Simulation

Special Issue: Simulation with Cellular Automata

Simulation: Transactions of the Society for Modeling and Simulation International
2016, Vol. 92(2) 101–115

DOI: 10.1177/0037549715611485
sim.sagepub.com
SAGE SCS

Modelling and simulation of complex cellular models using Cell-DEVS

Gabriel Wainer[1*] and Joaquín Fernández[2]

Abstract

Cellular automata are discrete dynamical systems that provide a mathematical framework for modelling, studying and predicting the behaviour and response of systems across many different disciplines and domains, ranging from physical and biological to computational and social models. Cell-DEVS is a formalism that provides a discrete event approach to define cellular models with timing delay constructions and using simple definition of complex timing. It has been shown that the application of the Cell-DEVS paradigm produces a significant reduction in the development times of cell-shaped models and a wide variety of complex models has been developed using this approach. In this work we present the definition of complex cellular automata models using the Cell-DEVS paradigm, we use the CD++ tool to obtain executable models and study their behaviour through computer simulation.

Keywords

discrete event simulation, cellular models, cellular automata

1. Introduction

Cellular automata (CA) are discrete dynamical systems that provide a mathematical framework for modelling, studying and predicting the behaviour and response of systems across many different disciplines and domains, ranging from physical and biological to computational and social models.

CA are defined as a lattice of cells, where each cell takes on a finite set of possible values that are updated at discrete time steps in a synchronous way according to a set of rules that depends on the state of some nearby cells (the neighbourhood). The state of the lattice at time zero $t = 0$ is referred to as the initial configuration. In subsequent time steps $(t + 1, t + 2, t + 3, \ldots, t + n)$ the state of each cell can be determined by the current state of the cell and its neighbours. The resulting configuration of the cell values defines the state of the system in the next time step. In this way, the configurations at different points in time describes the discrete evolution of many identical cells and despite their simple construction, they are shown to be capable of complex behaviour and to generate complex patterns with universal features.[1]

CA were originally introduced by John von Neumann as ideal structures for the construction of a self-replicating machine; in his work, he was able to describe the first CA capable of self-reproduction. A condition required by von Neumann for self-reproduction was the capacity for universal construction (i.e., the CA was designed to construct any machine described as input) to exclude passive or trivial replication, as a consequence the design of his machine was very complex. The model presented by von Neumann consists of a two-dimensional array with an initial configuration of thousands of cells and 29 possible states for each cell. This model was revised and simplified by Codd[2] where he was able to describe a self-reproducing CA using only eight states per cell but the initial configuration of the machine still requires thousands of cells. The concept of self-reproduction in CA was revised by Langton[3] where he stated that although the capacity for universal construction is a sufficient condition for self-reproduction, it is not a necessary condition. Following this idea he demonstrated the existence of simple self-reproducing structures that can be embedded in CA using

[1]Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada
[2]Laboratorio de Sistemas Dinámicos, CIFASIS-CONICET, Rosario, Argentina

[*]SCS member

Corresponding author:
Gabriel Wainer, Department of Systems and Computer Engineering, Carleton University, 4456 Mackenzie Building, 1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada.
Email: gwainer@sce.carleton.ca

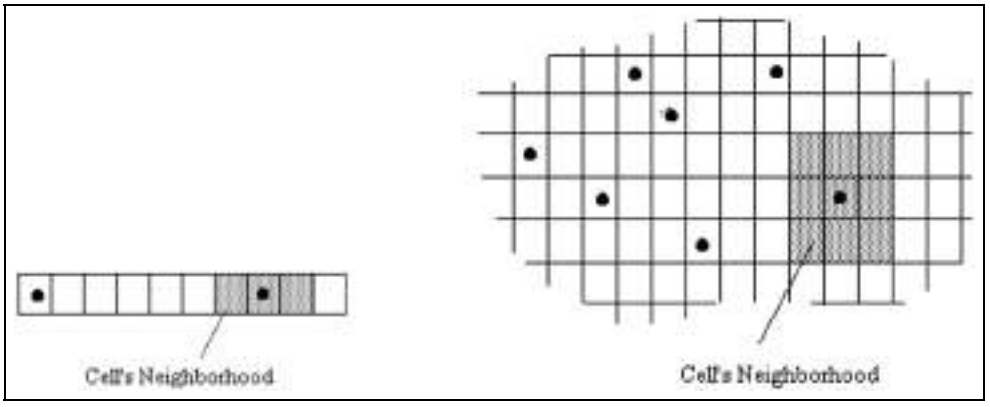

Figure 1. Possible neighbourhood definition for a cell in a one-dimensional and a two-dimensional CA, respectively.

only a rectangular area of just 10×15 cells without requiring the capacity of universal construction.

On the other hand, Weisstein[4] presented the simplest class of one-dimensional CA is presented. These elementary CA consist of two possible state values for each cell and the evolution rules depends on the current state of each cell and the values of the nearest neighbours. A total of 256 elementary CA can be defined that exhibit different kinds of output behaviour, in particular, the elementary CA called Rule 110 (due to the Wolfram numbering scheme[5]) although trivial to define, has been proven to be capable of universal computation,[6] emulating the activity of a Turing machine by encoding the Turing machine and its tape into a repeating left pattern, a central pattern and a repeating right pattern on which Rule 110 then acts on.

In order to be able to study, analyse and predict the behaviour of these theoretical CA models computer simulation of executable models is required. The use of a modelling formalism improves the development times of executable models. A formal paradigm can make the definition of models for the development of simulations easier.

The Cell-DEVS[7] formalism is based on the discrete event systems specification (DEVS) paradigm[8] and provides a discrete event approach to define cellular models with timing delay constructions and using simple definition of complex timing. Cell-DEVS models are defined as a space composed of individual cells that can be coupled to form a complete cell space where each cell is a continuous time model defined by simple rules and a few parameters.

It has been shown that the application of the Cell-DEVS paradigm produces a significant reduction in the development times of cell-shaped models[7] and a wide variety of complex models has been developed using this approach.[9–12]

The Cell-DEVS specification was used to develop the CD++ modelling and simulation tool[13] that allows the definition of executable models providing a high-level specification language where local transition rules can be easily defined and extended.

In this work we present the definition of complex CA models using the Cell-DEVS paradigm, we use the CD++ tool to obtain executable models and study their behaviour through computer simulation. We apply this approach to show that different kinds of applications can be easily faced. Towards this end, we present executable models for the elementary Rule 110 CA mentioned above, a three-state two-colour Turing machine presented by Weisstein[14] and a simple self-reproducing CA with shape encoding mechanism presented by Morita and Imai.[15]

2. Background

Cellular automata are discrete dynamical systems, they are defined as an infinite regular n-dimensional lattice whose cells can take finite value. The states in the lattice are updated according to a local rule in a simultaneous and synchronous way. The cell states change in discrete time steps as dictated by the local transition function using the present cell state and a finite set of nearby cells called the neighbourhood of the cell (Figure 1).

The evolution of a CA at successive time steps generates different output patterns and although they have simple construction, some CA are capable of complex behaviour. Based on the output behaviour, the CA evolution can be characterized into four different classes:[1,16]

1. evolution leads to a homogeneous state;
2. evolution leads to set of separated stable or periodic structures;
3. evolution leads to a chaotic pattern;
4. evolution leads to complex localized structures.

A wide range of complex systems can be modelled using CA from biological processes[17,18] to physical and computational systems[19–21] where sufficiently complicated CA have been shown to be computationally ''universal'', as they behave as a general-purpose computer.[22]

The Cell-DEVS[7] formalism is based on the DEVS[8] formalism, a continuous time technique. The main goal of Cell-DEVS is to build discrete-event cell spaces.

DEVS is a system specification formalism that provides a framework for the construction of hierarchical and modular models allowing model reuse. A DEVS model is composed of atomic sub-models that can be combined into coupled models. Formally, an atomic DEVS model is defined by the following structure:

$$M = (\mathbf{X}, \mathbf{S}, \mathbf{Y}, \delta_{int}, \delta_{ext}, \lambda, \mathbf{D})$$

where **X** is the set of input event values, i.e., the set of all possible values that an input event can adopt, **S** is the set of state values, **Y** is the set of output event values and λ is the output function. For each DEVS atomic model, the external transition function δ_{ext}, the internal transition function δ_{int} and the state time advance function **D** define the system dynamics that can be described in the following way.

