Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

AN INTRODUCTION TO CELLULAR AUTOMATA MODELS WITH CELL-DEVS

Gabriel A. Wainer

Department of Systems and Computer
Engineering, Carleton University
1125 Colonel By Dr.
Ottawa, ON K1S 5B6, CANADA

ABSTRACT

In recent years, grid-shaped cellular models have gained popularity in this sense. Cellular Automata (CA) have been widely used with these purposes. The Cell-DEVS formalism is an extension to CA that can be used to build discrete-event cell spaces, improving their definition by making the timing specification more expressive. Different delay functions to specify the timing behavior of each cell, allowing the modeler to represent the timing complex behavior in a simple fashion. CD++ is a modeling and simulation tool that implements DEVS and Cell-DEVS formalisms. Here, we show the use of these techniques through a formal specification, and a variety of application examples, including basic models using CA, and varied extensions including triangular and hexagonal topologies.

1 INTRODUCTION

Partial differential equations and related formalisms have been the tool of choice for studying complex physical systems theoretically, and different approximations based on difference equations exist to solve this kind of problems. In recent years, new methods consider instead a spatial representation of the system of interest, using a cell space for studying the system of interest. In particular, the Cellular Automata formalism (Wolfram 1986) has been widely used to describe many complex systems with these characteristics. Cellular Automata (CA) are discrete-time discrete models described as cells organized as n-dimensional infinite lattices. CA evolve by executing a global transition function that updates the state of every cell in the space. Each cell has a discrete value that is modified by a local computation function. The composition of these functions, which only depends on the results of a local execution in each cell, generate a global behavior that can be studied at the scale field. This local function uses the present value for the cell and a finite set of neighbor cells to compute the new state.

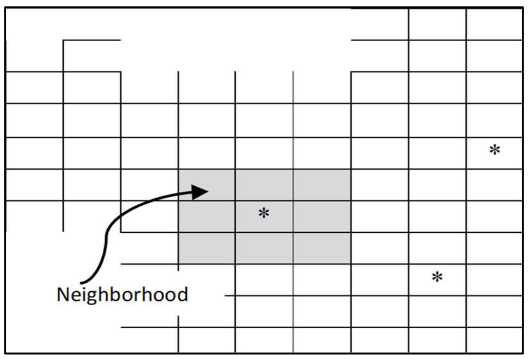

Figure 1: Sketch of a Cellular Automaton.

Conceptually, the local functions are computed synchronously and in parallel, using the state values of the present cell and its neighbors. The use of discrete time can reduce the precision of the model and reduce the performance of the simulation of the models when they become complex. To obtain higher precision, smaller time slots must be used, which results in more execution cycles that are sometimes unnecessary. Asynchronous CA can be used to avoid these problems.

In this tutorial, we introduce general concepts on Cellular Automata and provide a formal specification, and we discuss parallel execution of CA. We then show the definition of advanced CA in the CD++ toolkit (Wainer 2009): a simple CA (Brian's Brain), the Diamond-Square algorithm for generating terrain shapes, and we then discuss the application of triangular and hexagonal topologies. In this case, we discuss how to apply different topologies to excitable media CA, surface tension and lattice gas models.

2 FORMAL DEFINITION OF CELLULAR AUTOMATA

A conceptual cellular automaton can be formally defined as:

$$CCA = < S, n, C, N, \tau, T, c.\boldsymbol{Z}_0^+ >$$

Notation 1

$C_c \in S$ is the state of cell c, being $c \in \boldsymbol{Z}^n$, $c = (i_1,...,i_n)$ the cell's position into an n-dimensional cell space, where $\forall\ k \in [1,n]$, $i_k \in \boldsymbol{Z}$ is the position of the cell in the k-eth dimension. For conceptual CA, $\forall\ k \in [1,n]$, $i_k \in [-\infty,\infty]$.

Using this notation, each of elements of the tuple defined above can be defined as:

- **S** $\subseteq \boldsymbol{R} \wedge \#S < \infty$ is an alphabet used to represent the state for each cell;
- **n** $\in \boldsymbol{N}$ is the number of dimensions the cell space has;
- **C** $= \{ C_c / c \in \boldsymbol{R}^n \wedge C_c \in S \}$ is the state set for the cell space;
- **N** $= \{N_c / c \in \boldsymbol{Z}^n \}$ is the neighborhood set, with $N_c = \{ (v_{k1},...,v_{kn})_c / \forall\ (k \in \boldsymbol{N}, k \in [1, \eta_c]) \wedge (i \in \boldsymbol{N}, i \in [1, n]), v_{ki} \in \boldsymbol{Z} \}$. Here, $\eta_c \in \boldsymbol{N} \wedge \eta_c = \#N_c$;
- τ: $C_c \times N \times c.\boldsymbol{Z}_0^+ \rightarrow C_c$ is the local transition function, where $C_c[t+c] = \tau(C_{c+v1}[t],..., C_{c+v\eta c}[t])$, and for $t \in c.\boldsymbol{Z}_0^+ \wedge \forall\ (k \in \boldsymbol{N}, k \in [1, \eta_c])$, $vk \in N_c \wedge c + vk = (i_1+v_{k1}, ..., i_n+v_{kn})$;
- **T**: $C \times c.\boldsymbol{Z}_0^+ \rightarrow C$ is the global transition function; and
- $c.\boldsymbol{Z}_0^+ = \{ i / i \in \boldsymbol{N}, i = c.j \wedge j \in \boldsymbol{N} \} = \{ 0, c, 2c, 3c, ...\}$ $c.\boldsymbol{Z}_0^+$ is the discrete time base.

As we can see in the definition, the model is an **n**-dimensional cell space (**C**) that progresses in discrete time steps: the time base is defined by $c.\boldsymbol{Z}_0^+$ (a set of integer values separated by a time constant). The state for each cell in the space can take a value from a finite alphabet (**S**) whose elements can be Real numbers (although most CA use a limited number of Integer values). The cell's neighborhood is defined as a list of neighbors (η). The neighbors are defined as an n-tuple of positions relative to an "origin" cell, using an index (**k**) that allows to identify the neighbor number, and a second index (**i**) indicating the dimension for each of the neighbors' positions. The neighborhoods (which can be homogeneous or not) are defined as an array of neighborhood lists. For non-homogenous neighborhoods, each cell has a list composed of η_c elements, which consist of tuples of indexes relative to the origin cell. The cell space evolves by executing a global transition function (**T**) that changes the state of the cell space. The behavior of this function responds to the execution results of local transition functions (τ) that execute locally in the neighborhood for the cell (**N**). Conceptually, the computation for these local functions is done synchronously and in parallel for every cell in the space. The semantics of this behavior can be formally defined as:

$$\frac{C_c \in C \quad \forall\ c \in \mathbf{S} \subseteq \boldsymbol{R}^n,\ \#S < \infty; \qquad t \in c.\boldsymbol{Z}_0^+}{C[t+c] = T(C[t]), \quad \text{with } C_c[t+c] = \tau(N_c, C_c[t])\ \forall\ c \in \mathbf{S} \subseteq \boldsymbol{R}^n; \qquad t = t + c}$$

This definition considers that the global transition function analyzes all the cell space at the instant *t* (precondition in the rule above) and then it produces a change in the cell space for the next step (postcondition). The period for this step is of *c* time units. This change can be seen as the individual computation of the local transition function for each cell in the space.

