Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
1–27
© The Author(s) 2018
DOI: 10.1177/0037549718761596
journals.sagepub.com/home/sim
SAGE SCS

Advanced Cell-DEVS modeling applications: a legacy of Norbert Giambiasi

Gabriel A Wainer

Abstract
We describe a number of modeling applications of advanced Cell-DEVS models, a modeling formalism Norbert Giambiasi and I defined in the late 1990s. We discuss improved versions of these models built using the CD ++ toolkit, which was built in order to study, model, and simulate such cellular models. The models have removed some limitations that standard cellular models have, which allow each cell to use multiple state variables and multiple ports for inter-cell communications. We show the application of the formalism in three different areas of science and engineering: social models, pedestrian analysis and occupancy in buildings, and virus spreading in computer networks.

Keyword
Cell-DEVS, Cellular Models, DEVS, Discrete-Event Modeling and Simulation

1. Introduction

In 1995, under the supervision of Norbert Giambiasi, we started working on new modeling simulation techniques to study complex systems. The idea was to help understand the dynamic behavior of real systems where analytic solutions are impossible to find. We were initially interested in studying a combination of parallel simulation algorithms and formal models that could be executed by simulation engines. Formal models can improve the definition of the model and making easier their execution, as these executable models can be verified against a formal specification. A parallel simulator could run these executable models with high performance. Such models, built using a modeling formalism, could also be used to translate formal specifications into executable models, making the modeling process simpler. Even though it contains many assumptions, an executable model is far less ambiguous than a text-based requirements document.

One such formalism, known as DEVS (discrete events systems specification), was well suited to achieve our goals.[1] DEVS formal definitions allow modular description of the phenomena to model, and it attacks complexity using a hierarchical approach. This hierarchical modeling strategy allows the reuse of tested models, enhancing the security of the simulations, reducing the testing time and improving productivity. During our initial research on parallel simulation of DEVS models, we found an interest in extending these concepts to cellular models, based on the original formal specifications of cellular automata (CA). CA is a formalism well suited to describe complex systems with different description levels, using a spatial notation that is useful to understand the results of the simulations, and analyzing their results.[2] A CA is a formal model defined as an infinite regular n-dimensional lattice containing cells that can execute a local computing function. Each cell state is discrete and is modified in discrete time steps. To do so, it uses the local transition function based on the present cell state and a finite set of nearby cells (called the cell's neighborhood).

One of the issues we wanted to address was the fact that CA use discrete time, and DEVS is discrete-event. This is a major difference when we build executable cell spaces. DEVS timing representation is more general, and CA time-stepped definition poses restrictions in the precision of the model. Discrete time bases are more imprecise, as the time is represented as a positive integer number. The use of a continuous time base allows higher time precision

Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada

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

to be achieved, periods of inactivity are skipped, and the use of the computer resources is improved. Likewise, using a discrete time base makes it difficult to integrate phenomena with different timing delays. Finally, combining CA with other models is complex. Considering these issues, we extended the original definition of cellular models and defined Cell-DEVS, a timed cellular model specification based on DEVS with explicit timing delays. Based on Norbert's experience in the circuit design domain, we added semantics for transport and inertial delays, two basic constructions usually employed in circuit design. Norbert had a long research experience in the domain of digital circuits, and his experience was focused on this part of the formalism. Transport delays reflect the straightforward propagation of signals over lines of infinite bandwidth. Inertial delays, instead, allow defining preemptive semantics in the models. These constructions are very useful to define advanced timing delays in the cells, and we showed in different case studies how to use them to model other physical phenomena that can be described as cell spaces (i.e., fire propagation, environmental models, urban traffic, crowd modeling, etc.). The original ideas were presented elsewhere.[3–6]

It is important to notice that each of the cells in the cell space is updated at different times (a real number, or extended representations like the ones defined in Goldstein et al.),[7] as set by a rule's delay component. This is a clear departure from the classical approach to CA, where all active cells are updated at the same time, thus reducing computation time. Another advantage is that expressing timing delays is simple, which allows the modeler to reduce the development time related with timing control modeling. The specification of such a cellular model can be translated into an executable model.

The Cell-DEVS formalism has been used for modeling and simulation of varied phenomena,[8–11] based on our work with Norbert and our original discussions. This non-comprehensive list can be classified in various areas: traffic,[12–16] physics,[17–19] environmental modeling,[19–25] networking,[26–29] architecture and construction,[30–34] biology and medicine,[35–39] defense and emergency planning,[40–48] social models,[48–50] crowd modeling,[51–54] and many others, ranging from power models, music generation, image detection, etc.[55–57]

Cell-DEVS was extended and combined with different methods to improve the modeling further. One of the first contributions in the year 2000 was the definition of *activity-based models*, in which we proposed to use the level of activity of the different cells in the model to reduce the computation required to run the Cell-DEVS models. This idea, based on the dynamic quantization of the cell's value, was presented in the paper by Wainer and Zeigler and recently detailed by Wainer.[58,59] By conducting varied experimentation, we could realize that when we use a quantizer for a cell, and we reduce the outputs generated by each of them, we gain speed while having a cost of added error. We saw that when we increased the quantum size for the active cells, they became inactive quickly and reduced computation time (at increased error rate). Thus, detecting the activity level in the different zones of the cell space would improve the performance of the simulation even further. Then, we combined Cell-DEVS was combined with G-DEVS, another contribution of Norbert Giambiasi, in which we approximate the local computation function using a polynomial approximation of the local computing function.[60] We also combined with quantized DEVS models with hysteresis, which later became formalized and extended as the set of quantized state system numerical methods.[61] Other extensions included extending Cell-DEVS for quantum dot computing, computational fluid dynamics,[62] and finite element and finite differences methods.[63]

The Cell-DEVS formal specifications allowed us to define a variety of simulation algorithms in different environments, which include parallel simulation using conservative approaches,[64,65] optimistic algorithms, multi-core architectures, and using multi-coarse-grained parallelism.[66–69] All the models can also be executed in distributed fashion using a variety of distributed middleware algorithms.[70–72] The executable specification and the independence between simulation algorithms and model specification provided by DEVS allowed us to define these multiple simulation engines with ease.

The formal specifications for Cell-DEVS models were defined by different teams of students throughout the years, based on the formal specifications built based on our discussions with Norbert. The tools are different instantiations of the CD ++ toolkit,[7] which was originally built to simulate DEVS and Cell-DEVS models. CD ++ was originally built including a high level modeling language for binary CA combined with timing delays. It was then extended to include an advanced modeling language, which was further extended to enhance expressivity for defining complex applications.[73,74] All these advances have been summarized in Figure 1.

In the following sections we will discuss the basic ideas of Cell-DEVS, introduce the CD ++ modeling language for modeling Cell-DEVS applications, and will present different case studies that are useful to understand the formalism better. We will first present a simple example of a social model in which we study the influence of peers for choosing products in a duopolistic markets. We then focus on a complete different area: modeling of pedestrian movement and occupation of buildings, and the integration of the cellular model with three-dimensional (3D) visualization engines. Finally, we show a model of malware in computer networks, showing how Cell-DEVS models can be used to study this modern security problem. The models are based on existing literature, in which the authors have conducted validation of the correctness of the models

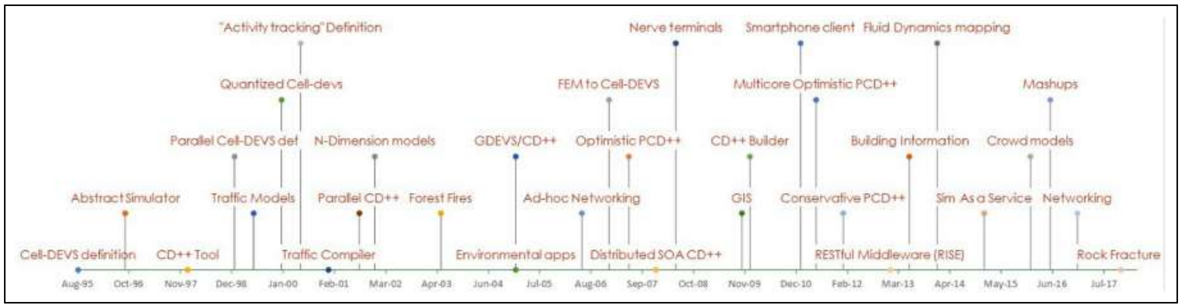

Figure 1. Timeline: definition of Cell-DEVS specifications, algorithms, tools, and major applications.

proposed. Here we show how to define those models and some basic extensions as Cell-DEVS models, and their implementation in CD ++ , as a way to show concrete results of the theoretical work we conducted with Dr. Giambiasi in three different areas of knowledge.