- When an input event arrives, the external transition function is executed and the state changes instantaneously. The new state value depends not only on the input event value but also on the previous state value and the elapsed time since the last transition.
- The time advance function returns a non-negative real number saying how long the system remains in a given state in the absence of input events.
- When the time units dictated by the time advance function are consumed, the output function is executed producing an output event that depends on the current state of the model. In addition, the internal transition function is executed and it can change the state of the model. The new value depends on the current state.

Atomic DEVS models can be coupled and DEVS theory guarantees that the coupling of atomic DEVS models defines new DEVS models (i.e., DEVS is closed under coupling).[8] Coupling in DEVS is usually represented through the use of input and output ports. With these ports, the coupling of DEVS models becomes a simple blockdiagram construction. Formally, these models are defined as

$$CM = \langle \mathbf{X}, \mathbf{Y}, \mathbf{D}, M_i, I_i, Z_{ij}, \mathbf{select} \rangle$$

where **X** is the set of input events, and **Y** is the set of output events. Here **D** is the set of indexes of the components, and for each $i \in \mathbf{D}$, M_i is a basic DEVS model, I_i is the set of influences of model i and **select** is the tie-breaking selector. Finally, for each $i \in \mathbf{D} \vee j \in I_i$, Z_{ij} is the i to j translation function. The translation function defines which output of model M_i are connected to the inputs of model M_j.

Cell-DEVS combines CA and DEVS allowing the definition of models as a space composed of individual cells that can be coupled. Each cell in the cell space is defined as an atomic model using timing delays with the following definition:

$$TDC = \langle \mathbf{X}, \mathbf{Y}, \mathbf{I}, \Theta, \mathbf{N}, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D \rangle$$

where **X** defines the set of external input events, **Y** is the set of external output events, **I** represents the interface of the model, Θ is the set of states for the cell, **N** is the neighbourhood set for the cell, *delay* defines the kind of delay used and d defines the delay duration, τ is the local computation function and, finally, δ_{int}, δ_{ext}, λ and D have the same semantics of a DEVS atomic model.

A cell uses a set of input values **N** to compute its future state, which is obtained by applying the local computation function τ. A *delay* function is associated with each cell, deferring the output of the new state to the neighbour cells. There are various types of delays, for instance, *inertial* and *transport* delays. When a *transport* delayed is used, the future value will be added to a queue sorted by output time. Therefore, all previous values that were scheduled for output but that have not yet been sent, will be kept. In contrast, *inertial* delays use a preemptive policy: any previous scheduled output value, unless the same as the new computed one, will be deleted and the new one will be scheduled. This activation of the local computation is carried by the δext function.

After the basic behaviour for a cell is defined, the complete cell space is constructed by building a coupled Cell-DEVS model:

$$GCC = \langle \mathbf{X_{list}}, \mathbf{Y_{list}}, I, \mathbf{X}, \mathbf{Y}, n, \{t_1, \ldots, t_n\}, \mathbf{N}, \mathbf{C}, \mathbf{B}, Z \rangle$$

where $\mathbf{X_{list}}/\mathbf{Y_{list}}$ are the input/output coupling lists, I represents the definition of the interface for the model, **X** and **Y** are the set of external input/output events, n is the dimension of the cell space, $\{t_1, \ldots, t_n\}$ is the number of cells on each of the dimensions, **N** is the neighbourhood set, **C** is the cell space, **B** is the set of border cells and Z is the translation function.

This specification defines a coupled model composed of an array of atomic cells. Each cell is connected to the cells defined in the neighbourhood, but as the cell space is finite, either the borders are provided with a different neighbourhood than the rest of the space, or they are *wrapped*, meaning that cells in one border are connected with those in the opposite one. Finally, the Z function defines the internal and external coupling of cells in the model.

The Cell-DEVS specification was used to develop the CD++ modelling and simulation tool.[13] In this tool the behaviour of atomic models are defined as C++ functions while a specification language is used to define coupled models. The language allows the definitions of size, influences, neighbourhood and borders according to the definition given above. In this way, we can construct the

Table 1. State transition.

Neighbourhood	111	110	101	100	011	010	001	000
New state	0	1	1	0	1	1	1	0

complete cell space. The behaviour of the local computing function is defined using a set of rules, with the following syntax:

```
VALUE DELAY {CONDITION}
```

According to this definition, when the `CONDITION` is satisfied the new state of the cell is set to `VALUE` after `DELAY` units of time.

2.1. Related work

An implementation of the Rule 110 CA can be found in Martínez et al.[23] where the different periodic structures (known as gliders), their properties and production due to collision found in the evolution space of the CA are extensively studied using executable models generated by the OSXLCAU21 and NXLCAU21 computational systems. The OSXLCAU21 system is specifically developed in order to satisfy the necessity of a detailed study in the evolution space of Rule 110 and the NXLCAU21 system is developed for the study of the evolution of two-dimensional CA.

Vukašinovic et al.[24] presented executable models for deterministic and non-deterministic Turing machines. These models are implemented using the symbolic processing and functional programming primitives of the Mathematica package. Mathematica is a commercial software program used in scientific, engineering, mathematical and computing fields, based on symbolic mathematics.

Finally, following the line of work introduced by Langton[3] as to what constitutes genuine self-reproduction, Byl[25] presents a simplified theoretical model of a small self-reproducing CA where he describes a six-state CA model in which the minimal configuration needed for self-reproduction consists of only 10 cells. Following this ideas, different theoretical self-reproduction CA models are found in Morita and Imai[26] and Reggia et al.[27]

This research shows how to define a generic and systematic approach to obtain executable models of theoretical CA. All of the models described in this work are classic synchronous CA (which implies that the timing definition for this models is simple and they do not take advantage of the Cell-DEVS asynchronous nature, thus, the type of delay used will not affect the output behaviour and for this reason in the specifications presented in this work the *transport* and *inertial* delays can be used without affecting the simulation results). Our goal is to show the advantages of using a modelling formalism and how this approach facilitates the model development for the end user while providing the means to define more advanced models (which could include asynchronous behaviour for different delay functions). Towards this end, the following sections are devoted to the application of the Cell-DEVS paradigm and related tools to these kind of models.

3. Rule 110 model

In this section we present a Cell-DEVS definition of the theoretical Rule 110 CA presented by Wolfram.[22] Rule 110 is defined as an elementary CA, where a cell can be *active* (1) or *inactive* (0) depending on the states of the immediate left, centre and right cells (the neighbourhood). Since there are $2 * 2 * 2 = 8$ possible combinations for the cells in the neighbourhood, total of 256 elementary CA can be defined by a table specifying the new state for a cell according to the value of the left neighbour, the actual state of the cell and the right neighbour. Each elementary CA is indexed using an 8-bit binary number according to all the possible output states. Table 1 shows the specification for the Rule 110 ($110 = 01101110_2$) CA.

Rule 110 has been described as capable of universal computation. At its simplest, Rule 110 is a simple Turing machine (capable of universality which means that many of their properties will be non-decidable, and not amenable to closed-form mathematical solutions). The function of the universal machine in Rule 110 requires an infinite number of localized patterns to be embedded within an infinitely repeating background pattern. The background pattern is 14 cells wide and repeats itself exactly every seven iterations (the pattern is 00010011011111). It has been shown that this CA generates a class 4 behaviour (that is neither stable nor chaotic).

The evolution over time of the execution of Rule 110 (which is an elementary one-dimensional CA) can be visualized using a two-dimensional representation where each row develops from the previous row with the number of rows representing the desired time units. Figure 2 shows an example of the execution of the Rule 110 CA for 600 units of time where the *active* cells (1) are represented by the black colour and *inactive* cells (0) are represented by the white colour.