As we can see, the conceptual CA has an infinite number of cells; nevertheless, in order to run in a computer, we need to modify the above definition. Formally, an *executable synchronous CA* is be defined as:

$$CA = < S, n, \{t_1,...,t_n\}, C, N, B, T, \tau, c.\boldsymbol{Z}_0^+ >$$

where all the elements in the tuple represent the same sets of the previous case, two sets have been added. In order to have a computable CA, we need to constrain its size, as follows:

- $\mathbf{S} \subseteq \boldsymbol{R} \wedge \#S < \infty$;
- $\mathbf{n} \in \boldsymbol{N}$; $n < \infty$;
- $\{\mathbf{t_1,...,t_n}\} \in \boldsymbol{N}$, with $t_k < \infty\ \forall\ k \in [1,n]$ is the number of cells in each of the dimensions;
- $\mathbf{C} = \{ C_c \,/\, c \in \mathbf{I} \wedge C_c \in S \}$, with $\mathbf{I} = \{ (i_1,...,i_n) \,/\, i_k \in \boldsymbol{N} \wedge i_k \in [1, t_k]\ \forall\ k \in [1, n] \}$ **(1)**
- $\mathbf{N} = \{N_c \,/\, c \in \boldsymbol{N}^n,\ c \in [1,t1]x...x[1,t_n]\}$, with $N_c = \{ (v_{k1},...,v_{kn})_c \,/\, \forall\ (k \in \boldsymbol{N}, k \in [1, \eta_c]) \wedge (i \in \boldsymbol{N}, i \in [1, n]), v_{ki} \in \boldsymbol{Z} \wedge t_i - | v_{ki} | \geq 0 \}$. Here, $\eta_c \in \boldsymbol{N} \wedge \eta_c = \#N_c$.
- **B** is the set of border cells, with $\mathbf{B} = \{\varnothing\}$ if the cell space is "wrapped" (that is, the cells in each border are connected with the cells in the opposite one), or $\mathbf{B} = \{C_b \,/\, C_b \in C$ with $b \in \mathbf{I} \}$, with **I** defined as in (1). In this case, B has the restriction that $\tau_b \neq \tau_c = \tau\ \ \forall\ (C_c \notin B) \wedge (C_b \in B)$.
- τ: $C_c \times N_c \times c.Z_0^+ \rightarrow C_c$; where $C_c[t+c] = \tau(C_{c+v1}[t],...,C_{c+v\eta}[t])$, **(2)**

with $t \in c.\boldsymbol{Z}_0^+ \wedge \forall\ (k \in \boldsymbol{N}, k \in [1, \eta_c]), vk \in N_c \wedge c + vk = (i_1+v_{k1} \bmod(t_1),..., i_n+v_{kn} \bmod(t_n))$ in the case that $B = \{\varnothing\}$, and $C_b[t+c] = \tau_b(C_b[t]),\ \forall\ b \in B$, with $t \in c.\boldsymbol{Z}_0^+$. In this case, $\tau_b \neq \tau_c = \tau\ \forall\ c \notin B$, and for these ones, C_c is computed like in (2). If the cell space is homogeneous, then $\eta_c = \eta \wedge N_c = N$.

- **T**: $C \times c.\boldsymbol{Z}_0^+ \rightarrow C$.
- $c.\boldsymbol{Z}_0^+ = \{ i \,/\, i \in \boldsymbol{N}, i = c.j \wedge j \in \boldsymbol{N} \} = \{ 0, c, 2c, 3c, ...\}$ $c.\boldsymbol{Z}_0^+$ is the discrete time base.

This definition for executable cellular automata differs in certain aspects of that of conceptual ones. The first difference is that the cell space is bounded in each of the dimensions ($\mathbf{t_1,...,t_n}$). The number of dimensions is also finite, and the cell's indexes are bounded to finite natural numbers. Another constraint is due to the loss of homogeneity in the cell space. This is due to the existence of a finite number of cells. Therefore, it is necessary to include a set of border cells (**B**) with different behavior than the others in the cell space. All the cells in the border have different behavior that those in the rest of the automaton. When $B \neq \{\varnothing\}$, it is used to be defined as: $B = \{C_b \,/\, C_b \in C$ with $b \in L\}$, being $L = \{ (i_1,...,i_n) \,/\, i_j = 0 \vee i_j = t_j\ \forall\ j \in [1, n] \}$, and with $\tau_b \neq \tau_c = \tau\ \forall\ c \notin L$. When wrapped models are considered, $B = \{\varnothing\}$ and the rules defined in (2) should be used.

The semantics for the transition function T is that the local transition functions in the automata are executed simultaneously in parallel, and is defined by:

$$\frac{C_c \in C \;\; \forall\, c = \{ (i_1,\dots, i_n) \,/\, i_j \in [1, t_j] \; \forall\, j \in [1,n] \}, \qquad t \in c.\boldsymbol{Z}_0^{+}}{C[t{+}c] = T(C[t]),\text{ with } C_c[t{+}c] = \tau(N_c,C_c[t]) \quad \forall\, c = \{ (i_1,\dots, i_n) \,/\, i_j \in \boldsymbol{N},\, i_j \in [1, t_j] \; \forall\, j \in \boldsymbol{N},\, j \in [1,n] \};\;\; t = t + c}$$

The meaning is similar that in the case of conceptual CA but adding boundaries to the cell space.

Following, we will show a simple example of a CA called "Brian's Brain" (Wilensky 2002). This is an extension of the Seeds pattern model (Silverman 1996) which resembles the Game of Life, but in this case the model's cells can be in any of three states: *firing*, *refractory*, and *off,* making an analogy with the brain. The model uses Moore's neighborhood (i.e. the neighbor cells are the eight adjacent cells). The CA follows three simple rules. At each time step, (1) an *off* cell turns to *firing* if it has exactly two firing neighbors; (2) a *firing* cell always evolves to *refractory*; and (3) a *refractory* cell always evolves to *off*. Following these simple rules, the CA shows an emergent behavior. The refractory cells tend to lead to a pattern that reappears after a certain amount of generations in the same orientation but in a different position. Although some Brian's Brain patterns explode messily and chaotically, but most of them will often contain diagonal waves of firing and refractory cells. These patterns cannot be predicted in advance. Following, we show the definition of this simple CA as a Cell-DEVS model (Ruiz-Martin and Wainer 2016).

$$Brian's\ Brain\ AM = < X, Y, \mathrm{I}, S, \theta, \mathrm{E}, \mathrm{delay}, \mathrm{d}, \delta_{ext}, \delta_{int}, \tau, \lambda, D >$$

where:
$X = Y = S = \{0,1,2\}$ // 0 = off; 1 = firing; 2 = refractory
$\theta = \{(\mathrm{s}, \mathrm{phase}, \sigma\mathrm{queue}, \sigma) \mid \mathrm{s} \epsilon S$ is the status value for the cell, phase ϵ {active, passive},
$\quad \sigma\mathrm{queue} = \{((s_1,\sigma_1), \dots, (s_m,\sigma_m)) | (m \,\epsilon N \;\wedge m < \infty) \wedge \forall (i \,\epsilon N, i \,\epsilon [1, m]), s_i \epsilon S \;\wedge\; \sigma_i \,\epsilon\, R_0^+ \cup \infty\}$ and $\sigma \,\epsilon\, R_0^+ \cup \infty\}$
E = {(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)}
delay: transport. d: 100ms
τ: is defined by the following rules:

(0,0)=0 **if** (0,0)=2
(0,0)=1 **if** ((0,0)=0) & ((number of neighbors with s=1)=2)
(0,0)=2 **if** (0,0)=1

δint, δext, λ and D are defined using Cell-DEVS specifications.