2. Formal description of Cell-DEVS models (revisited)

As discussed in the Introduction, Cell-DEVS was defined as a combination of CA and DEVS with explicit timing delays. The DEVS formalism provides a framework for the construction of modular hierarchical models, allowing for model reuse, and for reducing development time and testing. In DEVS, basic models (called *atomic*) are specified as black boxes, and several DEVS models can be integrated together forming a hierarchical structural model (called *coupled*). DEVS not only proposes a framework for model construction, but also defines an abstract simulation mechanism that is independent of the model itself. Cell-DEVS define a cell as a DEVS model and a cellular automaton as a coupled model, and introduces an explicit timing mechanism for each cell.

Using Cell-DEVS has different advantages. First, we have asynchronous model execution, which, as shown in our work with Giambiasi,[3,4] results in improved simulation execution times. Timing constructions permit defining complex conditions for the cells in a simple fashion, as showed in also early work done on the topic with Norbert.[5,6] As DEVS models are closed under coupling, seamless integration with other types of models in different formalisms is possible. The independent simulation engines permit these models to be executed interchangeably in single-processor, parallel or real-time simulators.

Cell-DEVS allows building cellular models in which each cell holds a state variable and a computing apparatus. This function is in charge of updating the cell state according to a local rule, by using the present cell state and those in a finite set of nearby cells (called its neighborhood). Each cell is defined as a DEVS atomic model, and it can be later integrated to a coupled model representing the cell space. A Cell-DEVS atomic model is defined as follows:

$$\mathrm{TDC} = < X, Y, I, S, E, \mathrm{delay}, \mathrm{d}, \delta_{\mathrm{int}}, \delta_{\mathrm{ext}}, \tau, \lambda, \mathrm{ta} >$$

- $X \in \boldsymbol{R}$ is a set of external input events;
- $Y \in \boldsymbol{R}$ is a set of external output events;
- I is the model's modular interface; $I = < \eta + \gamma, \mathrm{P}^x, \mathrm{P}^y >$ defines the model interface. Here, $\eta, \gamma \in \boldsymbol{N}$ are the number of inputs coming from the neighborhood size and other inputs, and, for $I = X$ or $I = Y$, P^I is a port definition (input or output, respectively), where $\mathrm{P}^I = \{(\mathrm{N}_i^I, \mathrm{Ti}^I) \mid \forall\, i \in [1, \eta + \gamma], \mathrm{N}_i^I \in [\mathrm{I}_1, \mathrm{I}_\eta]$ (port name), and T_i^I =binary (port type)$\}$;
- S is the state set, where $S = \{(\mathrm{s}, \sigma \text{ queue}) \mid \mathrm{s} \in \boldsymbol{R}$, σqueue $=$ σqueue$=\{((\mathrm{v}_1, \sigma_1),\ldots,(\mathrm{v}_\mathrm{m}, \sigma_\mathrm{m}))/\mathrm{m} \in \boldsymbol{N} \wedge \mathrm{m} < \infty \wedge \forall\, (\mathrm{i} \in \boldsymbol{N}, \mathrm{i} \in [1,\mathrm{m}]), \mathrm{v}_\mathrm{i} \in \boldsymbol{R} \wedge \sigma_\mathrm{i} \in \boldsymbol{R}_0^+ \cup \infty\}\}$; for transport delays.

The formal specification of a cell with inertial delays only changes the state definition:

$$\mathrm{S} = \{ (\mathrm{s}, \mathrm{f}, \sigma) / \mathrm{s}, \mathrm{f} \in \boldsymbol{R}, \text{ and } \sigma \in \boldsymbol{R}_0{}^+ \cup \infty \};$$

- E is the set of values for the input external events; $E \in R^{\eta + \gamma}$, is the set of the external events (from the neighborhood's and other inputs);
- delay is the type of delay: transport, inertial, or others defined;[7]
- $\mathrm{d} \in \boldsymbol{R}_0{}^+$ is the delay value for the cell;
- δ_{int}: S→S is the internal transition function;
- δ_{ext}: QxX → S is the external transition function, where Q is the state set defined by Q = $\{ (\mathrm{s}, \mathrm{e}) / \mathrm{s} \in \mathrm{S}$, and $\mathrm{e} \in [0, \mathrm{ta}(\mathrm{s})]\}$;
- τ: E → S is the local transition function;
- λ: S →Y is the output function; and
- ta: S $\rightarrow \boldsymbol{R}_0{}^+ \cup \infty$, is the time advance function.

A cell uses a set of input values E 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 neighbor cells. Two types of delays were defined originally, which are used in most applications: inertial and transport delays (although there are other variations).[7] When a transport delay is used, the future value will be added to a queue sorted by output time. Therefore, all the previous values that were scheduled for output but that have not yet been transmitted; will be kept on the queue. On the other hand, 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 δ_{ext} function.

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

$$GCC = \, < Xlist, \, Ylist, \, I, \, X, \, Y, \, n, \, \{t_1, ..., t_n\}, \, N, \, C, \, B, \, Z >$$

- *Ylist* = {(k,l) / k ∈ [0,m], l ∈ [0,n]} is the list of output coupling;
- *Xlist* = {(k,l) / k ∈ [0,m], l ∈ [0,n]} is the list of input coupling;
- $I = \, < \boldsymbol{\eta}, \boldsymbol{\gamma}^x, \boldsymbol{\gamma}^y, \boldsymbol{p}^x, \boldsymbol{p}^y >$ represents the definition of the interface for the modular model whose size is $\eta \in N, \eta < \infty$; $\boldsymbol{p}^x$ is the set of all input ports ($\boldsymbol{\eta}$ neighbor ports + $\boldsymbol{\gamma}^x$ external ports) and $\boldsymbol{p}^y$ is the set of all output ports ($\boldsymbol{\eta}$ neighbor ports + $\boldsymbol{\gamma}^y$ external ports);
- $X \in \boldsymbol{R}$ is a set of external input events;
- $Y \in \boldsymbol{R}$ is a set of external output events;
- $n \in \boldsymbol{N}$ is the dimension of the cell space;
- $\{t_1,...,t_n\}$ with t_i $i \in \boldsymbol{N} \forall$ i ∈ [1, *n*] is the number of cells in each of the dimensions;
- N ={ (i, j) / i, j ∈ $\boldsymbol{Z}$ i, j < ∞ } is the neighborhood set;
- C ={ $C_{k1,...,kn}$) / $k_i \in$ [0, t_i]} is the cell space;
- B ⊆ C ∪ {∅} is the set of border cells; and
- Z: P_{ij}Yq → $P_{kl}{}^{X}$q, where $P_{ij}{}^{Y}$q ∈ I_{ij}, $P_{kl}{}^{X}$q ∈ I_{kl}, q ∈[0, η] and, ∀ (f, g) ∈ N, k = (i + f) mod m; l=(j + g) mod n;
- Zij: $Y(f,g)_i \rightarrow X(k,l)_j \forall (f,g)$ $Ylist_i$, and (k,l) $Xlist_j$.; and
- select, is the tie-breaking selector function, with the restriction that select ⊆ mxn → / ∀ E ≠ {∅}, select(E) ∈ E.