Figure 2 shows three localized patterns of particular importance in the Rule 110 universal machine, surrounded

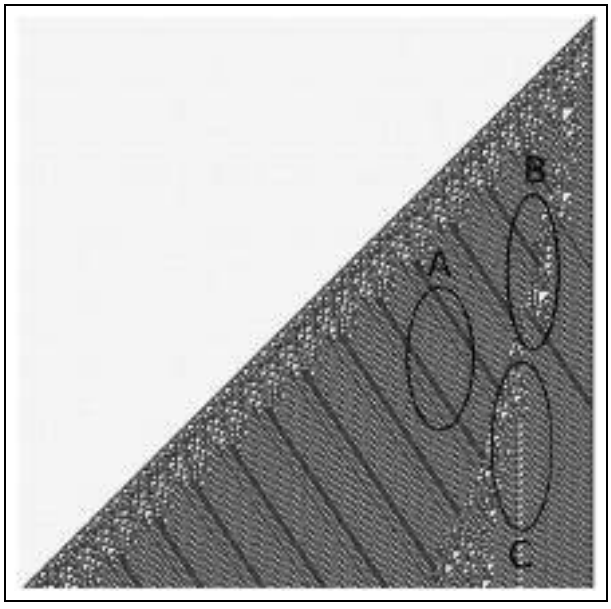

Figure 2. Sequential execution of the Rule 110 CA, each row of the two-dimensional representation reflects the current state of the CA for 600 units of time (600 × 600 cell space).

```
[Rule110]
type : cell
dim : (75,75)
delay : transport
border : nowrapped
neighbors : Rule110(-1,-1) Rule110(-1,0) Rule110(-1,1)
neighbors :                Rule110(0,0)
initialValue : 0
localtransition : Rule110-rule
```

Figure 3. Rule 110 CD++ coupled model definition.

by a repeating background pattern. The leftmost structure shifts two cells to the right and repeats every three generations. It comprises the sequence 0001110111 surrounded by the background pattern given above, as well as two different evolutions of this sequence. These patterns are present and highlighted by the circles labelled **A** and **B** respectively. The circle labelled **C** represents the interaction of these two structures as they pass through each other without interference to produce a third unique structure, deemed as self-reproduction.

In the following, we show a model specification for Rule 110 implemented on CD++ presented in Figure 3 as a coupled Cell-DEVS model with a 75 × 75 cellular space, using nonwrapped border, a transport delay and the neighbourhood described in Table 1. As previously stated, the two-dimensional representation of this one-dimensional CA allow us to observe how the model evolves over time, thus, in this model specification we will be able to see the evolution of the CA for the first 75 units of time.

```
[Rule110-rule]
rule : 1        1 {(((-1,0) = 0 and (-1,1) = 1 )
                    or ((-1,-1) = 0 and (-1,0) = 1)
                    or ((-1,0) = 1 and (-1,1) = 0))}
rule : {(0,0)} 1 {t}
```

Figure 4. Rule 110 transition rules.

For the transition rules implementation, taking into account that Rule 110 is a binary combination of the previous neighbour's data value, the Rule 110 logic can be described as a sum of products:

$$Rule110(A,B,C) = (AB\bar{C}) + (A\bar{B}C) + (\bar{A}BC) + (\bar{A}B\bar{C}) + (\bar{A}\bar{B}C) \quad (1)$$

where:

- A represents the left neighbour;
- B represents the centre neighbour;
- C represents the right neighbour.

Then, we use a Karnaugh Map to reduce Equation (1) to its simplest representation:

$$Rule110(A,B,C) = (A\bar{B}) + (B\bar{C}) + (\bar{B}C) \quad (2)$$

As we mentioned above, Rule 110's state is either a 1 or a 0. In order to be able to visualize the execution of this model, this 1-D CA was represented using a two-dimensional Cell-DEVS. Figure 4 shows the CD++ implementation of these rules.

The first rule implements Equation (2). The second transition rule is used to retain the current value of the cell if the precondition for (2) does not hold.

The results of the CD++ model execution for a 75 × 75 cell space using the Cell-DEVS Animation tool are presented in Figure 5. The opening row has the initial condition that all data values are 0 except for the rightmost digit which is a 1. With these initial values, the data evolves into the pattern shown in Figure 5a. The initial values of the execution of the model are shown in Figure 5b. We can see that for the last cell of the second row, the centre neighbour has its state set to 1 and the left neighbour has its state set to 0 ((–1, 0) and (–1, –1) in the CD++ neighbourhood description of the model respectively), thus, transition rule number 1 holds and the new state of the cell is set to 1. Similarly, for the next to last cell of the second row the centre neighbour has its state set to 0 and the right neighbour has its state set to 1, therefore, we can apply transition rule number 1 and the new cell state is set to 1. The evolution from the second to the third row and from the third to the fourth row can be obtained following the same reasoning and is shown in Figures 5b and 5e.

The executable model presented in this section allows the study of different periodic structures that appear in the

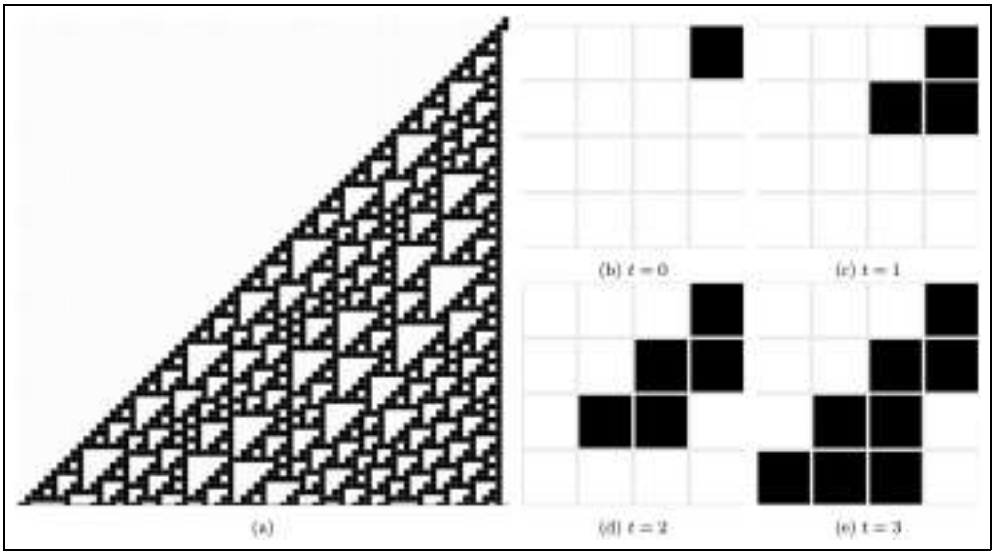

Figure 5. Rule 110 simulation results.

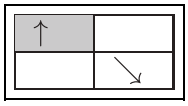

Figure 6. Three-state two-colour Turing machine template instruction example.

evolution of the Rule 110 CA as well as the complex behaviour that emerges from the execution.

4. Three-state two-colour Turing machine

Weisstein[14] specified a CA template for a three-state, two-colour Turing machine. In this template the state of each cell is characterized by a square that contains:

- a pointer indicating the direction, {↑, ↘, ↙};
- the colour of the cell, {"black", "white"}

In order to be able to define the machine behaviour the template defines an instruction that is represented by three squares, with the top one representing a possible direction and colour of the active cell and the bottom ones giving the new colour the active cell and the direction in which the tape should be moved. The special state 0 (with no pointer) indicates a state at which the Turing machine should halt (i.e., cease computation).

Figure 6 shows a possible definition of an instruction. This instruction indicates that if the direction of the active cell is set to ↑ and its colour is set to "black", then the new colour of the cell is set to "white" and the direction of the tape should be set to ↘, where the active cell state is represented by the top square and the new colour of the cell and its direction are represented by the left and right bottom squares, respectively.

