Theoretical and Practical Issues on Cellular Automata (ACRI 2000), S. Bandini and T.Worsch (eds), Springer Verlag, London, 2001, 47-54

JCASim – a Java system for simulating cellular automata

Uwe Freiwald and Jörg R. Weimar

Institute of Scientific Computing, Technical University Braunschweig,
D-38092 Braunschweig, Germany
J.Weimar@tu-bs.de

Abstract. The program system JCASim is a general-purpose system for simulating cellular automata in Java. It includes a stand-alone application and an applet for web presentations. The cellular automata can be specified in Java, in CDL, or using an interactive dialogue. 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 show several examples to demonstrate the wide applicability of the simulation system.

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 [9]. 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 the system *cellsim*, developed by J. Dana Eckart [1] 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 [5]), 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) [4].

2 The JCASim approach

The features distinguishing the new system JCASim [2] 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 or *CDL* (in the future, translators for other languages, such as *cellang* will be constructed). 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. They are formally equivalent to regular CA, but are much more convenient to formulate in many cases, especially where conservation laws must be observed. 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 simple 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 2 and 3 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 Varieties of CA

A large number of different possibilities can be selected in using CAs for simulation. 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 (to be described elsewhere) 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.

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`.

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 Examples

4.1 Parity CA

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:

```
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);
        }}}
    }
}
```

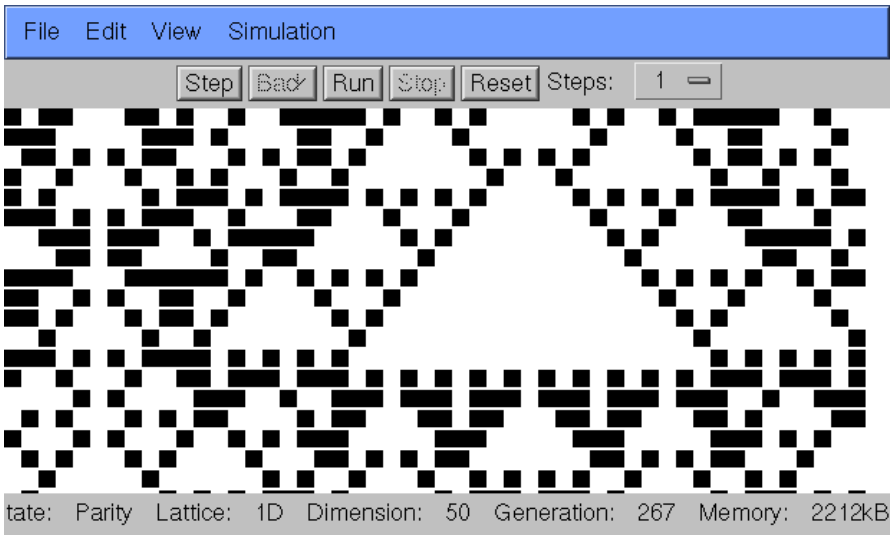

Fig. 1. 1-dimensional Parity automaton in the JCASim application.

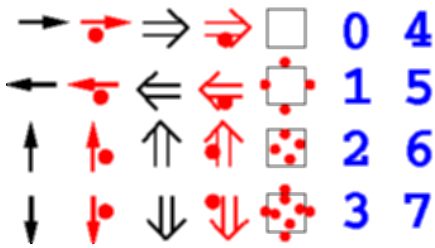

Fig. 2. Icons used for the 29 states of the von Neumann automaton (empty state is blank).

The resulting CA can be simulated in all dimensions, with any lattice, and any neighborhood as selected in the simulation system. This is due 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 1, where the older states are shown below the top line (time runs up).

4.2 von Neumann 29 state automaton

The 29-state self-reproducing automaton developed by John von Neumann [7] is shown as a *CDL* example in [4]. Here we use that code directly, enhanced