Once the behavior of each cell is defined, the cell space is built as a Cell-DEVS coupled model:

$$Brian'sBrain\ CM = < Xlist, Ylist, \mathrm{I}, \mathrm{X}, \mathrm{Y}, \mathrm{n}, \{t_1, \dots, t_n\}, \mathrm{N}, \mathrm{C}, \mathrm{B}, \mathrm{Z}, \mathrm{select} >$$

where:
X=Y=*Xlist*=*Ylist*=∅
n=2 and $\{t_1,\dots,t_n\}=\{20,20\}$
N= select={(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)}
C = *Brian's Brain AM*
B=∅ //wrapped

$$Z=\{ \begin{array}{llll}
P_{i.i}^{Y1} \to P_{i.i-1}^{X1} & P_{i.i+1}^{Y1} \to P_{i.i}^{X1} & P_{i.i}^{Y2} \to P_{i+1.i}^{X2} & P_{i-1.i}^{Y2} \to P_{i.i}^{X2} \\
P_{i.i}^{Y3} \to P_{i.i+1}^{X3} & P_{i.i-1}^{Y3} \to P_{i.i}^{X3} & P_{i.i}^{Y4} \to P_{i-1.i}^{X4} & P_{i+1.i}^{Y4} \to P_{i.i}^{X4} \\
P_{i.i}^{Y5} \to P_{i.i}^{X5} & P_{i.i}^{Y5} \to P_{i.i}^{X5} & P_{i.i}^{Y6} \to P_{i-1.i-1}^{X6} & P_{i-1.i-1}^{Y6} \to P_{i.i}^{X6} \\
P_{i.i}^{Y7} \to P_{i-1.i+1}^{X7} & P_{i-1.i+1}^{Y7} \to P_{i.i}^{X7} & P_{i.i}^{Y8} \to P_{i+1.i-1}^{X8} & P_{i+1.i-1}^{Y8} \to P_{i.i}^{X8} \\
P_{i.i}^{Y9} \to P_{i+1.i+1}^{X9} & P_{i+1.i+1}^{Y9} \to P_{i.i}^{X9} & \}
\end{array}$$

This CA can be easily implemented in CD++. To do so, we need to map the information from the formal model definition to define the possible cells states, the neighborhood set, the rules, the size of the cell space, the type and value of the delay, the type of borders and the initial value of the cells. All these parameters are defined as shown as in the code below:

```
[briansbrain]
type: cell             width: 400          height: 400
delay: transport border: wrapped
neighbors: (-1,-1),(-1,-0),(-1,1) (0,-1),(0,-0),(0,1) (1,-1),(1,-0),(1,1)
localtransition: briansbrain-rule

[briansbrain-rule]
rule : 0 100 { (1,0)=2 }
rule : 1 100 { (0,0)=1 and trueCount = 2 }
rule : 2 100 { (0,0)=1 }
rule: 0 100 {t}
```

Figure 2: Brian's Brain CD++ model definition for a 20x20 grid.

Figure 2 shows the implementation in CD++ of Brain's Brain model in a 400x400 grid. First, we define the name of the Cell-DEVS component. Then, we define the type of model (a cell space) and its size (as it is a 2D model, we only need to define its width and height). The model uses a transport delay, and wrapped borders. After the keyword "neighbors", we define the neighborhood set (in this case, a Moore's neighborhood). Finally, the local transition function is defined as a set of rules. They are implemented using CD++ specifications, which have the form:

rule: POSTCONDITION DELAY { PRECONDITION }

These indicate that when the PRECONDITION is satisfied, the state of the cell will change to the designated POSTCONDITION, whose computed value will be transmitted to other components after consuming the DELAY. If the precondition is false, the next rule in the list is evaluated until a rule is satisfied or there are no more rules.

Figure 3 shows a snapshot of the Brian's Brain CA simulation in CD++ with a random initial configuration (Ruiz-Martin and Wainer 2016). Off cells are represented in black, firing cells are represented in white and refractory or refractory cells in blue. In the figure, we can easily appreciate diagonal waves of firing and refractory cells that appear as emergent behavior. Firing and refractory cells expand and move around the whole grid after few steps. A full simulation is available in https://www.youtube.com/watch?v=8MCDuRVEbeg, where we can see how diagonal waves that appear as emergent behavior move and change over time.

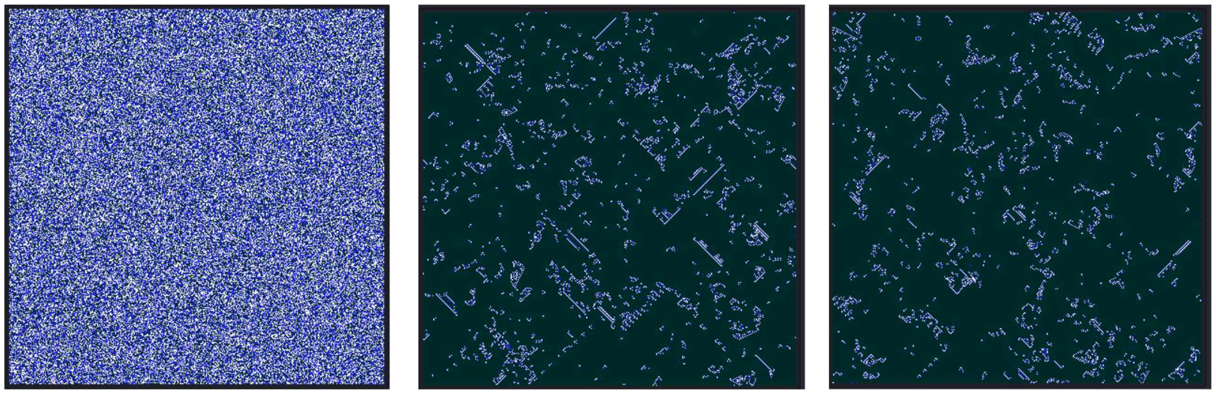

Figure 3: Snapshot of Brian's Brian rules CA implemented in CD++.

CA can be executed in parallel through more or less straightforward implementations. Their topology and local computation rules make it ideal for multiprocessor execution and we have implemented various parallel versions of CA in different platforms, ranging from the Cell BE (running the local rules in SIMD mode) up to CUDA libraries (Ribacki and Himmelspach 2009). We used Conway's life game (Gardner 1970) with different initial configurations, and the evidence showed that the calculation time on the given problem domain has reduced by applying the parallel solving method the cellular automata model. Moreover, another type of CA model such as the modified von Neumann CA model has been used to validate the stability of the parallel approach as well. Figure 4 shows a comparison between sequential and parallel execution of the Life CA on the Cell BE processor.