Thus, for a concrete implementation of the template, a total of $3\bigstar2 = 6$ instructions that define the behaviour of the machine for all possible states has to be provided. In addition, taking into account the empty square state (with no pointer), each cell has eight possible states. We can see that in the general case of a k-state n-colour Turing machine we need $k\bigstar n$ rules to define the behaviour of the machine and each cell will have $(k + 1)\bigstar n$ possible states.

The model implemented in this section derives from the one originally presented by Wolfram[22], and it represents an example of a concrete implementation of the CA template for a three-state two-colour Turing machine. The corresponding set of rules are shown in Figure 7.

In this model, the semantics of the first instruction represented in Figure 7a were explained previously, the second instruction, depicted in Figure 7b, states that if the direction of the active cell is set to ↑ and its colour is set to "white", then the new colour of the cell is set to "black" and the direction of the tape should be set to ↙. In this way, we can interpret each of the instructions of the template and we can use a two-dimensional representation of this CA in order to be able to visualize the evolution over time as we did with the Rule 110 model in the previous section.

The complete set of instructions defined in Figure 7 was implemented in CD++ as a coupled Cell-DEVS model. The coupled model consist of a 15×31 cellular space, which means that in this model we will visualize the evolution of the CA for 31 units of time, an unwrapped border and a transport delay. The model uses Moore's neighbourhood (i.e., the nine nearest neighbours of the

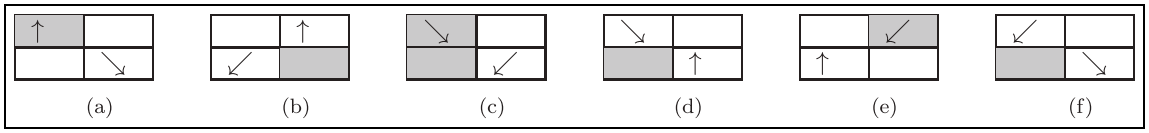

Figure 7. Three-state two-colour Turing machine instruction concrete implementation.

```
[TuringMachine]
type : cell
dim : (15,31)
delay : transport
border : nowrapped
neighbors : TuringMachine(-1,-1) TuringMachine(-1,0) TuringMachine(-1,1)
neighbors : TuringMachine(0,-1)  TuringMachine(0,0)  TuringMachine(0,1)
neighbors : TuringMachine(1,-1)  TuringMachine(1,0)  TuringMachine(1,1)
localtransition : TuringMachine-rule
```

Figure 8. Three-state two-colour Turing machine CD++ specification.

Table 2. Turing machine state mapping.

State			↑	↑	↘	↘	↙	↙
Number	0	1	2	3	4	5	6	7

```
[TuringMachine-rule]
rule : 4 1 { (-1, -1) = 3 and (-1, 0) = 0 }
rule : 5 1 { (-1, -1) = 3 and (-1, 0) > 0 }
rule : 0 1 { (-1, 0) = 3 }
```

Figure 9. Three-state two-colour Turing machine transition rules.

cell), the state mapping for the machine is depicted in Table 2 while Figure 8 shows the specification for this model.

Once the coupled model is defined, we need to implement the rules that represent the behaviour of the machine from the instructions in Figure 7, and they can be easily translated into CD++ transition rules. For example, in order to translate the first rule (Figure 7a) we need to recognize two configurations.

1. If the neighbour located in the north-west has its colour property set to ''black'' and the direction is set to ↑ and the west neighbour has no direction set, the new colour of the cell is set to the colour of the west neighbour and the direction is set to ↘.
2. If the neighbour located in the north has its colour property set to ''black'' and the direction set to ↑, then the new colour of the cell is ''white''.

Figure 9 shows the CD++ rules that implement the conditions described above using the state mapping from Table 2. Rules 1 and 2 implement first configuration while the last rule represents the second possible configuration.

```
[TuringMachine-rule]
rule : 1 1 { (0, 0) = 1 }
rule : 2 1 { (0, 0) = 2 }
rule : 3 1 { (0, 0) = 3 }
rule : 4 1 { (0, 0) = 4 }
rule : 5 1 { (0, 0) = 5 }
rule : 6 1 { (0, 0) = 6 }
rule : 7 1 { (0, 0) = 7 }
rule : 4 1 { (-1, -1) = 3 and (-1, 0) = 0 }
rule : 5 1 { (-1, -1) = 3 and (-1, 0) > 0 }
rule : 0 1 { (-1, 0) = 3 }
rule : 6 1 { (-1, 1) = 2 and (-1, 0) = 0 }
rule : 7 1 { (-1, 1) = 2 and (-1, 0) > 0 }
rule : 1 1 { (-1, 0) = 2 }
rule : 6 1 { (-1, -1) = 5 and (-1, 0) = 0 }
rule : 7 1 { (-1, -1) = 5 and (-1, 0) > 0 }
rule : 1 1 { (-1, 0) = 5 }
rule : 2 1 { (-1, -1) = 4 and (-1, 0) = 0 }
rule : 3 1 { (-1, -1) = 4 and (-1, 0) > 0 }
rule : 1 1 { (-1, 0) = 4 }
rule : 2 1 { (-1, 1) = 7 and (-1, 0) = 0 }
rule : 3 1 { (-1, 1) = 7 and (-1, 0) > 0 }
rule : 0 1 { (-1, 0) = 7 }
rule : 4 1 { (-1, -1) = 6 and (-1, 0) = 0 }
rule : 5 1 { (-1, -1) = 6 and (-1, 0) > 0 }
rule : 1 1 { (-1, 0) = 6 }
rule : 1 1 { (-1, 0) > 0 }
rule : 0 1 { t }
```

Figure 10. Three-state two-colour Turing machine transition rules.

In this way, any specification given for a concrete implementation of a three-state two-colour Turing machine can be easily represented using Cell-DEVS. The complete CD++ implementation is depicted in Figure 10.

Finally, Figure 11 shows the results of simulating the CD++ model with three different initial conditions. The first row indicates the initial condition of the machine and the evolution from one row to the next is determined by the instructions that can be applied and represents the changes of the CA over time. For example, in Figure 11a,

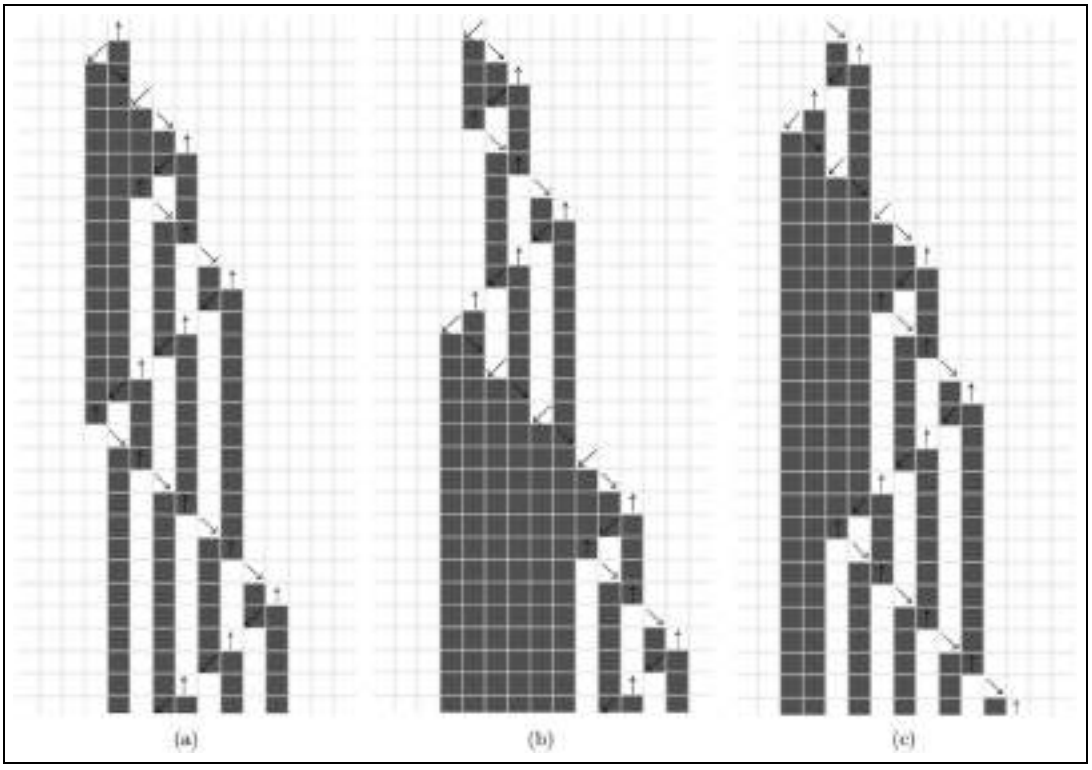

Figure 11. Three-state two-colour Turing machine simulation results with different initial configurations.