This specification defines a coupled model composed of an array of atomic cells. Each cell is connected to the cells defined in the neighborhood, but as the cell space is finite, either the borders are provided with a different neighborhood 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. This function translates the outputs of the *m*th output port in cell C_{ij} into values for the *m*th input port of cell C_{kl}. Each output port will correspond to one neighbor and each input

```
δext((s, σqueue), e, x) = (s', σqueue'), where
        Insert x in the E entry corresponding to the input port;
        s' = τ(E);
        for each ((σqueue).$d_i$ ∈ σqueue) $\sigma_i = \sigma_i - e$;  /* Update the time of delays in the queue */
        if (s = s') / * quiescent cell */
             if σqueue is empty then ta(s) = ∞; /*  passivate */
        else
              ta(s) = d /* Cell's transport Delay */ ;
              σqueue = insert(σqueue, (s', ta(s)));

λ (s, σqueue) = (s', σqueue), where
        out = first(σqueue.v);
        send (out, output port);

δint(s, σqueue) = (s', σqueue), where
        if σqueue is empty then ta(s) = ∞; /*  passivate */
        else ta(s) = first(σqueue.σ) /* Time until next end of Delay */ ;
        σqueue = tail (σqueue);   /* Clean up queue */
```

Figure 2. Definition of transition functions for cells with transport delays.

port will be associated with one cell in the inverse neighborhood.

The transport delay model allows introducing a delay between the occurrence of an external transition function and the state change of the cell. Only when the transport delay is consumed, the internal transition function is executed and the system changes its state. The σqueue is introduced because new external events can occur while the transport delay is consumed. These must be recorded and later executed by the internal transition functions.

In this definition, presented in Figure 2, *insert, first, tail* and *empty* are the traditional functions employed to manage a FIFO queue. The external transition function schedules a new time for an internal transition function. To do so, it uses the value of the transport delay. The local transition function is executed using the new input values stored in E. Then, we update the times of the waiting events in the transport delay queue, and update the state of the cell. This is only done only if there was a change; otherwise, the cell passivates. The output function is activated at the next time for an internal event, and it generates an output based on the first element in the queue. The internal transition function then cleans up the queue and schedule the next internal event based on the queue information.

The inertial delay constructions allow a behavior to be represented with a preemptive semantic. The construction says that, if an input value is not kept a certain period (the inertial delay), the state change is not recorded. Instead, if the value is kept during that time, the state changes after the delay. To model this kind of delays, the transition functions are different.

The last arrived event can be preempted if a new external event (with different value) arrives before the end of the inertial delay. If a new external event has the same value of the old one, the result is equivalent to have a unique external event. This is described in Figure 3.

If an event occurs in one cell, the neighbors are influenced through the execution of the Z function. Besides, certain cells in the space can be chosen as input and output cells, and they will be included in the *Xlist* and the *Ylist*, respectively. *Xlist* is a list of cell's positions where the inputs to the model are received. *Ylist* records the cells whose outputs will be sent to the other models in the hierarchy.

When a Cell-DEVS model is executed, the Z_{ij} function translates inputs into outputs by using both lists. The names of the input and output ports are also defined by using the contents of the *Xlist* and *Ylist.*

The specification models here presented are independent of the simulation technique used. Therefore, they

```
δext(((s, f, σ),  e, x) = ((s, f, σ)'), where
        Insert x in the E entry corresponding to the input port;
        f = τ(E);
        if (s = f)  /* no change */
                σ = σ - e; }  /* Adjust time until next internal event */
        else    /* preemption */
                σ = d;  /* Inertial Delay */
                s = f;

λ (s) = out, where
        out = s;
        send (out, output port);

δint (((s, f, σ),  e, x) = ((s, f, σ)'), where
        ta(s) = ∞;
```

Figure 3. External and internal transition function for inertial delays models.

allow specifying the system behavior independently of the implementation details of the chosen simulation technique.

3. The CD++ toolkit

CD++ is a tool built to implement DEVS and Cell-DEVS models.[7] There are numerous tools, and some recent efforts include DesignDEVS,[75] DEVS-SOA,[76] PythonDEVS,[77] and others.[78–80] The tool allows models to be defined according to the specifications introduced in the previous section. DEVS atomic models can be incorporated into a class hierarchy in C++. Coupled and Cell-DEVS models are defined using a specification language specially defined with this purpose, following DEVS and Cell-DEVS formal definitions. The tool includes an interpreter for a specification language that allows describing the behavior of each cell, including the local computing function and delay. In addition, it allows defining the coupled model, including size of the cell space and its connection with other DEVS models, the border and the initial state of each cell.

The behavior specification of a cell is defined using a set of rules, each indicating the future value for the cell's state if a precondition is satisfied. The local computing function evaluates the first rule, and if the precondition does not hold, the following rules are evaluated until one of them is satisfied or there are no more rules. Figure 4 shows an example for the specification of a Cell-DEVS model developed using CD++. The specification follows the Cell-DEVS coupled model's formal definitions introduced in the previous section. In this case, *Xlist* = *Ylist* = $\{\emptyset\}$. Here, the dimension $n = 2$, therefore the set $\{t_1, t_2\}$ is defined by the keywords *width-height*, which specifies the size of a two-dimensional (2D) cell space (in this example, $t_1 = 20$, $t_2 = 40$). The N set is defined by the sentence *neighbors*. The border (B) is wrapped. Using this information, CD++ builds an executable cell space, defines the I/O ports, and the Z translation function following Cell-DEVS specifications.

The behavior of these rules, which define the local transition function, is defined using a set of rules in which there is a precondition to the right, a postcondition to the left, and a delay value between them. When the precondition is satisfied, the new value of the cell should change the postcondition value. The output of such value should be delayed using a transport, inertial, or other delay for the specified time. The tool's main operators available to define rules include: Boolean, comparison, arithmetic, neighborhood values, time, conditionals, angle conversion, pseudo-random numbers, error rounding and constants (i.e., gravitation, acceleration, light, Planck, etc.). In the example, the local computing function executes very simple rules. The first one indicates that, whenever a cell state is 1 and the sum of the state values received in the input set E is 7, the cell state changes to 0. This state change will be spread to the neighboring cells after 200 ms. The second rule states that, whenever a cell state is 0 and the sum of the inputs whose value is 0 is smaller than 4, the cell value changes to 1 and the output is sent after 300 ms. In any other case (t = true), the result remains unchanged, and it will be spread to the neighbors after 110 ms. As we can see, cells evolve using a discrete-event approach.

The local computing function scans the specification, verifying the logical expressions included and computing the new state value for the cell. Several errors of the specification can be found at runtime, allowing the detection of inconsistencies in the model definition:

- Ambiguous models: a cell with the same precondition can produce different results;
- Incomplete models: no result exists for a certain precondition;
- Non-deterministic models: different preconditions are satisfied simultaneously. If they produce the same result, the simulation can continue, but the modeler is notified. Instead, if different results are found, the

```
[ex]
width : 20     height : 40    border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1)  (0,0)  (0,1) (1,-1)  (1,0)  (1,1)
localtransition : tau-function

[tau-function]
rule: 0 200 {(0,0)=1 and (truecount=7)}
rule: 1 300 {(0,0) = 0 and false <= 4 }
rule: (0,0) 110 { t }
```

Figure 4. A Cell-DEVS specification in CD++.

simulation should stop because the future state of the cell cannot be determined.

CD ++ extended the concept of "one state variable per cell" defined by CA, and also includes the means to define ports to/from the neighbors, which are used to send/receive values from one cell to another in a coupled cell model. Using this idea, we need to declare state variables, which is done as follows (once declared, the state variables can be referenced in the rules). The first line declares the list of state variables that can be used by every cell. The second line declares the default initial values for these states variables.[74]

```
StateVariables: pend temp vol
StateValues: 3.4 22 -5.2
```

The basic grammar for the rule is as follows:

```
[ <port_assigns>] <value>
[ <assignments> ] <delay>
 <precondition>
```

The precondition is a set of expressions that, if met, will result in the postcondition. This is a mix of three components: assigning values to output ports (optional, if there are any), to state variables (if any), and changing the value of the current cell's main state variable.[74] A variable is referenced by the name declared in *StateVariables* sentence, preceded by a **$**, from any part of a rule, for instance:

```
rule:{ (0,0,0) + 1} { $temp:=$vol/2;
$pend:=(0,1,0);} 10{ (0,1,0) > 5.5}
```

In the example, we are not using the optional port assignment section. Here, if the condition (0,1,0) > 5.5 is true, the variable *temp* will be assigned half of *vol* value, *pend* will be assigned the value of the neighbor cell (0,1,0), and *vol*'s value will remain unmodified. The new value will be the one the cell holds plus one, and this value will be transmitted after 10 time units. The identifier '**:=**' is used to assign values to a state variable. Assignments can be placed in an expression within the rules (enclosed between curly brackets). A list of assignments can be defined, separated by semicolons.[74]