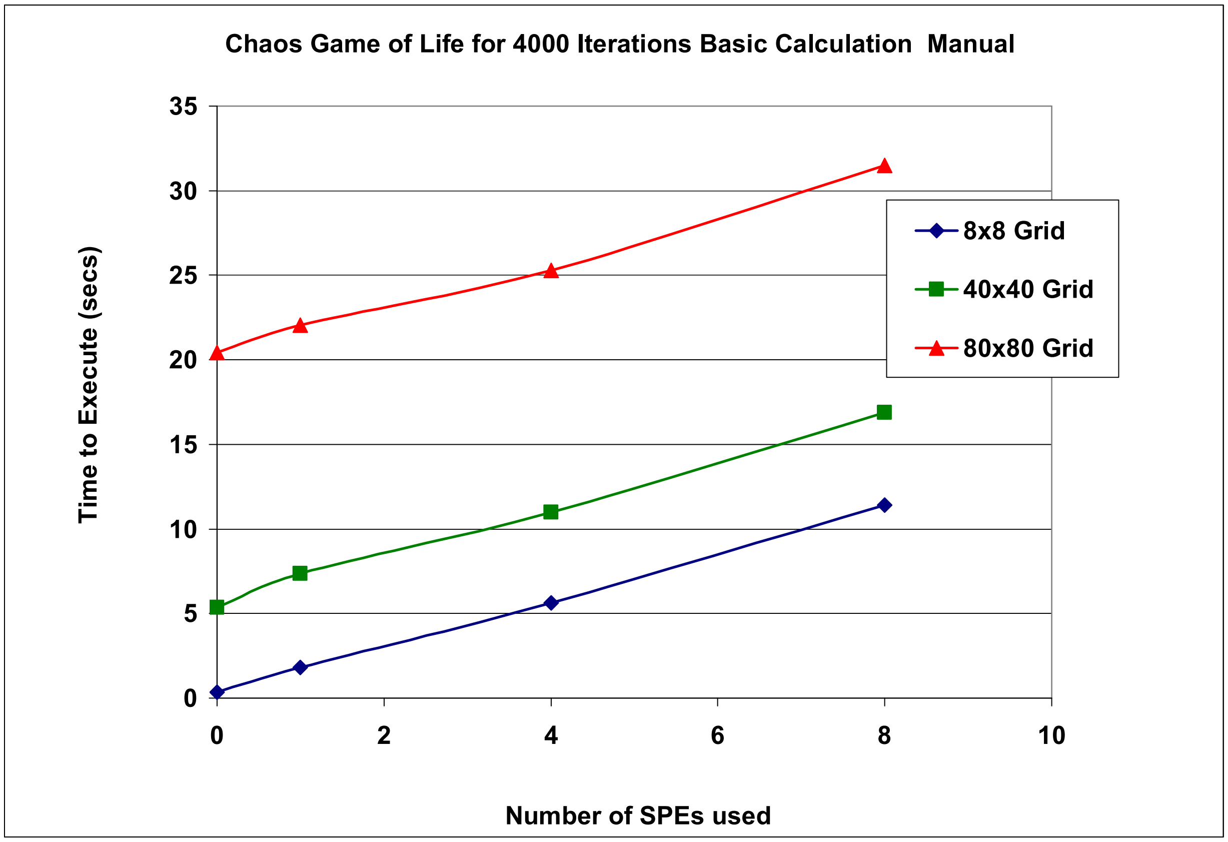

Figure 4: Performance comparison for the Game of Life on multiple SPUs and different grid sizes.

The Cell BE includes a main processor, and 8 SIMD units, called SPUs. These are Reduced Instruction Set Computing processors with 128-bit Single Instruction Multiple Data (SIMD) organization that allows it to execute single and double precision instructions. The method involves performing many calculations simultaneously. The transfer of data in a Cell BE is done through Direct Memory Access (DMA) transfers. The implementation uses an array of vectors which are simple to parallelize on the SPUs through DMA. As we can see in the figure, there are performance gains, but they are restricted because there is a need to spend time is being spent on executing PPU code (the main core of the Cell BE unit), which requires transferring data to buffer from the CA data structure (a matrix), and putting values back values from the buffer into the simulation space.

CUDA (Compute Unified Device Architecture), is a parallel computing platform and programming model that implemented by NVDIA company by providing the access to a virtual instruction set and the usage of the memory in graphic processing units (GPUs). By using this programming language, a series of independent data could be calculated through various threads concurrently (Che et al. 2008). This boosts up the throughput of a general program in order to maximize the efficiency of the running program. The host issues a succession of kernel invocation to the device. Each kernel is executed as a grid of thread blocks. We organized the space in 3D array of threads using a specific *id* for the thread that has been designated to calculate the state counts for a typical cell itself. In addition, same calculation methods have been implemented in the kernel function including the von Neumann neighborhood calculation. The steps are as follows:

- Initialize two 1D arrays in the host memory (i.e. the system memory).
- Pass arrays to the GPU device global memory.
- Distribute the block dimension and size for running the core function.
- Call the global function to process the cell calculation in each thread.
- Synchronized all the threads and record the time consuming for this iteration.
- Copy out the data after each iteration to the host memory and continue running the program in the next time step until receive the interrupt signal from the user.

In order to evaluate the reliability in large elements surroundings, the tests have included the scale of N^2 which is upmost to 100 million elements in single processing and two-processor processing. The initial values are at random. Figure 5 shows the execution time in the corresponding scales. We can see how the execution time executing the CA in CUDA is lesser than running in single processor or two-processor, which is similar to the expectation.

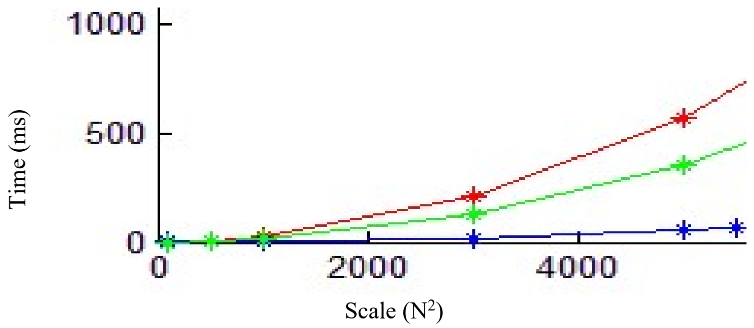

Figure 5: Execution time in single processor (red), 2 processors (green) and multi-processors (blue).

3 MEAN FILTER FOR IMAGE PROCESSING

CA have been used image processing because pixels can be treated as cells and because of the parallelism capabilities of CA discussed above, which improves computing efficiency. Several image processing techniques used for solving a variety of problems, such as edge detection, noise reduction, have been defined using CA. In this section we show a simple mean filter implemented as a noise reduction technique.

```
[meanFilter]
type : cell         dim : (100, 100)          delay : transport
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)

localtransition : filter-rule

[filter-rule]
rule : {((-1,-1)+(-1,0)+(0,0)+(-1,1)+(0,-1)+(0,1)+(1,-1)+(1,0)+(1,1))
                / 9} {100} {t}
```

Figure 6: Mean Filter implementation in CD++

A mean filter is a linear filter that preserves the borders and reduces the noise while smoothing the image. It replaces each pixel in the image by the average of all neighbors (using Moore's neighborhood). The mean filter could be used as the initial stage for edge detection since it smoothes the image reducing the chance of getting wrong edges. Figure 6 shows the model implemented in CD++.