at time 0 the fifth column has the value 2 and the rest of the cell space is set to 0. With these initial conditions we can apply the instruction described in Figure 7b as follows:

- The cell located at the fourth column has its north-east neighbour direction set to ↑ and the colour set to ''white'', also the neighbour located at the west has no direction set and its colour set to ''white'', as a consequence the new cell direction is set to ↙ and the colour is set to ''white'' (state number 6).
- The cell located at the fifth column has its north neighbour direction set to ↑ and the colour set to ''white'', thus the new cell colour is set to ''black'' and no direction is set (state number 1).

Similarly, for the initial conditions given in Figure 11b we can apply the instruction from Figure 7f and for Figure 11c we can apply the instruction from Figure 7d. In this way starting from the given initial conditions we can obtain the corresponding values for each row in the model execution shown in Figure 11.

The Cell-DEVS executable model described in this section can be easily modified in order to obtain a different three-state two-colour Turing machine concrete implementation to study the behaviour of this kind of complex CA models.

5. Self-reproducing CA

In this section we introduce a model of a self-reproducing CA, originally introduced by Morita and Imai[15] where a model of a 12-state CA SR_{12} is defined. In this CA, signal transmission lines called *Worms* and *Loops* can self-reproduce using a shape-encoding method (the minimum number of cells needed for self-reproduction are four for the *Worms* and eight for the *Loops*). SR_{12} is a CA with von Neumann neighbourhood (i.e., top, down, left and right cells) and the following state set:

$$Q = \{\#, \uparrow, \downarrow, \rightarrow, \leftarrow, \mathbf{S}, \mathbf{R}, \mathbf{L}, \mathbf{B}, \bigstar, \bullet, +\}$$

which can be organized into three groups as follows:

1. ''Signal transmission'' states $\{\uparrow, \downarrow, \rightarrow, \leftarrow\}$, used to define signal movements in the direction indicated by the arrow.
2. ''Signal encoding'' states $\{\mathbf{S}, \mathbf{R}, \mathbf{L}, \mathbf{B}\}$, used to define the shape of the object; the meaning of each of them will be defined in the following paragraphs.
3. Finally, $\{\#, \bigstar, \bullet, +\}$ are control states, where:
   (a) # is the quiescent state;
   (b) ★ is used in *Worms* and *Loops* reproduction rules;
   (c) • is used in *Loops* reproduction rules;
   (d) + is used in *Worms* reproduction rules.

Table 3. The SR_{12} state mapping.

State	∅	↑	→	←	↓	**S**	**L**	**R**	**B**	★	•	+	#
Number	0	1	2	3	4	5	6	7	8	9	10	11	12

In this CA, signal transmission lines are formed by placing ''signal transmission'' and signal ''encoding states'' alternately. As we mentioned above, we can define two signal transmission line configurations.

- *Worms* which are simple transmission lines with open ends. Thus, they have a *head* (an end cell to which signals flow) and a *tail* (an end cell from which signals flow).
- *Loops* which are closed transmission lines.

For both configurations, the signal encoding states act as commands with the following associated operations:

1. **S** advance the head of the transmission line straight ahead;
2. **L** advance the head of the transmission line to the left;
3. **R** advance the head of the transmission line to the right;
4. **B** branch the head of the transmission line in three ways.

In the case of *Worms* configurations, the ''signal encoding'' states **S**,**R**,**L** and **B** are decoded and executed at the *head* of the transmission line. It is worth mentioning that if a **B** signal is found, the self-reproduction process begins by creating a three-way branch at the *head* of the transmission line. After this step, signals coming to this point are copied and propagated in three ways until the *tail* of the transmission line is processed. On the other hand, the encoding process of ''signal transmission'' states is the reverse of decoding and takes place in the *tail* of the transmission line.

For the *Loops*, if the configuration consist only of **S**, **L**, **R** or **B** encoding states, the transmission line simply rotates, and self-reproduction does not occur (simple *Loops*). However if a • control state is found at some appropriate position self-reproduction begins. The process can be summarized as follows:

1. When the state • reaches a corner, it first makes an ''arm'' and it constructs a daughter *Loop*.
2. Then the shape of the mother *Loop* is encoded into a sequence of advance commands. This process is controlled by the states ★ and • which are generated when the arm is created. These states go through all of the cells of the mother *Loop*, and they return back to the branching point. Then, the ''arm'' is cut off.
3. Finally, the daughter *Loop* acts like a *Worm* until the *head* meets the *tail* of the transmission line.

```
[SR–CA]
type : cell
dim : (60,60)
delay : transport
border : wrapped
neighbors :                    SR–CA(1,0)
neighbors : SR–CA(0,-1) SR–CA(0,0) SR–CA(0,1)
neighbors :                    SR–CA(-1,0)
initialvalue : 0
initialCellsValue : init.val
localtransition : TransmissionLine
```

Figure 12. The SR_{12} CD++ model definition.

After the self-reproduction process ends, the property is inherited by the daughter *Loop*.

This model description is implemented in CD++ as a coupled Cell-DEVS model. The coupled model consists of a 60×60 cellular space, a wrapped border, transport delay and a von Neuman neighbourhood. Figure 12 shows the CD++ specification. In addition, the state mapping for the *CA* is depicted in Table 3.

Once an initial configuration is given, the model will behave according to the local transition rules defined in `TransmissionLine`.

In the case of *Worms* and simple *Loops* configurations, we have to define rules for decoding and encoding the state signals as well as the rules in charge of the movement of the body.

As described by Morita and Imai[15], signal decoding requires three steps.

1. If only one ''signal encoding'' state is detected in the neighbourhood of the cell, and the cell is empty then the cell can potentially become the *head* of the transmission line. Thus, it must wait in the quiescent state (#) for the new values of the neighbours.
2. Of all of the candidate cells, if only one ''signal transmission'' state (an arrow) is detected that points in the direction of the cell, then the cell becomes active (★). It is worth mentioning that in this step more than one cell can be active if a branch **B** command is being processed.
3. Finally, for all of the active cells in the previous step, the only ''signal encoding'' state present in the neighbourhood becomes the new state for the cell.

```
[TransmissionLine]
.
.
.
% Transmission line head decoding rules.

#beginMacro(isHeadS1)
((stateCount(0) = 4 and (0,0) = 0 and
(stateCount(5) = 1 or stateCount(6) = 1
or stateCount(7) = 1 or stateCount(8) = 1)) or
(stateCount(0) = 3 and stateCount(12) = 1 and (0,0) = 0 and
(stateCount(5) = 1 or stateCount(6) = 1 or
stateCount(7) = 1 or stateCount(8) = 1)))
#endMacro

#beginMacro(isHeadS2)
(stateCount(12) = 1) and ((0,0) = 12) and
(stateCount(1) = 1 or stateCount(2) = 1 or
stateCount(3) = 1 or stateCount(4) = 1 or stateCount(11) = 1)
#endMacro

#beginMacro(isHeadS3)
(0,0) = 9 and
(stateCount(5) = 1 or stateCount(6) = 1 or
stateCount(7) = 1 or stateCount(8) = 1)
#endMacro
.
.
.
rule : 12 1 {(#macro(isHeadS1))}
rule : 9  1 {(#macro(isHeadS2)) and
             ((0,-1) = 2 or (-1,0) = 4
             or (0,1) = 3 or (1,0) = 1
             or (-1,0) = 11 or (0,1) = 11
             or (1,0) = 11 or (0,-1) = 11)}
rule : {(0,-1)} 1 {(#macro(isHeadS3))
                   and (0,-1) > 4 and (0,-1) < 9}
rule : {(0,1)}  1 {(#macro(isHeadS3))
                   and (0,1) != 0 and (0,1) < 9}
rule : {(1,0)}  1 {(#macro(isHeadS3))
                   and (1,0) != 0 and (1,0) < 9}
rule : {(-1,0)} 1 {(#macro(isHeadS3))
                   and (-1,0) != 0 and (-1,0) < 9}
```

Figure 13. Transition rules for decoding *head* signals.