We can use multiple I/O ports to communicate with the neighbors (besides a default port that transmits the cell's value). They are defined as a list of neighbor port names as follows:

```
NeighborPorts: alarm weight number
```

The input and output neighbor ports share names, making possible to calculate automatically the influences: an output port from a cell will influence exclusively the input port with the same name in every cell in its neighborhood. In the example, we define three ports (*alarm, weight*, and *number*). When a cell outputs a value through one of these ports, it will be received by all its neighbor cells through their input ports with the same name.[74] A cell can read the value sent by one of its neighbors, specifying the input port. Both the cell and port must be specified, separated by a tilde (**~**):

```
rule : 2 300{ (1,0)~weight> 20}
```

In this case, if the cell receives an input in the *weight* port from the cell to the left and that value is larger than 20, the cell state will change to 2, and this change will be transmitted through the default output port 300 time units after that. As one might need to output values through many ports at the same time, the assignment can be used as many times as needed (each followed by a semicolon), as follows:

```
rule:{ ~alarm := 1; ~weight := (0,-1)~weight;
} 100{ (0,1)~number > 50}
```

In this example, if we receive a value larger than 50 from the port *number* in the cell to the right, we will wait 100 time units, and we will generate an output of 1 in the *alarm* port, and we will copy the *weight* value received from the cell to the left into the *weight* output port.[74]

The rules defining the models coupling and those related to the behavior of a cell should be translated into an executable definition. To do so, the rules specifications are associated with a function's identifier, which is registered by each cell, and each one of the rules is represented with a tuple (value, delay, condition) represented by a tree. In order to evaluate a rule, we evaluate the tree that represents the condition recursively. If the result of the evaluation is *True*, we evaluate the trees corresponding to the value and the delay, and the result of these evaluations are the values used by the cell. To do so, we built a lexical analyzer for the new language, whose grammar can be found in the Appendix. The <port_assignments> produces a sequence of output operations triggered by the output function.[74]

```
rule:{ ~alarm := 0;send(alert, 1);} 100
{portref(alert)=0 and ~alarm!= 0}
```

4. Simulation mechanism

The software architecture of CD ++ is based on the algorithms presented elsewhere,[1,4,5,8] which use the architecture in Figure 5. *Models* is the basic abstract class, from which all the models are subclasses. It is responsible for managing all the input and output ports, knowing when the next event is scheduled, and knowing its identifier and

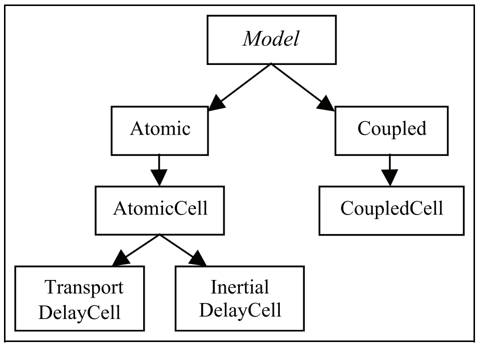

Figure 5. Model hierarchy.

its parent model. *Atomic* is an abstract specialization of the *model* class that represents the interface of an atomic model. In addition to all the responsibilities inherited from *model*, it also provides the interfaces for the initialization function, the internal and external transition functions, the output function, and the time advance function. *Atomic Cell*, which is focused on Cell-DEVS models specifically, is a specialization of *Atomic* that provides the interfaces for the cells of a cellular model. Its *responsibilities* are to knowing the local computation function, the cell's neighborhood, the available ports, and the cell's value. When an instance of a non-abstract subclass is created, this class will take care of notifying the neighbor cells the cell's initial value (using default ports called *neighborChange* for inputs and *out* for outputs; the other ports are created dynamically and they stored in two lists named *in* and *output*).

As described in Zeigler et al.'s *Theory of modeling and simulation*,[1] the simulation mechanism is driven by messages passing between processing entities (*processors*). This simulation mechanism has been described in the literature and will not be repeated in this section.[1,8] In order to implement the simulation algorithms, we use different message types carrying information specific to the type of event they represent (input, output, internal transition, etc.). The message passing mechanism is encapsulated, thus the message distribution policy can be changed without affecting the rest of the software modules.

Figure 6 shows the root abstract class for all messages. It is responsible for knowing the time of the message and its sender. Here, *InitMessage* is a subclass of *Message* representing the information that the processors receive when the simulation begins. *InternalMessage* tells the destination processor that the time for an internal event has arrived. *ExternalMessage* is used to transmit the information carried by external events (*X* set). In addition to the information provided by *Message*, this class includes the port of arrival, and its value. *DoneMessage* sent from a child processor to its parent indicating the time for the child's next scheduled event (which is defined by programming the time advance function *ta* as discussed earlier). *OutputMessage* represents the output messages (*Y* set). In addition to the information provided by its superclass, it includes the output port, and its value.

As we discussed in the previous section, the rules used for defining the models coupling and the cell's behavior should be translated into an executable definition. We will now discuss how the rules for cellular models are evaluated (and the expressions used for defining coupled models are handled similarly). The rules specifications are associated with a function's identifier, which is registered by each cell. When the models to be simulated are loaded, if the definition of a transition function is not is registered, then it is added to a table that includes the function name (identifier), and each of the rules represented as a tuple (postcondition, delay, precondition). Each element of the tuple is represented as a tree. The following class hierarchy

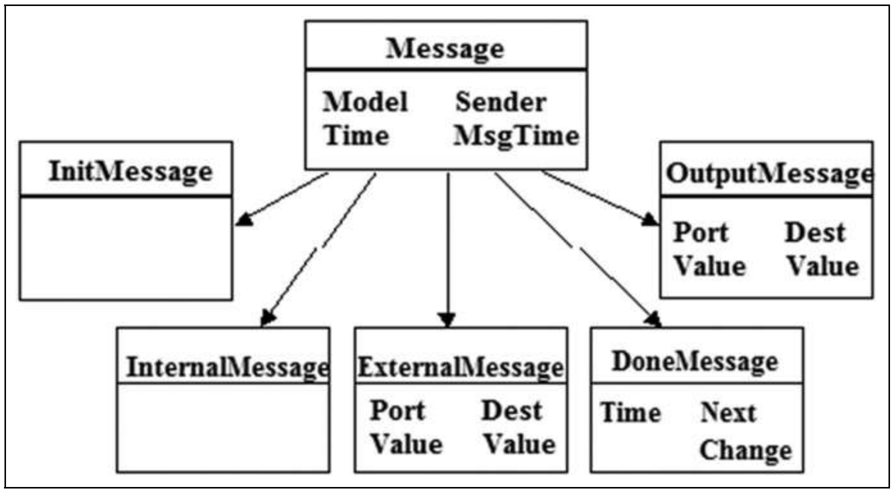

Figure 6. Message hierarchy.

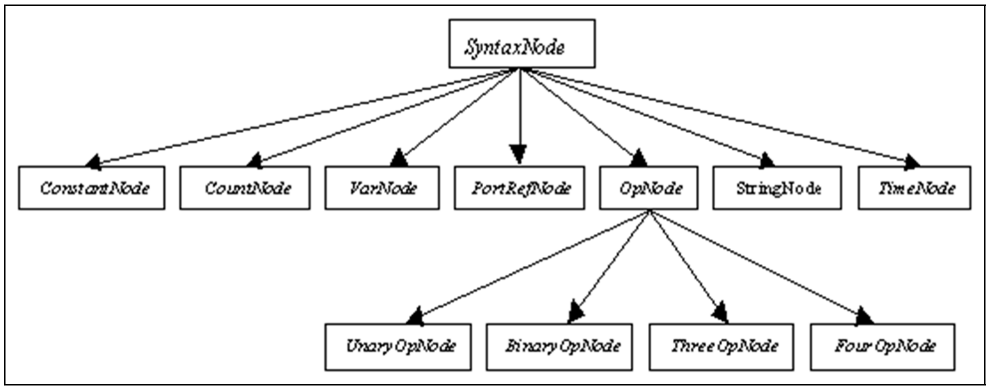

Figure 7. Subclasses of a *SyntaxNode*.

is devoted to build such rule definition tree presented in Figure 7.