Figure 7 shows the results obtained from extracting the information of various images and printing the value of each pixel as a grayscale image. Grayscale images carry only intensity information which means that the RGB values are all the same. All the images are pictures with salt and pepper noise. As we can see, the mean filter reduces the amount of noise on the image, however it blurs the image therefore reducing its quality, which is a drawback of this technique. Other kinds of filters present some parameters that assist on controlling the blurring problem.

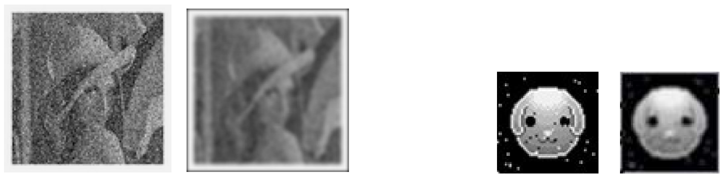
Figure 7: Left: image with salt-and-pepper noise; right: smooth image.

4 THE DIAMOND – SQUARE MODEL

In this section, we show how to define a traditional CA to implement the Diamond–Square Algorithm, which is used for generating height maps. To show the model's behavior, let us give an example using a 5x5 array. In figure 8.a) we can see a four corner "seed" values highlighted in black:

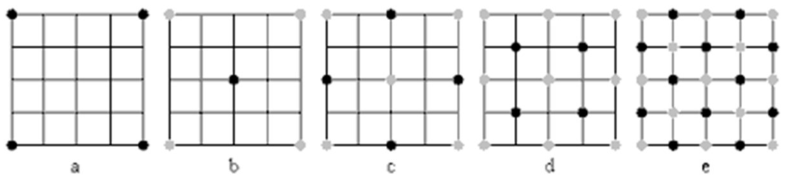

Figure 8: The Diamond-Square algorithm.

After this starting point, an iterative subdivision routine, is conducted in two steps:

- The diamond step: taking a square of four points, we generate a random value at the square's midpoint, where the two diagonals meet. The midpoint value is calculated by averaging the four corner values, plus a random amount. This gives you diamonds when you have multiple squares arranged in a grid, as seen in Figure 8b).
- The square step: we consider each diamond of four points and generate a random value at the center of the diamond. We calculate the midpoint value by averaging the corner values, plus a random amount generated in the same range as used for the diamond step, to build squares, as seen in Figures 8.d) and 8.e).

The process is repeated iteratively. A partial version of the model can be found following:

We now show an implementation of this CA model represented as a Cell-DEVS model using CD++. We defined a 3D model that uses two layers: the bottom one represents the height values of the map, and second level represent which phase are we currently computing (even is for the diamond step, and odd is for the square step). The model is defined in the following code snippet. The model uses two layers (in this example, two layers of 17x17 cells), and we assign different rules to be executed on each of the layers (using the *zones* statement). The second layer is used for computing the current phase step and to clean unused values. At the beginning, all the values of the second layer are 0, and after 200 ms, they become

0.1 which represents that they must be cleaned (as seen in the last rule, commented as "clear"). We use the integer part of the state value to record the direction of the model. The *second-rule* switches the value of the cell and adds 1 or 0.1 on each step to decide if we are in the square or diamond step.

```
[DiamondSquare]
type: cell              dim: (17,17,2)
delay: transport        border: nowrapped
neighbors:(-1,-1,0)(-1,0,0)(-1,1,0)(0,-1,0)(0,0,0)(0,1,0)(1,-1,0)(1,0,0)(1,1,0)(0,0,1)
zone: DiamondSquare-rule { (0,0,0)..(16,16,0) }
zone: second-rule { (0,0,1)..(16,16,1) }

[DiamondSquare-rule]
rule: {trunc((0,0,0))+0.1} 200 {fractional((0,0,0)) = 0}
rule: {trunc((0,0,0))+ 1 } 100 {fractional((0,0,0)) = 0.1}

[second-rule]
rule: {trunc((0,0,0))+0.1} 200 {fractional((0,0,0)) = 0}
rule: {trunc((0,0,0))+ 1 } 100 {fractional((0,0,0)) = 0.1}


rule: {0} 0 {fractional((0,0,0)) > 0 and (fractional((0,0,1)) > 0)}  % Clear
```

The bottom layer contains the direction for transmission of the information in the next step, and this is stored in the fractional part of the value (0.1 – N; 0.2 – E; 0.3 – S… 0.8 – NW). The integer part of the value contains the height of the cell. We transmit the values in different directions, and then, in the diamond step, we send the value through its diagonal to the center of those points. This behavior is defined using the following rules:

```
% send value
rule : {trunc((-1,-1,0))+0.81} 10 { (0,0,0)=0 and (-1,-1,0) != 0 and
       (fractional((-1,-1,0)) = 0.8 or fractional((-1,-1,0)) = 0 ) and even((0,0,1))}
rule : {trunc((1, 1,0))+0.61} 10 { (0,0,0)=0 and (1,  1,0) != 0 and
       (fractional((1,1,0)) = 0.6 or fractional(( 1, 1,0)) = 0 ) and even((0,0,1))}
...
% keep direction
rule : {trunc((0,0,0))+0.8} 10 {fractional((0,0,0))=0.81 }
rule : {trunc((0,0,0))+0.7} 10 {fractional((0,0,0))=0.71 }
```

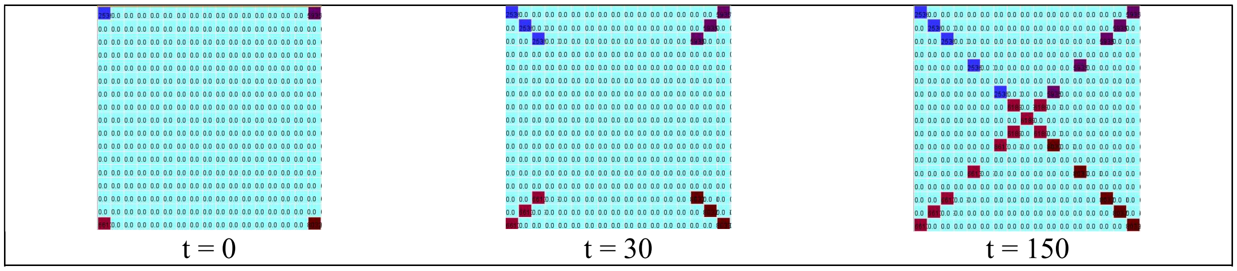

Figure 9: The diamond step: sending values through the diagonal.