Figure 13 shows the transition rules implemented for signal decoding at the *head* of a *Worm* configuration. For example, the first rule corresponds to step number 1, the second rule corresponds to step number 2 and rules 3–6 correspond to step number 3. The macros `isHeadS1`, `isHeadS2` and `isHeadS3` are in charge of detecting the three scenarios described above.

Similarly, signal encoding at the *tail* of the transmission line takes three steps that involve:

- detecting the *tail* of the transmission line (only one cell can be the *tail*);
- retracting the *tail* by one cell;
- generating the corresponding signal encoding state.

For the movement of the body of transmission lines, we have two scenarios. If the current cell state is a ''signal transmission'' state, the new cell state corresponds to the ''signal encoding'' state in the neighbourhood that appears in the opposite direction of the current state (i.e., the opposite direction of the ''arrow''). On the other hand, if the current cell state is a ''signal encoding'' state, we have to take into account the location of the signal transition states present in the neighbourhood and their direction.

Figures 14–15 present different examples showing the CD++ model execution of different *Worm* configurations.

Figure 14 illustrates the simulation results obtained from executing the smallest self-reproducing *Worm* configuration at different points in time. The initial configuration is depicted in Figure 14a where the *head* is highlighted and contains a **B** (state number 8) ''signal encoding'' state. As a consequence, at time $t = 1$ the control state + (state number 11) that is in charge of controlling the three-way branching process becomes the *head* of the transmission line and the self-reproduction procedure begins, as illustrated in Figure 14b. As we mentioned above, signals coming to the branching point (highlighted in Figure 14b) are copied and propagated in the directions indicated by cells with state # (state number 12) located in the

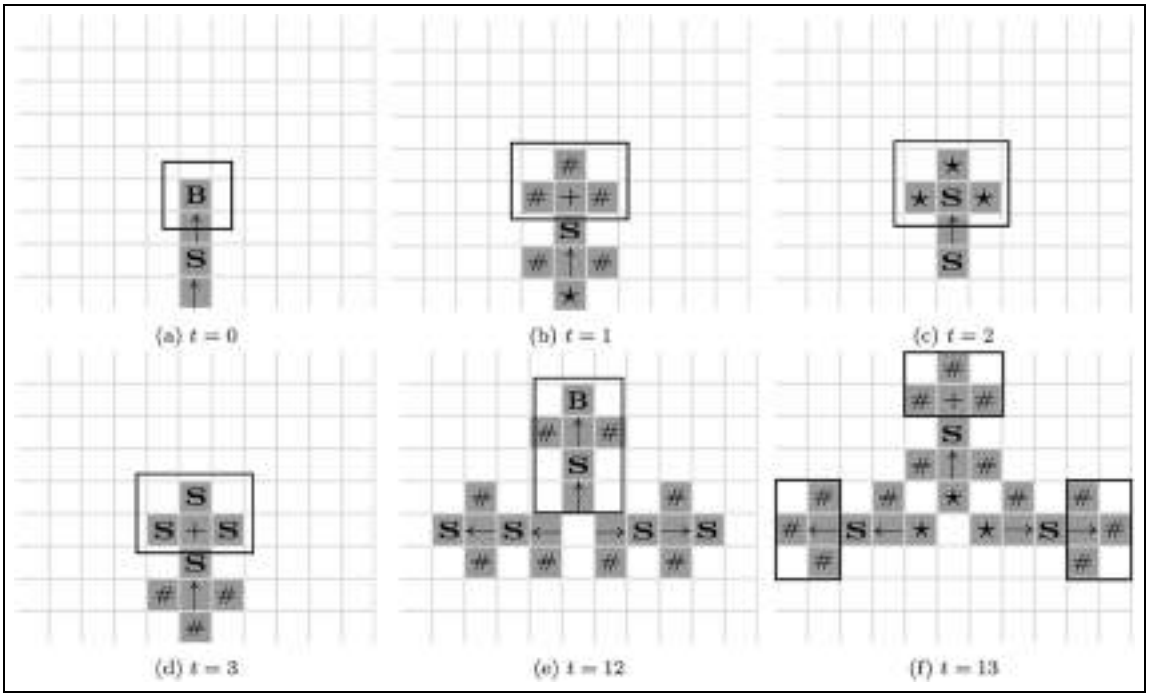

Figure 14. CDA animation results for the smallest self-reproducing *Worm* configuration at different simulation times.

neighbourhood of the *head* of the transmission line. This procedure is illustrated in Figures 14c and 14d, where we can see that at time $t = 2$ the **S** (state number 5) ''signal encoding'' state reaches the branching point and the control states ★ indicates the directions where the signal has to be propagated, then at time $t = 3$ the signal is effectively copied. This process continues until time $t = 12$ where the *tail* of the mother *Worm* is processed, this is reflected in Figure 14e where we can see the original mother *Worm* and the two children configurations that reproduce the mother shape. After this point, the two children configurations will behave like ''normal'' *Worms* (i.e., the self-reproduction property is not inherited) and the self-reproduction process of the mother *Worm* begins again as we can see in Figure 14f.

Another example of a self-reproducing *Worm* is presented in Figure 15a, the initial configuration that contains a **B**''signal encoding'' state is highlighted in Figure 15a. This ''encoding signal'' reaches the *head* of the transmission line at time $t = 7$ as depicted in Figure 15b. After this point, the three-way branching procedure mentioned above takes place which is reflected in Figures 15c and 15d. Figure 15e shows the end of the self-reproduction process at time $t = 48$. Finally, the self-reproduction cycle is repeated after 48 units of time, as we can see in Figure 15f at time $t = 96$.

So far, we have discussed the transition rules implemented for the *Worms* and simple *Loops* configurations. Now we will partially describe the configurations that need to be recognized for self-reproduction of *Loops*. In order to do this, we need to extend the transition rules, which can be easily done using the Cell-DEVS formalism.

As we mentioned above, in the self-reproduction process of *Loops* can be characterized by three sets of rules. The beginning of the reproduction itself, the encoding of the mother shape and the rules needed to generate the daughter *Loop*. For example, as part of the beginning of the self-reproduction process, the ★ encoding signal has to be generated to differentiate the beginning of the daughter ''arm'' from the mother's shape encoding itself. This rule can be implemented in CD++ as shown in Figure 16.

The first rule says that when a cell is empty and there is only one transition signal state (an arrow) pointing to the cell, then the cell must be marked as the beginning of the daughter *Loop* and the new cell state is ★. On the other hand, rules 2–5 say that if the cell state is an arrow pointing to an empty cell and there is a • in the neighbourhood, then the cell becomes the ''last'' cell of the mother and the new state is also ★.

In this way, we can extend our set of rules in a very simple manner, increasing the complexity of the modelled behaviour.

As another example, the generation of the daughter *Loop* requires that when the cell state is the • control state and the neighbourhood consist of • and a signal encoding state, then the new cell state is the ★ control state. This rule is implemented in CD++, as shown in Figure 17.