SyntaxNode is an abstract class that allows describing a node in the rule evaluation tree. It is composed by the following subclasses:

- *ConstantNode*: it is used to store a constant with domain in the Reals, Integers, or three-valued logic.
- *CountNode*: it allows defining the functions TrueCount, FalseCount, and UndefCount.
- *VarNode*: it stores a reference to a neighbor cell, defined as an offset from the present cell.
- *PortDefNode*: it defines a reference to an input port of the cell.
- *StringNode*: it contains a string of characters representing the name of an input port of the cell. It is used to evaluate the *PortValue* function.
- *TimeNode*: it allows defining the *Time* function, which returns the present simulated time.
- *OpNode*: it is an abstract class representing functions with one or more parameters. It has the subclasses: *UnaryOpNode, BinaryOpNode, ThreeOpNode*, and *FourOpNode*, which represent the functions with 1–4 parameters, respectively.

Each rule defining the behavior of a cell can be represented by a tree structure. For example, let us assume we want to represent the following rule extracted from the *Life Game* Cell-DEVS model:[8]

```
rule : 1 10 { (0,0) = 1 and (truecount = 3) or
(truecount = 4) }
```

This rule says that whenever there is a cell with value of 1, and 3 or 4 neighbors with value 1, the cell's next state is still 1. This information is transmitted to the neighbors after 10 time units. Internally, such a rule is represented as in Figure 8.

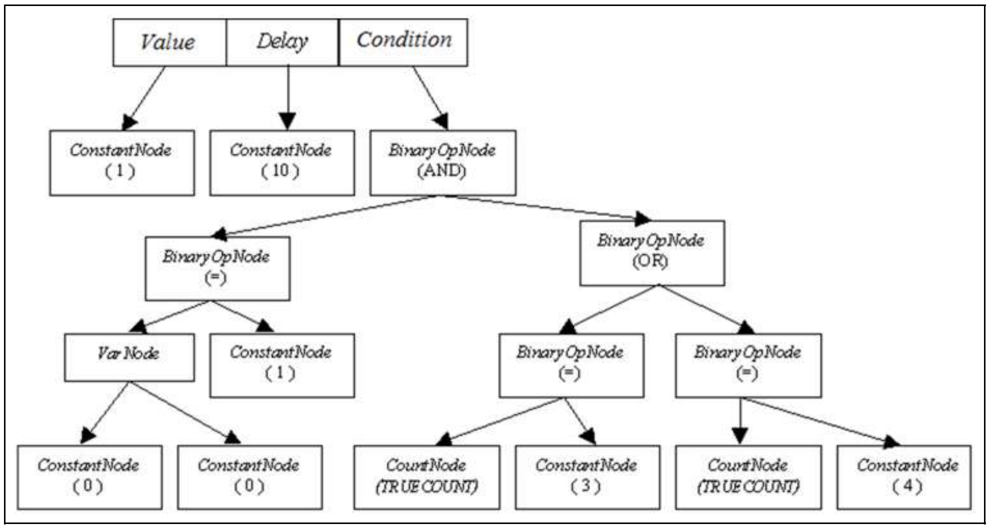

Figure 8. Tree structure used to represent the rule above.

After the rule tree is defined, every time we need to evaluate it, the tree that represents the condition is parsed recursively. If the result of the evaluation is *True*, it proceeded to evaluate the trees corresponding to the value and the delay, and the result of these evaluations is the values used by the cell.

We have now defined Cell-DEVS models formally, then we have discussed the implementation of Cell-DEVS models in the CD ++ tool, and we will now show different application examples, focusing on the use of the formalism in different domains. We have chosen three different areas. We will first show a social model in which we use a cellular model implemented in Cell-DEVS to study the influence of consumers in a duopolistic market based on consumer preference and their influence to their peers. We then show the application of Cell-DEVS to human movement in buildings and the application of this model to study occupancy in buildings and integration with 3D tools. The objective of this example, which has been presented earlier in related research, is to show a different version of the model in which we can see how to build a four-dimensional (4D) model in Cell-DEVS, showing a higher dimensional model, and discussing advantages and issues with such approach. Finally, we focus on a model with application to information technology. In this case, we discuss a model of malware spreading in computer networks. This model shows the application of an advanced model using varied state variables and input/output ports on each of the cells.

5. A model of consumer influence in a duopolistic market

As discussed earlier, in this section we show a Cell-DEVS model representing the influence of consumers in making their choices for a given product in a duopolistic market. Consumers choose among competing products, and their purchasing decisions are normally made not only based on their own preferences but also under the influence by others. The model is based on that proposed by Martín del Rey,[82] in which the author discusses a cellular model in which three factors contribute to consumers' purchasing behavior: the utility obtained, the effect of network externality (i.e., the influence exerted by peers), and price. Based on the original article's ideas, we here show how to build a Cell-DEVS version of such model in CD ++ , in which each cell in the space represents an individual who will choose between products in a duopolistic market.

There are two choices of products (representing, for instance, two types of cell phones or internet service providers). Each cell has three possible states: it has one instance of Product A, of Product B, or none (denoted as 1, 2, or 0, respectively). We use the following notation to describe a cell's state variable: $X(c_{ij}, n, t)$, with n = 1 or 2 for cell c_{ij}, which represents a user of product n at time t. $X(c_{ij}, n, t) = 0$ if the cell c_{ij} does not have a user. There is only one possible product for each cell. The values of the contributing factors to make a decision are computed as follows:

$$V(c_{ij}, n, t) = U(c_{ij}, n, t) + E(c_{ij}, n, t) - P(c_{ij}, n, t)$$

for all the three possible states, and choose the state that maximizes $V(c_{ij}, n, t)$ as its next state. Here:

$$U(c_{ij}, n, t) = U_{\min} + \theta(U_{\max} - U_{\min})L(c_{ij}, n, t)$$

is the utility of the product. $U_{\min}$ is the basic utility for a beginner user, while $U_{\max}$ is the maximal utility that a user can get within a time cycle. The second term represents the effect of consumers' learning-by-doing: the utility of the product increases when we use it longer. Here, $L(c_{ij}, n, t)$ is the skill needed for using a product that a consumer has acquired until time t. The skill is defined as

$$L(c_{ij}, n, t) = \begin{cases} 0 & \text{for} t = 0 \\ \varepsilon X(c_{ij}, n, t-1), \varepsilon = \lambda(1-\lambda) & \text{for} t \geqslant 1 \end{cases}$$

where λ is a constant between 0 and 1 that represents the speed of skill depreciation. $L(c_{ij}, n, 0)$ is an initial condition between 0 and 1. As a result, the learning-by-doing effect is the product of the skill accumulation $L(c_{ij}, n, t)$ and its absolute weight on the total utility $\theta(U_{\max} - U_{\min})$.

$$E(c_{ij}, n, t) = (1-\theta)(U_{\max} - U_{\min})N(c_{ij}, n, t-1)$$

is the *network externality*, where $N(c_{ij}, n, t-1) = \sum_{\text{neighborhood}} X(c_{\text{neighborhood}}, n, t-1) / \eta$. Here η is the number of cell's neighbors. For each cell, we compute the number of neighbors that use the product n at the previous time step. The term $(1-\theta)(U_{\max} - U_{\min})$ is the absolute weight for the effect of network externality.

$$P(c_{ij}, n, t) = (X(c_{ij}, n, t) + X(c_{ij}, 3-n, t))P_n(c_{ij}, n, t) \quad n = 1 \text{or} 2$$

is the pricing function, where $P_n(c_{ij}, n, t)$ is the price for the corresponding product, defined as follows:

$$P_n(c_{ij}, n, t) = Q(n, t) + R(m, n, t)$$

Here $Q(n, t)$ is a given constant representing the global price for a product and $R(m, n, t)$ is a variable representing the changing price for the product. In order to maintain a balance between profits and market share, companies may reduce the local price to attract more consumers when it loses market share to its rivals, and raise the local price to gain more profits when it has a bigger market share than its competitors. The changing local price is defined as follows:

$$R(m,n,t) = Rn_{min} + \mu(Rn_{max} - Rn_{min})\\ \left(N(c_{ij}, n, t-1) - N(c_{ij}, 3-n, t-1)\right)$$

where Rn_{max} and Rn_{min} are the maximum and minimum local price for the product, and μ is a constant representing a fluctuation coefficient. The value of $N(c_{ij}, n, t-1) - N(c_{ij}, 3-n, t-1)$ reflects the influence of the market share on the local price. When a product has a bigger share of the local market, this item has a positive value, which means the local price will rise; when the local market share goes down, the local price will fall accordingly.