In the next step, we should first send the values to its up, down, left and right direction, and get to the center of those points. When a cell has value 0, depending on the values of the four neighbors, it will start to calculate its own value, as sketched in the following rule.

```
rule : {trunc(((0,-1,0)+(0,1,0)+(-1,0,0)+ (1,0,0))/4 + uniform(-1,1) * 10000 *
          power(2,-1 *(0,0,1)))} 10  { (0,0,0)=0 and (0,-1,0) > 0  and (0,1,0) > 0 and
              (-1,0,0) > 0 and (1,0,0) > 0 and fractional((-1,0,0)) = 0.31}
```

Figure 9 shows the execution results of this step.

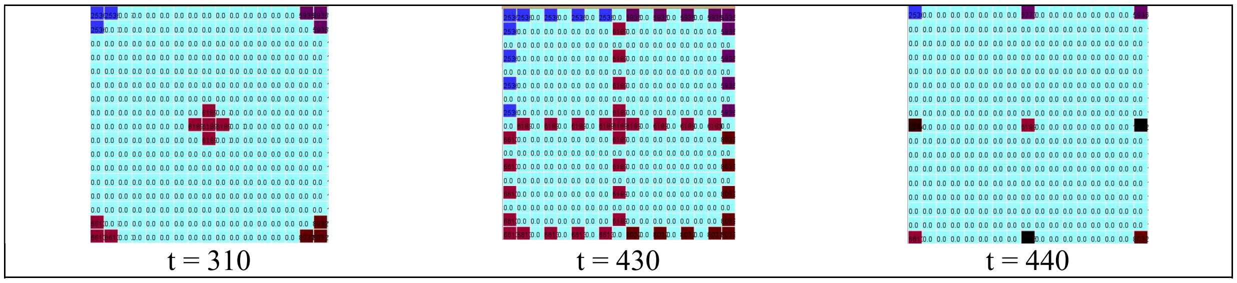

Figure 10: Computing the square rule.

Finally, when there are no places left to transmit values, the computation should finish, and we have obtained the map. The following rules are used:

```
%last cal
rule : {trunc(((-1,-1,0)+(1,1,0)+(-1,1,0)+(1,-1,0))/4 + uniform(-1,1) * 10000 *
         power(2,-1*(0,0,1)))} 10 { (0,0,0)=0 and (-1,-1,0) > 0 and (1,1,0) > 0 and
           (-1,1,0) > 0 and (1,-1,0) > 0 and   fractional((-1,-1,0)) = 0 and
             fractional(1, 1,0)=0 and fractional(-1, 1,0) = 0 and fractional(1,-1,0)=0}
rule : {trunc(((0,-1,0)+(0,1,0)+(-1,0,0)+ (1,0,0))/4 + uniform(-1,1) * 10000 *
         power(2,-1 *(0,0,1)))} 10   { (0,0,0)=0 and (0,-1,0)>0 and (0,1,0)>0 and
           (-1,0,0)>0 and (1,0,0)>0 and fractional((0,-1,0))=0 and
            fractional(0, 1,0)=0 and fractional((-1,0,0))=0 and fractional((1,0,0))=0 }
```

As we can see in Figure 10, at t=1520, we compute the diamond step, and then the square step. As there are no places left, the last computation is executed, and at t=1560 we obtain the final result.

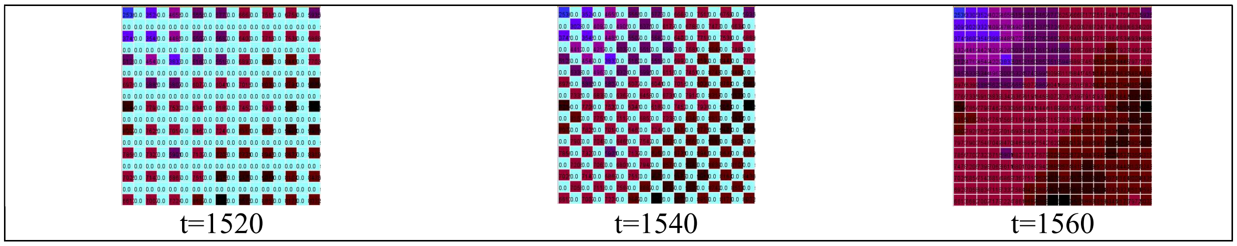

Figure 11: Final computation step.

5 TRIANGULAR AND HEXAGONAL TOPOLOGIES

Cellular models as the ones presented in earlier sections portray space a square geometry. There are numerous physical models that have attributes of expansion or diffusion in which a rectangular space is not adequate. Hexagonal topologies have shown better results in some of those cases, as it provides equal distances from all its neighbors, hence providing isotropic nature. Triangular topologies also are popular, but they are normally reserved for cases wherein a limited number of neighbors are required to observe a certain phenomenon. Triangular meshes permit covering zones with additional fluctuated topology, while allowing each cell to have a set number of nearby neighbors. Hexagonal meshes on the other hand have higher isotropy which is the ability to represent identical conduct in each direction. While limiting the neighbors gives us data abstraction wherein the number of messages communicated from the neighbors are lowered, but the representation and visualization of such models become difficult.

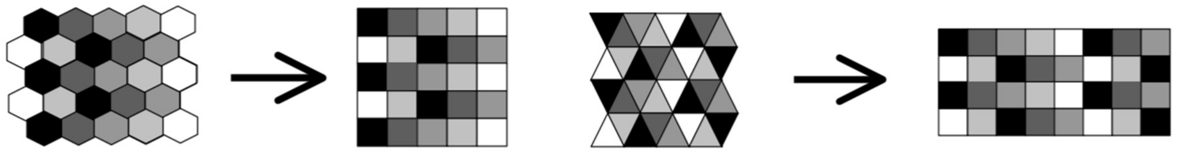

Figure 12. Hexagonal Mapping and Triangular Mapping.

In order to allow triangular and hexagonal topologies, we built a lattice translator that converts triangular and hexagonal topologies and translates them into square compatible rules using shift mapping in CD++. In CD++ a cell in a 2D space is referenced as a tuple (x, y) where x denotes the row and y denotes the columns. The hexagonal mapping, which has been defined in (Weimar 1997) is shown in Figure 11. It uses a function that shifts the alternate rows in opposite directions, at the same time preserving the boundary conditions. The triangular mapping, also shown in Figure 11, is done in an analogous way every other cell has different orientation and each row of the triangle is mapped to a row of square reliant on the parity x+y.

In the rest of this section we will discuss focus on the description of various CA and their Cell-DEVS implementation using triangular and hexagonal meshes and their translation and simulation.

5.1 Excitable Media

An excitable medium is a nonlinear dynamical framework which has the ability to engender a wave of some specification and which cannot allow passage to another wave until the point that a specific measure of time has passed (Demongeot et al. 1985). For instance, magnetic fields, the heart tissue, or forest fires can be considered as excitable media. This phenomenon has been modeled with CA, and for every excitable medium a cell has been usually modelled as having one of three states: resting, excited or recovering. For instance, in a forest fire the states can be unburnt, burnt or burning. The model, originally presented in (Ameghino and Wainer 2000), is represented as a cell space with square topology as follows:

```
[excitable]
type : cell      width : 11             height : 11      delay : transport
border : wrapped
neighbors : (-1,0) (0,-1) (0,0) (0,1) (1,0)
localtransition : excitable-rule

[excitable-rule]
rule : 0 100 {(0,0)=0 and statecount(2)=0 }
rule : 2 100 {(0,0)=0 and statecount(2)>0 }
rule : 1 100 { (0,0) = 2 }
rule : 0 100 { (0,0) = 1 }
rule : { (0,0) } 100 { t }
```

The first rule states that the cell value is 0 when there are cell neighbors with value 0, in other words if there are no excited cells nearby, in which case the cell will remain in the resting state. The second rule says that if the cell has value 0 and there are neighbors with value 2 (excited), then it becomes excited (value 2). The third and the fourth rule are used to change the cell to a transient refractory period, after which the cell rests. A default rule allows the cell to stay in its present state.