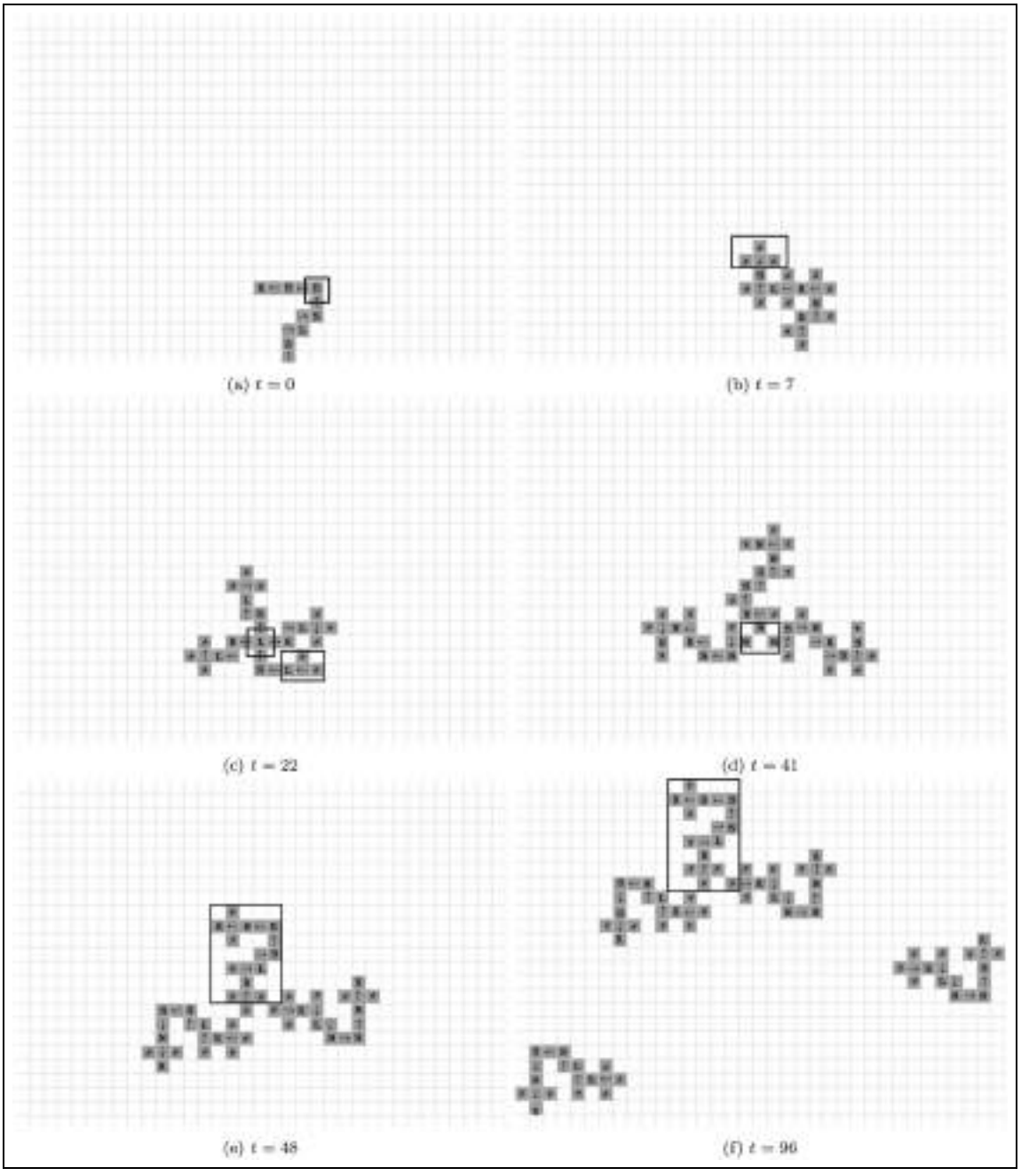

Figure 15. CDA animation for a complex self-reproducing *Worm* configuration at different simulation times.

```
rule : 9 1 {(0,0) = 0 and #macro(oneArrow) and
            (#macro(East) or #macro(West) or
            #macro(North) or #macro(South))}
rule : 9 1 {(0,0) = 2 and (0,-1) = 10 and ((-1,0) = 1
            or (1,0) = 4) and stateCount(0) = 2}
rule : 9 1 {(0,0) = 3 and (0,1) = 10 and ((-1,0) = 1
            or (1,0) = 4) and stateCount(0) = 2}
rule : 9 1 {(0,0) = 1 and (1,0) = 10 and ((0,-1) = 3
            or (0,1) = 2) and stateCount(0) = 2}
rule : 9 1 {(0,0) = 4 and (-1,0) = 10 and ((0,-1) = 3
            or (0,1) = 2) and stateCount(0) = 2}
```

Figure 16. The SR_{12} CD++ self-reproduction of *Loops* rules.

As we did for simple *Loops* and *Worms*, we present examples that shows the CD++ model execution of different *Loop* configurations.

Figure 18 shows the model execution obtained when running the implemented model in CD++ for the smallest self-reproducing *Loop*. Figure 18a shows the initial configuration at time $t = 0$, it contains a • control state (state number 10) located in the bottom left corner. The self-reproduction process begins when the • control state reaches the opposite corner at time $t = 5$ as depicted in

```
[TransmissionLine]
rule : 9 1 {(0,0) = 10 and stateCount(10) = 2 and #macro(oneCommand)}
```

Figure 17. The SR_{12} CD++ self-reproduction of *Loops* rules.

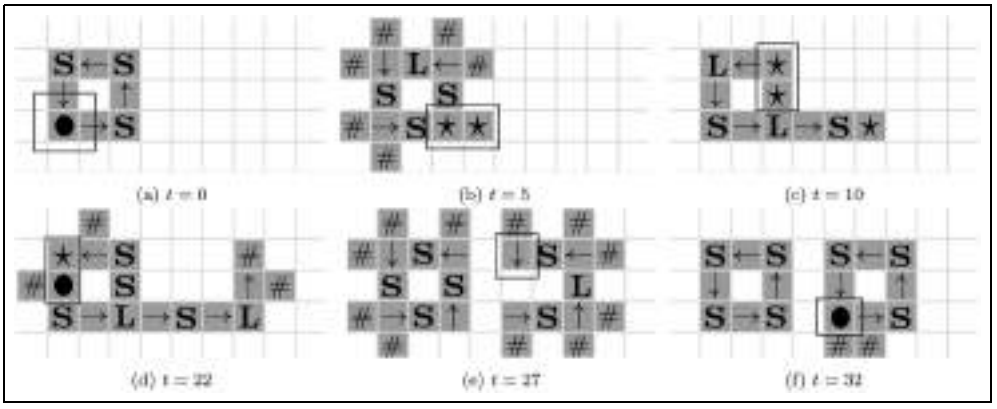

Figure 18. CDA animation for the smallest self-reproducing *Loop* configuration at different simulation times.

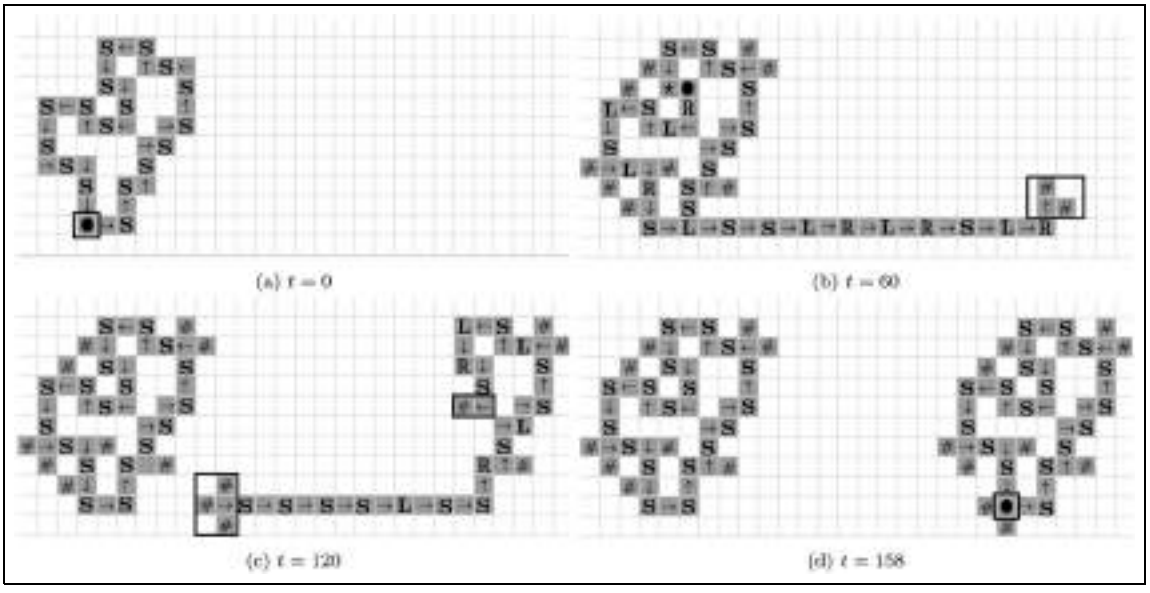

Figure 19. CDA animation for a complex self-reproducing *Loop* configuration at different simulation times.