We used these rules, which were defined by Martín del Rey as a standard CA,[82] and we redefined them using Cell-DEVS formal specifications and executed the model using CD++. In the following discussion, we will use a 30 × 30 cell space that represents the market, with Moore neighborhood (the 9 near neighbors) and transport delays. Based on this, the coupled Cell-DEVS model is as follows:

Duopolistic Market

$$= < Xlist, Ylist, I, X, Y, n, \{t_1, t_2\}, N, C, B, Z, select >$$

where $Xlist = Ylist = I = X = Y = \Phi$ as this is a closed model.

As this is a 2D model, $n = 2$, and $\{t_1, t_2\} = \{30, 30\}$.

N is the neighborhood set, defined as **N** = {(−1,−1), (−1,0), (−1,1), (0,−1), (0,0), (0,1), (1,−1), (1,0), (1,1)}.

C is the cell space set, defined as **C** = $\{C_{ij} / i \in [1,30], j \in [1,30]\}$, where each C_{ij} is an atomic Cell-DEVS model that will be defined below.

B is the border cells set, here the cell space is wrapped, that is, **B** = {ϕ}.

Z is the translation function, defined by:

$P_{ij}Y_1 \rightarrow P_{i,j-1}X_1$	$P_{i,j+1}Y_1 \rightarrow P_{ij}X_1$
$P_{ij}Y_2 \rightarrow P_{i+1,j-1}X_2$	$P_{i-1,j+1}Y_2 \rightarrow P_{ij}X_2$
$P_{ij}Y_3 \rightarrow P_{i+1,j}X_3$	$P_{i-1,j}Y_3 \rightarrow P_{ij}X_3$
$P_{ij}Y_4 \rightarrow P_{i+1,j+1}X_4$	$P_{i-1,j-1}Y_4 \rightarrow P_{ij}X_4$
$P_{ij}Y_5 \rightarrow P_{i,j+1}X_5$	$P_{i,j-1}Y_5 \rightarrow P_{ij}X_5$
$P_{ij}Y_6 \rightarrow P_{i-1,j+1}X_6$	$P_{i+1,j-1}Y_6 \rightarrow P_{ij}X_6$
$P_{ij}Y_7 \rightarrow P_{i-1,j}X_7$	$P_{i+1,j}Y_7 \rightarrow P_{ij}X_7$
$P_{ij}Y_8 \rightarrow P_{i-1,j-1}X_8$	$P_{i+1,j+1}Y_8 \rightarrow P_{ij}X_8$
$P_{ij}Y_9 \rightarrow P_{i,j}X_9$	$P_{i,j}Y_9 \rightarrow P_{i,j}X_9$

The tie-breaking function **select** = {(−1,−1), (−1,0), (−1,1), (0,−1), (0,0), (0,1), (1,−1), (1,0), (1,1)}.

The atomic Cell-DEVS specification is as follows:

Market Atomic

$$= < X, Y, I, S, E, \text{delay}, \text{d}, \delta_{int}, \delta_{ext}, \tau, \lambda, \text{ta} >$$

where

X = {0, 1, 2} is the set of external input events;

Table 1. Local transition function rules.

Result	Rule
1	$V(c_{ij}, 1, t) > V(c_{ij}, 2, t)$ AND $V(c_{ij}, 1, t) > 0$
2	$V(c_{ij}, 2, t) > V(c_{ij}, 1, t)$ AND $V(c_{ij}, 2, t) > 0$
0	$0 > V(c_{ij}, 1, t)$ AND $0 > V(c_{ij}, 2, t)$

Y = {0, 1, 2} is the set of external output events;

I = $< \eta, \gamma, P^x, P^y >$, where $\eta = 9$, $\gamma = 0$, $P^x = \{(X_1,\text{integer}), (X_2,\text{integer}), (X_3,\text{integer}), (X_4,\text{integer}), (X_5,\text{integer}), (X_6,\text{integer}), (X_7,\text{integer}), (X_8,\text{integer}), (X_9,\text{integer})\}$, $P^y = \{(Y_1,\text{integer}), (Y_2,\text{integer}), (Y_3,\text{integer}), (Y_4,\text{integer}), (Y_5,\text{integer}), (Y_6,\text{integer}), (Y_7,\text{integer}), (Y_8,\text{integer}), (Y_9,\text{integer})\}$;

$S = \{s, \sigma queue \,/\, \text{where } s \in \boldsymbol{\theta}, \sigma queue = \{((v_1, \sigma_1),..., (v_m, \sigma_m)) \,/\, m \in \boldsymbol{N} \wedge m < \infty \wedge \forall\ (i \in \boldsymbol{N}, i \in [1,m]), v_i \in \{0,1,2\} \wedge \sigma_i \in \boldsymbol{R}_0^+ \cup \infty\}$, and

$\boldsymbol{\theta} = \{ value, vproduct1, vproduct2\} = \{0, 1, 2\} \times \boldsymbol{R}_0^+ \times \boldsymbol{R}_0^+$ is the set of states for the cell, where

0 represents no users; 1 represents users of product 1, and 2 represents users of product 2;

E = $\{0, 1, 2\}^9$;

delay = transport; **d** = 100;

$\boldsymbol{\delta_{int}}$: S→S, $\boldsymbol{\delta_{ext}}$: Q × X→S, **λ**: S→Y and **ta**: S→$\boldsymbol{R}_0^+ \cup \infty$, are defined as in Section 2.

τ: E→S is the local transition function, computed as a set of rules considering the skill accumulation function $L(c_{ij}, n, t)$ discussed above, and the corresponding rules shown in Table 1.

Using this formal specification as a basis, we defined the corresponding rules and the coupled model in CD++, which can be categorized as rules for a mature market and for a new market. The former group uses an initial cell space where the three possible states for each cell are uniformly distributed to represent a mature market; the latter group uses only a few cells active. In each group, we modeled products having the same price, products with different prices, and prices fluctuating. The atomic Cell-DEVS specification uses variables *vproduct1* and *vproduct2* to keep the value of $L(c_{ij}, n, t)$, whose values are updated at the beginning of each time step to keep track of the values of $L(c_{ij}, 1, t)$ and $L(c_{ij}, 2, t)$, respectively. The resulting rules are as follows:

```
  [ choice_rule]
  % if state = 0, vProduct1 and vProduct2
are deprecated by λ(1 - λ), i.e., 0.25
  rule                                  :
{ if((stateCount(1) + 9*$vProduct1)>(st-
ateCount(2) + 9*$vProduct2), 1,
     if((stateCount(2) + 9*$vProduct2)-
>(stateCount(1) + 9*$vProduct1), 2 ,0))}
```

```
      { $vProduct1:=$vProduct1*0.25; $vProduct2:=$vProduct2*0.25;} 100{ (0,0)=0}

% if state = 1, vProduct1=(vProduct1 + 1)*0.25, vProduct2=vProduct2*0.25
rule : { if((stateCount(2) + 9*$vProduct2) > (stateCount(1) + 9*$vProduct1), 2, 1)}
  { $vProduct1  :=  ($vProduct1 + 1)*0.25;  $vProduct2  :=  $vProduct2*0.25;  }  100
  { (0,0)=1}

% if state = 2, vProduct1=vProduct1*0.25, vProduct2=(vProduct2 + 1)*0.25
rule : { if((stateCount(1) + 9*$vProduct1) > (stateCount(2) + 9*$vProduct2), 1, 2)}
  { $vProduct1  :=  $vProduct1*0.25;  $vProduct2  :=  ($vProduct2 + 1)*0.25;  }  100
  { (0,0)=2}
```

When we execute this model, the results obtained are as shown in Figure 9.