In order to execute the Excitable Media Coupled model using a hexagonal neighborhood, we first define the model using hexagonal rules, and convert them into square compatible rules. The new hexagonal rules are:

```
[excitable-rule]
rule: 0 100 {[0]=0 and statecount(2)=0 }
rule: 2 100 {[0]=0 and statecount(2)>0 }
rule: 1 100 { [0] = 2 }
rule: 0 100 { [0] = 1 }
rule: { [0] } 100 { t }
```

These rules should now be translated into a square meshes, using the method depicted in Figure 11. The lattice translation converts the rules above into the equivalent hexagonal rules as follows:

```
[excitable-rule]
rule : 0 100 { (  (0,0)=0 and ( if( (statecount(2)-(if((-1,1=2,1,0)) -
             (if((1,1)=2,1,0)))< 0,0,(statecount(2)-(if((-1,1)=2,1,0)) -
                (if((1,1) = 2,1,0)))) = 0 ) ) and even(cellpos(1)) }
...
rule : {(0,0)} 100 { t and even(cellpos(1)) }
rule : {(0,0)} 100 { t and odd(cellpos(1)) }
```

In a square lattice, we use 8 neighbors, against 6 used in the hexagonal mesh. Therefore, when we translate the rules from one topology to the other, we should ask not to count those neighbors that should not be included. The function *statecount(n)*, which returns the number of cells whose value is n, is used with this purpose. We can see that we compute *statecount(9.99)* like in the previous case, and then, in the even rows, we subtract the values of cells ((-1,1) and (1,1)). The following rule is evaluated only in odd rows. The function *cellpos(n)*, which returns the n-eth value of the tuple referencing a cell is used to check the cell row used.

Figure 12 shows the simulation results for two initial scenarios (one with one seed and a second with two) for an excitable medium represented on a hexagonal mesh. The excitable media is set at the centre of the mesh and it can be seen that how it evolves over time. In the second case we show what happens when two excitable media are put up on a grid at a few cells apart.

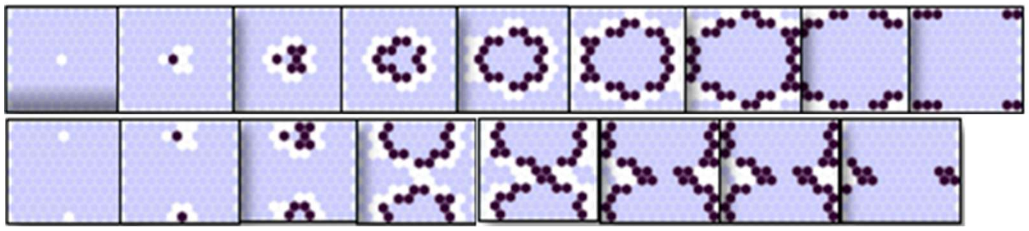
Figure 13: Simulation results for Hexagonal Mapping of the excitable media model.

Similarly, we can apply the procedure depicted in Figure 11 and define the excitable model using a triangular mesh; a similar selection should be made, and the resulting rules are as follows:

```
[excitable-rule]
rule : 0 100 { ( ( (0,0) = 0 ) and ( if((statecount(2) - (if((-1,-1) = 2,1,0)) -
             (if((-1,0) = 2,1,0)) - (if((-1,1) = 2,1,0)) - (if((1,-1) = 2,1,0)) -
                    (if((1,1) = 2,1,0))) < 0,0,(statecount(2) - (if((-1,-1) = 2,1,0))-
                       (if((-1,0)=2,1,0)) - (if((-1,1)=2,1,0)) - (if((1,-1)=2,1,0)) -
                    (if((1,1) = 2,1,0)))) = 0 ) ) and even(cellpos(0) + cellpos(1)) }
...
rule : {(0,0)} 100 { t and even(cellpos(0) + cellpos(1)) }
rule : {(0,0)} 100 { t and odd(cellpos(0) + cellpos(1)) }
```

Figure 13 shows the simulation results of the excitable media model on a triangular geometry considering the nearest 3 neighbors.

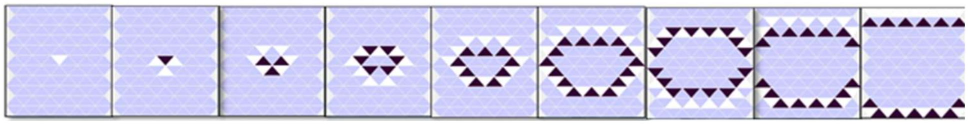
Figure 14: Simulation results for Triangular Mapping of the excitable media model.

5.2 Surface Tension

Surface tension can be defined as an elastic tendency of a fluid which influences it to obtain the slightest surface area possible. The model representing surface tension is considered as a majority voting system, and it was originally defined in (Ameghino and Wainer 2000). In this case, each step the new state of the cell will depend on most of its neighbors. Two types of states are defined to represent the cells presence and absence of particles corresponding to values 1 and 0. According to the majority voting system if at least 5 neighbors are occupied then the cell state remains the same else it changes.

```
[surfacetension]
type : cell      dim : (60,60)     border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1)  (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)
localtransition : tension

[tension]
rule : 0 100 { statecount(0) >= 5 }
rule : 1 100 { t }
```

We extended the surface tension rules and defined them using hexagonal geometry as done in section 4.1, using the hexagonal neighbor referencing from Figure 11. The model uses a majority voting specification to calculate the tension but as it considers hexagonal geometry only 6 neighbors are considered hence; minimum 4 neighbors are taken which can be occupied to make the majority. Using the lattice translator tool, the rules will be translated into square compatible rules for CD++ as follows:

```
[calculatetension-rule]
rule : 0 100 { ( if((statecount(0) - (if((-1,1) = 0,1,0)) - (if((1,1) = 0,1,0))) <
          0,0,(statecount(0) - (if((-1,1) = 0,1,0)) - (if((1,1) = 0,1,0)))) >= 4 ) and
              even(cellpos(1)) }
rule : 0 100 { ( if((statecount(0) - (if((-1,-1) = 0,1,0)) - (if((1,-1) = 0,1,0))) <
          0,0,(statecount(0) - (if((-1,-1) = 0,1,0)) - (if((1,-1) = 0,1,0)))) >= 4 )
                and odd(cellpos(1)) }
rule : 1 100 { t and even(cellpos(1)) }
rule : 1 100 { t and odd(cellpos(1)) }
```

Figure 15 shows the results for the hexagonal topology. Figure 14.a) shows the result for a 20x20 cell space, and Figure 14.b) shows a case for 50x50 cells. In both cases, we show the initial configuration of the distribution of particles, and the final scenario where the model simulation reaches a steady state.