Figure 18b, there we can see that 2 ★ control states are generated, indicating that an ''arm'' of the mother *Loop* has to be created. The control states • and ★ are in charge of handling the different configuration possibilities that appear during the self-reproduction process, as depicted in Figures 18c and 18d. As we mentioned before, when these states return back to the branching point, the ''arm'' is cut off, this is reflected in Figure 18e, after this point the daughter configuration will behave as a *Worm* until the *head* (highlighted in Figure 18e) joins the *tail*. This is reflected at time $t = 32$, when the daughter *Loop* is created, it can be noted in Figure 18f that the daughter *Loop* has inherited the self-reproduction property (i.e., a • control state in the bottom left corner of the configuration).

Finally, Figure 19 shows the model execution obtained when running the implemented model in CD++ for a complex self-reproducing *Loop*. The initial configuration is presented in Figure 19a. Then, Figures 19b and 19c shows the state of the self-reproduction procedure at two different point in time. The first figure shows the state at time $t = 60$, there we can see the control states • and ★ traveling through the mother *Loop* and signals been processed at the branching point and at the *head* of the daughter configuration (the three situations are highlighted in Figure 19b).

The second figure shows the state at time $t = 120$ where we can see that the daughter *Loop* behaves like a *Worm* with signals been processed at the *tail* and the *head* of the configuration. Figure 19d illustrates the end of the self-reproduction process at time $t = 158$ once the daughter *Loop* has been created.

6. Conclusions

We have presented the definition of three complex CA models using the Cell-DEVS paradigm using the CD++ tool to obtain executable models. These examples allow us to show the potential application of the formalism and related tools to attack different problems in a generic and systematic fashion, allowing the definition of complex transition rules for cell-shaped models.

We show how models that present complex output behaviour can be easily defined in order to study and predict the behaviour and response of systems modelled, facilitating the model development and testing for the end user.

Acknowledgements

The Rule 110 CA CD++ model was developed by Colin Timmons and the CD++ model for the three-state two-colour Turing machine was developed by Bo Feng.

Funding

Funding for this study was provided by the Canadian Network for Research and Innovation in Machining Technology, Natural Sciences and Engineering Research Council (NSERC) of Canada (10.13039/501100002790).

References

1. Wolfram S.Cellular automata as models of complexity. *Nature* 1984; 311(5985): 419–424.
2. Codd EF. *Cellular Automata*. New York: Academic Press, 2014.
3. Langton CG. Self-reproduction in cellular automata. *Physica D* 1984; 10: 135–144.
4. Weisstein EW. *Elementary cellular automaton*. Wolfram Research, Inc., 2002.
5. Wolfram S. Statistical mechanics of cellular automata. *Rev Mod Phys* 1983; 55: 601.
6. Cook M. Universality in elementary cellular automata. *Complex Syst* 2004; 15: 1–40.
7. Wainer G and Giambiasi N. Timed Cell-DEVS: modeling and simulation of cell spaces. In Sarjoughian H and Cellier F (eds), *Discrete Event Modeling and Simulation: Enabling Future Technologies*. Berlin: Springer-Verlag, 2001.
8. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. New York: Academic Press, 2000.
9. Bonaventura M, Wainer G and Castro R. A graphical modeling and simulation environment for devs. *Simulation* 2013; 89: 4–27.
10. Wainer GA. *Discrete-event modeling and simulation: a practitioner's approach*. Boca Raton, FL: CRC Press/Taylor and Francis, 2009.
11. Muzy A, Innocenti E, Aiello A, et al. Specification of discrete event models for fire spreading. *Simulation* 2005; 81: 103–117.
12. Wainer GA and Giambiasi N. Application of the Cell-DEVS paradigm for cell spaces modelling and simulation. *Simulation* 2001; 76: 22–39.
13. Wainer G. CD++: a toolkit to develop DEVS models. *Softw Practice Experience* 2002; 32: 1261–1306.
14. Weisstein EW. Turing machine. *MathWorld - A Wolfram Web Resource*, 2002. Available at: http://mathworld.wolfram.com/TuringMachine.html.
15. Morita K and Imai K. A simple self-reproducing cellular automaton with shape-encoding mechanism. In *Artificial Life V: proceedings of the fifth international workshop on the synthesis and simulation of living systems*, pp. 489–496.
16. Wolfram S. Universality and complexity in cellular automata. *Physica D* 1984; 10: 1–35.
17. Ermentrout GB and Edelstein-Keshet L. Cellular automata approaches to biological modeling. *Journal of theoretical Biology* 1993; 160(1): 97–133.
18. Moreira J and Deutsch A. Cellular automaton models of tumor development: a critical review. *Adv Complex Syst* 2002; 5: 247–267.
19. Toffoli T. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D* 1984; 10: 117–127.
20. Vichniac GY. Simulating physics with cellular automata. *Physica D* 1984; 10: 96–116.
21. Wolfram S. Computation theory of cellular automata. *Commun Math Phys* 1984; 96: 15–57.
22. Wolfram S. *A New Kind of Science*. Wolfram Media, 2002.
23. Martnez GJ, McIntosh HV and Mora J. Gliders in Rule 110. *Int J Unconvent Comput* 2006; 2(1): 1.
24. Vukašinovic SV, Stanimirovic PS, Petkovic MD, et al. Turing machine and its symbolic simulation. *Facta Universitatis (NIS) Ser Math Inform* 2009; 24: 53–72.
25. Byl J. Self-reproduction in small cellular automata. *Physica D* 1989; 34: 295–299.
26. Morita K and Imai K. Self-reproduction in a reversible cellular space. *Theoretical Computer Science* 1996; 168: 337–366.
27. Reggia JA, Armentrout SL, Chou HH, et al. Simple systems that exhibit self-directed replication. *Science* 1993; 259: 1282–1287.

Author biographies

Gabriel A Wainer (SMIEEE, SMSCS) received a PhD (1998, with highest honors) at the University of Marseille, France. In July 2000, he joined the Department of Systems and Computer Engineering at Carleton University where he is a full professor. He has authored 3 books and over 300 research articles; he edited 4 other books and helped organizing a large number of conferences, being one of the founders of SimAUD, SIMUTools and TMS-DEVS. He was Vice-President Conferences and

Vice-President Publications of SCS (Society for Modeling and Simulation International), where he is a current member of the Board of Directors. He is the Special Issues Editor of *SIMULATION*, member of the Editorial Board of *IEEE Computational Science and Engineering, Wireless Networks* (Elsevier) and *Journal of Defense Modeling and Simulation* (SCS). He received the IBM Eclipse Innovation Award, the SCS Leadership Award, and various Best Paper awards; also, the First Bernard P. Zeigler DEVS Modeling and Simulation Award (2010), the SCS Outstanding Professional Award (2011), Carleton University's Mentorship Award, the SCS Distinguished Professional Achievement Award (2013) and Carleton University's Research Achievement Award (2005 and 2014).

Joaquín Fernández received his BS degree in computer science in 2012 from the Universidad Nacional de Rosario, Argentina. He is currently a PhD student at the French Argentine International Center for Information and Systems Sciences (CIFASIS). His research interests include hybrid system simulation, real-time and parallel simulation, simulation tools and modelling languages.