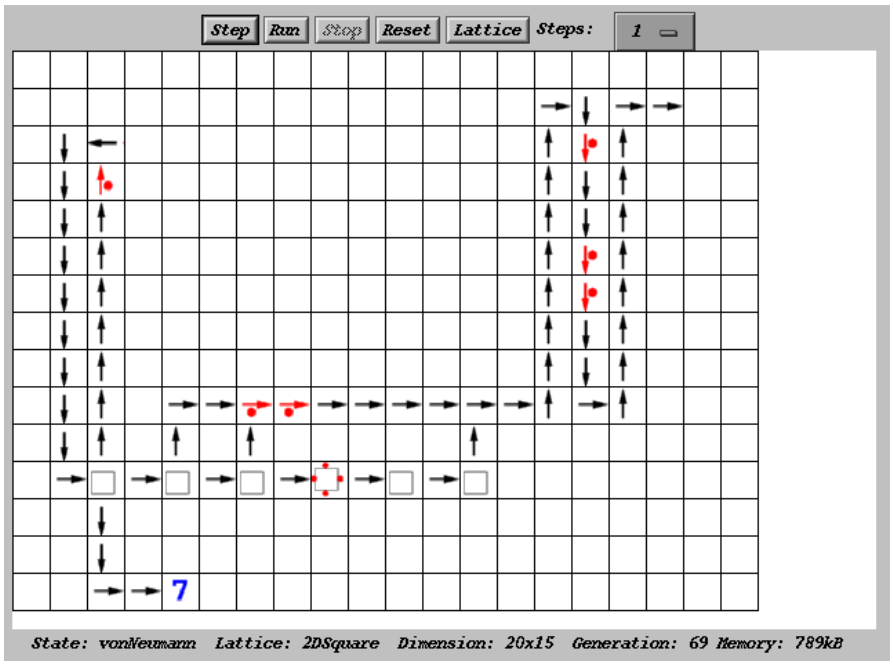

Fig. 3. Simulation of the von Neumann 29 state-CA (CDL code unchanged from [4]) as it appears in the applet in a web-browser.

by the use of icons for the representation of cells. The icons are collected in one image (see Figure 2) and an appropriate icon is selected for each state. A simulation with a coding machine and a pulser ring is shown in Figure 3.

4.3 Variety of 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 4 on the left. A reaction-diffusion system using the Schlögl reaction is also shown in Figure 4 as a three-dimensional example. Further information on these examples can be found in [8].

5 Performance

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.

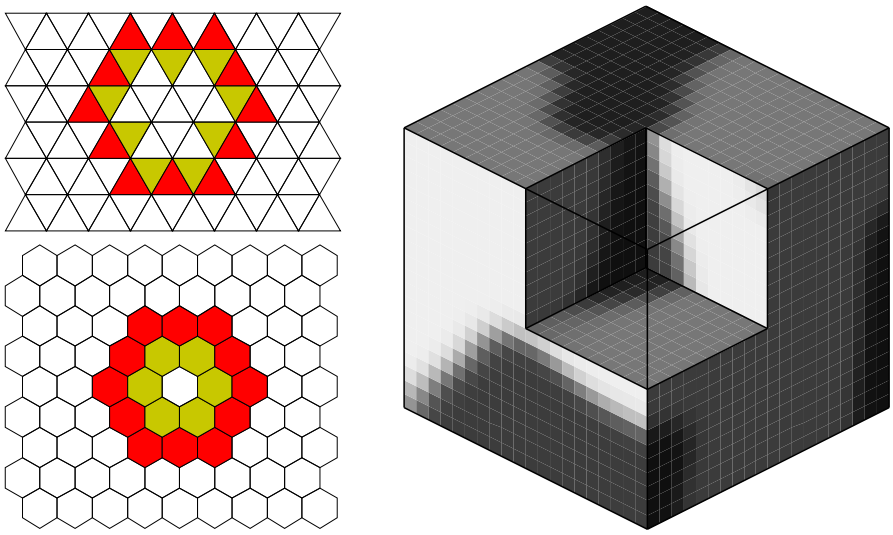

Fig. 4. Hexagonal and triangular Greenberg-Hastings automaton (left) and 3-dimensional reaction-diffusion system. Only the simulation area is shown as the output of the "Print" command of the simulation system.

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 [6], 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 optimiztions 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.

6 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.jweimar.de/jcasim/`.

References

1. J. Dana Eckart. A cellular automata simulation system. http://www.cs.runet.edu/~dana/ca/cellular.html, Radford University, 1995.
2. Uwe Freiwald. Eine Java - Simulationsumgebung für Zellularautomaten. Diplomarbeit, Inst. of Scientific Computing, Techn. University Braunschweig, 1999.
3. C. Hochberger and R. Hoffmann. CDL - a language for cellular processing. In Giacomo R. Sechi, editor, *Proc. Second Int. Conf. on Massively Parallel Computing Systems*, pages 41–64. IEEE, 1996.
4. Christian Hochberger. CDL – Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen . Dissertation, TU Darmstadt, 1998.
5. Rolf Hoffmann and Klaus-Peter Völkmann. CEPRA-8: A cellular processing machine. In M. Mango Furnari, editor, *Massiv Parallelism Hardware, Software and Applications*, pages 379–391, Capri, 1994. World Scientific.
6. Paul Houle. RngPack 1.0 http://www.msc.cornell.edu/~houle/rngpack
7. J. Von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966. (edited and completed by A. Burks).
8. Jörg R. Weimar. *Simulation with Cellular Automata.* Logos-Verlag, Berlin, 1998.
9. Thomas Worsch. Programming environments for cellular automata. In S. Bandini and G. Mauri, editors, *Cellular Automata for Research and Industry (ACRI 96)*, pages 3–12, London, 1996. Springer-Verlag.