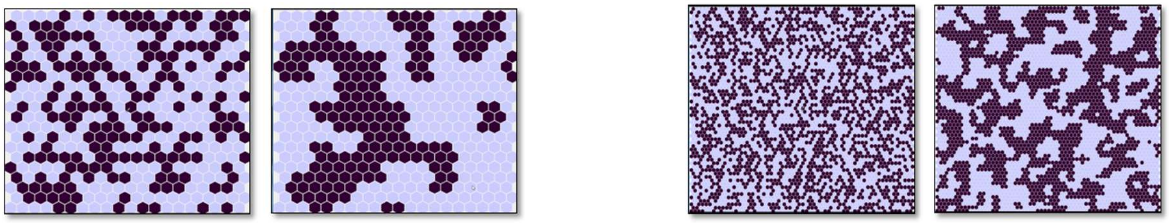
Figure 15: Simulation results for Hexagonal Mapping of the surface tension model.

As in the previous example, we also built the model using triangular rules, and translated it to a square lattice and executed the model in CD++. Triangular geometry considers nearest 3 neighbors hence, threshold of 2 neighbors are considered for calculating the majority of the votes. The following rule shows the modification of the generated rules generated by the lattice translator for the triangular mesh.

```
[tension]
rule : 0 100 { ( if((statecount(0) - (if((-1,-1) = 0,1,0)) - (if((-1,0) = 0,1,0)) -
          (if((-1,1) = 0,1,0)) - (if((1,-1) = 0,1,0)) - (if((1,1) = 0,1,0))) <
                 0,0,(statecount(0) - (if((-1,-1) = 0,1,0)) - (if((-1,0) = 0,1,0)) -
                    (if((-1,1) = 0,1,0)) - (if((1,-1) = 0,1,0)) - (if((1,1) =
                           0,1,0)))) >= 2 ) and even(cellpos(0) + cellpos(1)) }
...
rule : 1 100 { t and even(cellpos(0) + cellpos(1)) }
rule : 1 100 { t and odd(cellpos(0) + cellpos(1)) }
```

The translated rules are now included into the coupled model. The presence of particles is randomly distributed, and the coupled model is build using triangular topology.

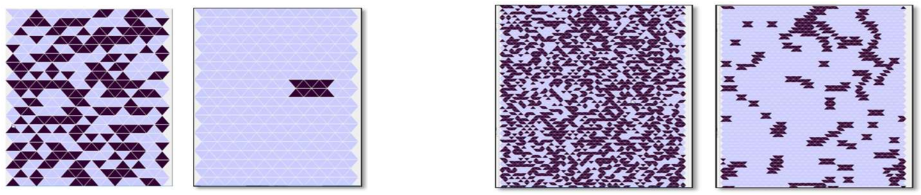

Figure 16: Simulation results for Triangular Mapping of the surface tension model.

CONCLUSION

Cell–DEVS allows describing physical systems using an n-dimensional cell-based formalism. In this case we have showed the definition of a number of different Cellular Automata using the CD++ tool, which permits defining complex cell-shaped models using a high-level specification language. We showed that various kinds of applications can be easily developed, allowing the study of different problems through simulation. Finally, the use of formal models improves the development, checking and maintaining phases, facilitating the testing and reuse of their components.

The tool and the examples are the public domain and they can be obtained at: http://cell-devs.sce.carleton.ca/

ACKNOWLEDGEMENTS

This work has been partially funded by NSERC. We want to acknowledge the work of V. Silva, Q. Liang, C. Ruiz-Martin and R. Pandey, who developed various versions of the models presented here.

REFERENCES

Ameghino, J. and Wainer, G. 2000. "Application of the Cell-DEVS paradigm using N-CD++", Proceedings of the 32nd SCS Summer Computer Simulation Conference, Vancouver, BC, Canada.

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. 2008. "A Performance Study of General-Purpose Applications on Graphics Processors using CUDA". *Journal of Parallel and Distributed Computing* 68(10):1370–1380.

Demongeot, J., E. Golés, and M. Tchuente. 1985. *Dynamical Systems And Cellular Automata*. New York, NY: Academic Press.

Gardner, M. 1970. "Mathematical Games – The Fantastic Combinations Of John Conway's New Solitaire Game "Life"". *Scientific American* 223(4):120–123.

Rybacki, S. and J. Himmelspach. 2009. "Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata". In *Proceedings of the First International Conference on the Advances in System Simulation*, 62-67. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Silverman, B. 1996. *Changing the Rules*. The Virtual Computer, Mathematical Association of America.

Wainer, G. 2009. *Discrete-event Modeling and Simulation: A Practitioner's Approach.* CRC/Taylor & Francis. Boca Raton, FL.

Wainer, G. and N. Giambiasi. 2001. "Application of the Cell-DEVS Paradigm for Cell Spaces Modeling and Simulation". *Simulation* 71(1):22-39.

Wainer, G., C. Ruiz-Martín, and A. López-Paredes. 2016. "Cellular Models for Emerging Traffic Behavior". *Cellular Automata Modeling for Urban and Spatial Systems*. Québec City, PQ. Canada. 2016.

Weimar, J. 1997. *Simulation with Cellular Automata*. Logos-Verlag, Berlin,.

Wilensky, U. 2002. Netlogo Brian's Brain Model. https://ccl.northwestern.edu/netlogo/models/Brian'sBrain, accessed 21st Sept 2019. Center For Connected Learning And Computer-Based Modeling, Northwestern University, Evanston, Il.

Wolfram, S. 1986. *Theory And Applications Of Cellular Automata*. Philadelphia, PA: World Scientific.

AUTHOR BIOGRAPHIES

GABRIEL A. WAINER, FSCS, SMIEEE, received the M.Sc. (1993) at the University of Buenos Aires, Argentina, and the Ph.D. (1998, with highest honors) at UBA/Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada), where he is now Full Professor and Associate Chair for Graduate Studies. He has held visiting positions at the University of Arizona; LSIS (CNRS), Université Paul Cézanne, University of Nice, INRIA Sophia-Antipolis, Université de Bordeaux (France); UCM, UPC (Spain), University of Buenos Aires, National University of Rosario (Argentina) and others. He is one of the founders of the Symposium on Theory of Modeling and Simulation, SIMUTools and SimAUD. Prof. Wainer was Vice-President Conferences and Vice-President Publications, and is a member of the Board of Directors of the SCS. Prof. Wainer is the Special Issues Editor of SIMULATION, member of the Editorial Board of IEEE Computing in Science and Engineering, Wireless Networks (Elsevier), Journal of Defense Modeling and Simulation (SCS). He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He has been the recipient of various awards, including the IBM Eclipse Innovation Award, SCS Leadership Award, and various Best Paper awards. He has been awarded Carleton University's Research Achievement Award (2005, 2014), the First Bernard P. Zeigler DEVS Modeling and Simulation Award, the SCS Outstanding Professional Award (2011), Carleton University's Mentorship Award (2013), the SCS Distinguished Professional Award (2013), and the SCS Distinguished Service Award (2015). He is a Fellow of SCS. His email address is gwainer@sce.carleton.ca.