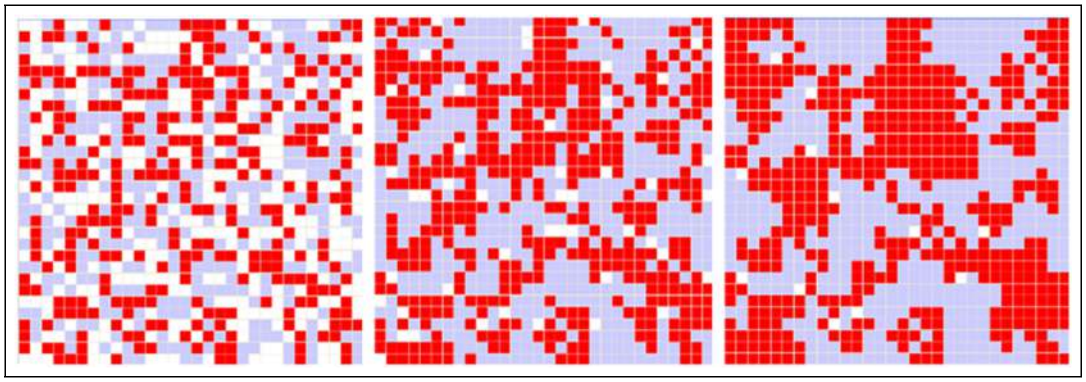
Figure 9. Mature market and same price scenario.

The simulation result shows that non-users begin to use one of the two products with approximately equal probability, and users using the same products tend to aggregate together to form their own society, which in turn enhances the network externality.

When we modify the model to consider products with different price, the local computation rules change as follows:

```
[ choice_rule]
% if state = 0, vProduct1 and vProduct2 are deprecated by λ(1 - λ), i.e., 0.25
rule : { if((stateCount(1) + 9*$vProduct1 + 2.25) > (stateCount(2) + 9*$vProduct2), 1 ,
   if((stateCount(2) + 9*$vProduct2) > (stateCount(1) + 9*$vProduct1 + 2.25), 2 , 0))}
     { $vProduct1:=$vProduct1*0.25; $vProduct2:=$vProduct2*0.25;} 100{ (0,0)=0}

% if state = 1, vProduct1=(vProduct1 + 1)*0.25, vProduct2=vProduct2*0.25
rule : { if((stateCount(2) + 9*$vProduct2) > (stateCount(1) + 9*$vProduct1 + 2.25), 2, 1)}
  { $vProduct1  :=  ($vProduct1 + 1)*0.25;  $vProduct2  :=  $vProduct2*0.25;  }  100
  { (0,0)=1}

% if state = 2, vProduct1=vProduct1*0.25, vProduct2=(vProduct2 + 1)*0.25
   rule : { if((stateCount(1) + 9*$vProduct1 + 2.25) > (stateCount(2) + 9*$vProduct2),
1 , 2)}
     { $vProduct1  :=  $vProduct1*0.25;  $vProduct2  :=  ($vProduct2 + 1)*0.25;  }  100
     { (0,0)=2}
```

When we execute this model, the results obtained are as in Figure 10.

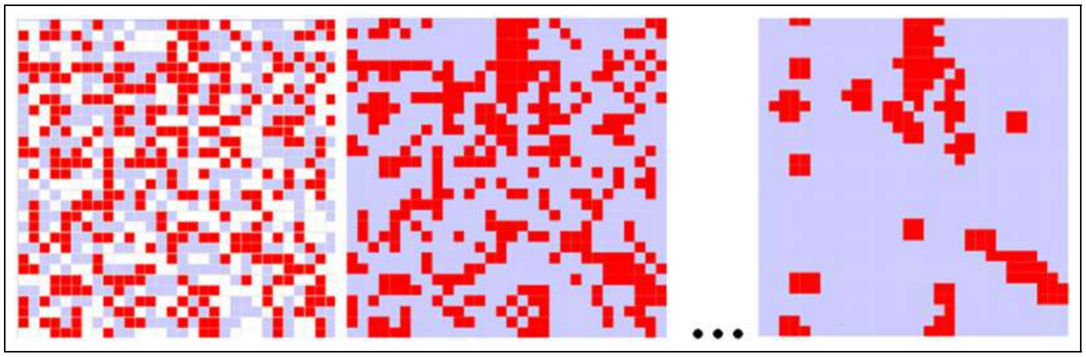
Figure 10. Mature market and different price scenario.

Since the price for *product1* is lower than for *product2* and all other parameters are the same, most of the non-users choose to use *product1*. At the same time, some original *product2* users shifted to *product1* as well. The effect of network externality results in the aggregation of users of the same products.

When we consider a mature market with fluctuating price, we modified he local computation rules as follows:

```
[ choice_rule]
% if state = 0, vProduct1 and vProduct2 are deprecated by λ(1 - λ), i.e. 0.25
  rule :{ if((9*$vProduct1 - 9*$vProduct2) > (0.4*stateCount(2) - 0.4*stateCount(1)), 1 ,
    if((1.8*$vProduct2-1.8*$vProduct1) > (0.08*stateCount(1)-0.08*stateCount(2)),
    2, 0))}
      { $vProduct1 := $vProduct1*0.25; $vProduct2 := $vProduct2*0.25; } 100{ (0,0)=0}

% if state = 1, vProduct1=(vProduct1 + 1)*0.25, vProduct2=vProduct2*0.25
rule : { if(1.8*$vProduct2-1.8*$vProduct1) > (0.08*stateCount(1)-0.08*stateCount(2),
 2, 1)}
{ $vProduct1:=($vProduct1 + 1)*0.25; $vProduct2:=$vProduct2*0.25;} 100{ (0,0)=1}

% if state = 2, vProduct1=vProduct1*0.25, vProduct2=(vProduct2 + 1)*0.25
  rule :{ if((9*$vProduct1-9*$vProduct2) > (0.4*stateCount(2)-0.4*stateCount(1)), 1, 2)}
    { $vProduct1  :=  $vProduct1*0.25;  $vProduct2  :=  ($vProduct2 + 1)*0.25;  }  100
    { (0,0)=2}
```

When we execute this model, the results obtained are as shown in Figure 11.

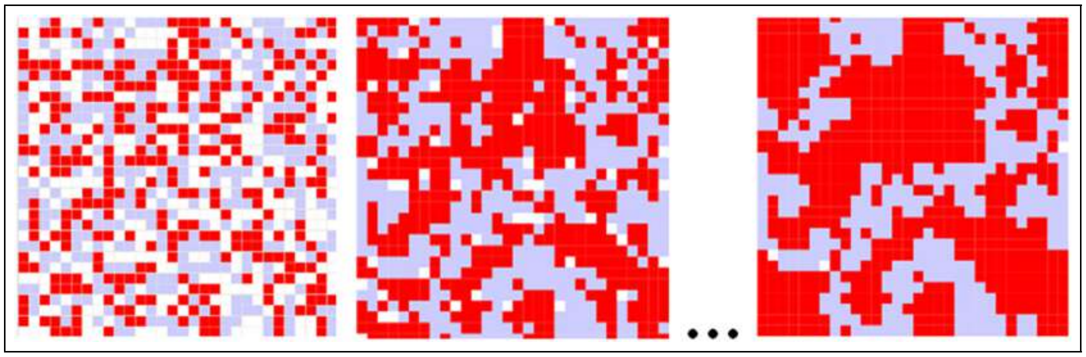
Figure 11. Mature market and fluctuating price scenario.

