. Hexagonal and triangular Greenberg-Hastings automaton (left) and 3-dimensional reaction-diffusion system. Only the simulation area is shown as the output of the "Print" command of the simulation system.

6.2 Execution speed

A common objection to the use of Java for simulation is the poor performance of Java code. The performance of the JCASim system actually varies widely. A number of measures improve the performance drastically to make it quite acceptable:

- **New:** The creation of new objects takes several microseconds in current Java systems. This means that all object creation should be avoided in the inner loop, especially in the methods `transition`, `copy`, and `getColor`. The simulation system itself does not create any objects in the inner simulation loop, but reuses existing objects.
- **Neighbors:** The access to the neighbors of a cell should be done through the method `getNeighbors()`. In this case the vector of neighbors can be calculated once at the first time step, and stored and reused thereafter. This improvement saves approximately 9 microseconds for each neighborhood access.
- **Random:** The builtin random number generator `Math.random()` is very slow and of low quality. A better and much faster random number generator can be found in [16], which saves about 7 microseconds for each random number generated.
- **JIT:** Using a just-in-time compiler improves the simulation speed by a factor 3 to 10, where the higher number applies when all the other optimizations have been applied. Therefore it is important to make sure the Java virtual

machine uses a JIT compiler, which all recent versions do.

For a complex CA with several variables, while neglecting all of these points leads to a measured performance of 2500 cell updates per second (400 microseconds per cell update), using all improvements leads to a performance of 70,000 cell updates per second (14 microseconds per cell update). The automatic translation from CDL takes into account all of these recommendations. On a modern PC the simulation performance of the JCASim system varies between 50,000 and 500,000 cell updates per second. This is fast enough for most applications. The slowest element in the simulation is the display (between 500 and 50,000 cells displayed per second), which means that often the perceived speed is the same whether the state is displayed at every time step, or only every 10 or 50 time steps. Here the operating system and virtual machine implementation have strong influence on the performance.

7 Conclusion and Availability

We have described a new system for simulating cellular automata in Java, JCASIM. The cellular automata can be specified in Java or *CDL* and the system supports many different lattice geometries (1-D, 2-D square, hexagonal, triangular, 3-D), neighborhoods, boundary conditions, and can display the cells using colors, text, or icons. We have shown several examples to demonstrate the wide applicability of the simulation system.

The JCASim system is available for free download or for testing as embedded applets at `http://www.jcasim.de/`.

References

[1] T. Worsch, Programming environments for cellular automata, in: S. Bandini, G. Mauri (Eds.), Cellular Automata for Research and Industry (ACRI 96), Springer-Verlag, London, 1996, pp. 3–12.

[2] J. D. Eckart, A cellular automata simulation system, http://www.cs.runet.edu/~dana/ca/cellular.html, Radford University (1995).

[3] C. Hochberger, R. Hoffmann, CDL - a language for cellular processing, in: G. R. Sechi (Ed.), Proc. Second Int. Conf. on Massively Parallel Computing Systems, IEEE, 1996, pp. 41–64.

[4] R. Hoffmann, K.-P. Völkmann, CEPRA-8: A cellular processing machine, in: M. M. Furnari (Ed.), Massiv Parallelism Hardware, Software and Applications, World Scientific, Capri, 1994, pp. 379–391.

[5] C. Hochberger, CDL – Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen, Dissertation, TU Darmstadt (1998).

[6] U. Freiwald, Eine Java - Simulationsumgebung für Zellularautomaten, Diplomarbeit, Inst. of Scientific Computing, Techn. University Braunschweig (1999).

[7] T. Toffoli, N. Margolus, Invertible cellular automata: a review, Physica D 45 (1990) 229–253.

[8] T. Toffoli, T. Bach, A common language for "programmable matter" (cellular automata and all that), Bulletin of the, Italian Association for Artificial Intelligence (2001).

[9] J. R. Weimar, Simulation with Cellular Automata, Logos-Verlag, Berlin, 1998.

[10] M. Briesen, Eine verteilte Simulationsumgebung für gekoppelte Zellularautomaten in Java, Diplomarbeit, Inst. of Scientific Computing, Techn. University Braunschweig (1999).

[11] M. Briesen, J. R. Weimar, Distributed simulation environment for coupled cellular automata in Java, Parco 2001 (2001).

[12] B. Schönfisch, A. de Roos, Synchronous and asynchronous updating in cellular automata, BioSystems 51 (1999) 123–143.

[13] J. von Neumann, Theory of Self-Reproducing Automata, University of Illinois Press, 1966, (edited and completed by A. Burks).

[14] J. R. Weimar, Coupling microscopic and macroscopic cellular automata, Parallel Computing 25 (5) (2001) 601–611.

[15] J. R. Weimar, Simulating reaction-diffusion cellular automata with JCASim, in: T. Sonar (Ed.), Discrete Modelling and Discrete Algorithms in Continuum Mechanics, Logos-Verlag, Berlin, 2001, pp. 217–226.

[16] P. Houle, Rngpack 1.0, http://www.msc.cornell.edu/~houle/rngpack.