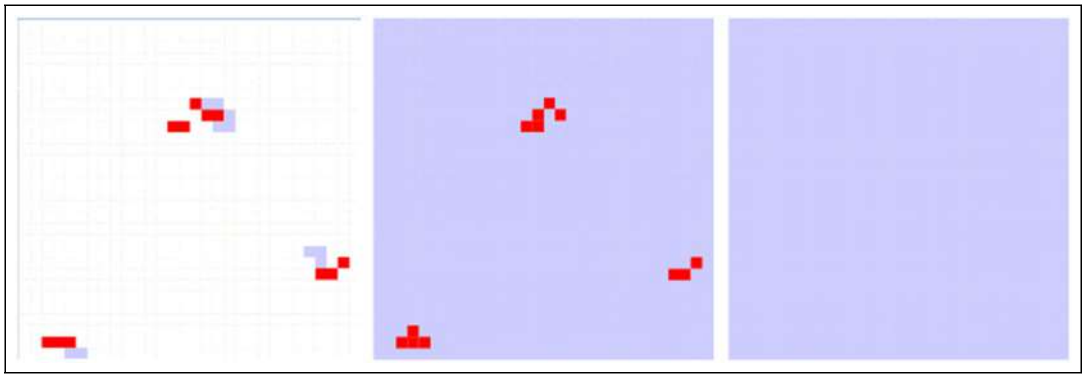

Figure 12. New market and different price scenario.

In this case, *product2* has higher pricing flexibility ($\mu_2 = 1$), while *product2* offers more rigid ($\mu_1 = 0.2$) prices. As a result, *product2* gains bigger market share. Besides, if the local market shares for both products are equal, i.e., for cells that have exact four red neighbors and four blue neighbors, the price fluctuation disappears. A tie happens. In this situation, the network externality becomes the sole force in determining consumers' decisions.

In the case of a new market with different price, the local computation rules are the same as those in a mature market (but the initial values of the model are different, as seen in Figure 12). When we run this model, we obtain the results shown in Figure 12.

Figure 12 shows how *product1* rapidly monopolizes the whole market by virtue of its lower prices (sensitivity of price is high in a new market).

6. Occupancy model: pedestrian movement in a building

In this section, we show a Cell-DEVS model that simulates the behavior of individuals moving through a 3D object representing a building, as presented elsewhere.[30,34,52] As can be seen in Figure 13, the building has two floors connected by stairs. Each floor uses 10×22 cells to represent the building space; cell (3, 0, 0) represents the entrance, cell (8, 21, 1) represents the stair going downstairs, etc.

If we build this model using a traditional CA, we need to consider that we have a single state variable per cell. How can one model such application using a Cell-DEVS model that mimics the CA behavior? In order to represent a complex model like this one, we can use a higher dimensional model, as seen in the figure. In this example, we can add a fourth dimension to each cell to represent different phenomena. We can see the model as a set of 3D rectangular cuboid interconnected by a fourth dimension orthogonal to each cell. Each cuboid contains the computation devices for different phenomena: movement, pathway, layout, and hot zones. Therefore, each ''cell'' in the figure above is, in reality, a 3D model with four layers each. Each layer is defined as follows:

- **Movement**: this layer is used to track the movement of individuals. A cell with a value of 0 is empty, and 1 means the cell is occupied; the cuboid also records other states related to four phases that reflect the relationship with the neighbors.
- **Phase**: each person's movement is done in a four phase cycle (Intent, Grant, Wait, Move), whose details will be discussed later.

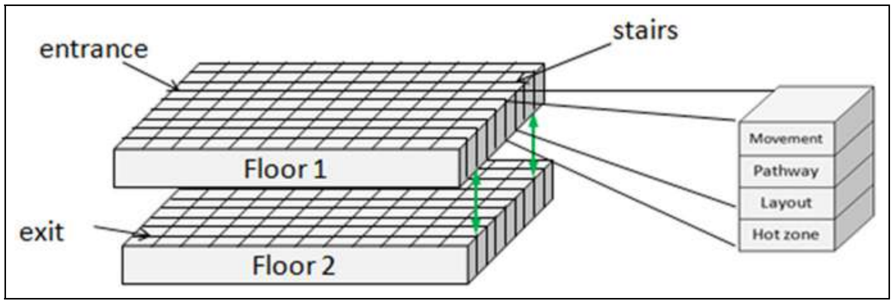

Figure 13. Four-dimensional cells.

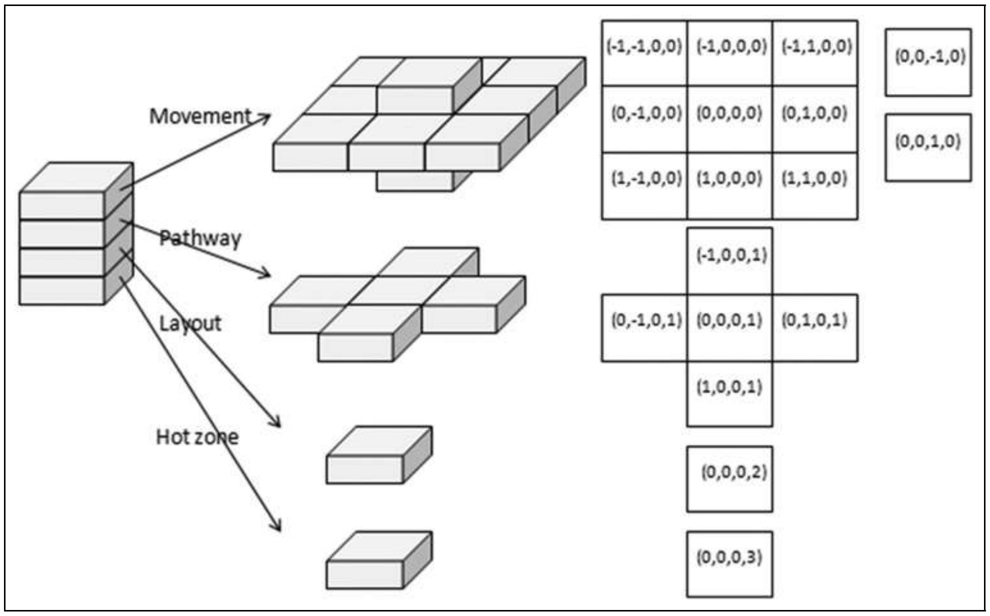

Figure 14. Neighborhood.

- **Pathway**: shows the possible visit routes. Visitors of the building tend to move following this pathway with a given probability.
- **Layout**: the shape of the building, including walls and obstacles.
- **Hot zone**: this cuboid reflects the popularity levels of different areas of the building, which is defined by using different delays. The higher value of the hot zone is, the higher the probability that a visitor would stay there.

In order to represent the people's movement in all possible directions, the model uses a combination of an extended Moore and Von Neumann neighborhood, as shown in Figure 14.

We first use nine neighbors ((−1,−1,0,0)…(1,1,0,0)), indicating one cardinal direction each. For the stairs, we need two other neighbors: one to go down (0,0,−1,0), and one for receiving people from upstairs (0,0,1,0). We also need to access the pathway (0,0,0,1), the layout information (0,0,0,2), and hot zone information (0,0,0,3), each represented by a different cuboid connected to the others through orthogonal 4D links. The pathway layer uses a Von Neumann neighborhood ((0,1,0,1)…(−1,0,0,1)). The pathway is a route plan for the movement, which is built by overlaying a Voronoi diagram of the route to an exit or stair. The layout layer contains the building information, which enables a building designer to use and extend this layer during the design process.

As discussed earlier, the movement behavior is divided into four phases (*Intent, Grant, Wait*, and *Move*), which can be represented by the state diagram shown in Figure 15.

When a cell is occupied, the individual will first choose a direction at random at the first phase, known as the *intent* phase, where we check and see if the target cell accepts the individual. In that case, the occupied cell changes to the *get grant* state, and then we *wait* a certain amount of time according to the hot zone data for the cell where the visitor is standing. After this period, which reflects the visitor moving through the building, we *empty* the cell at the *move* phase. If we *get rejected*, the visitor waits and the cell keeps occupied. An *empty* cell chooses a neighbor that is at the *grant* phase and changes to *occupied* during the *move* phase.

At the beginning of the simulation, we add visitors at the main entrances with certain probability, in order to mimic different input flow rates with rush/slash hour during the opening time. Each cycle takes 4 time units. We first check to see if it is the beginning of the cycle *(remainder(time, 4)=0)*, we check the second dimension cuboid (where the building layout is store) in order to see if this cell is an entrance (`(0,0,0,2)=entrance`), and then generate a person at each entrance of Floor 1, as seen in the following rule:

```
rule : {1} 4 { remainder(time,4)=0 and
(0,0,0,0)=0 and (0,0,0,2)=entrance}
```

Phase 1: Intent

During this phase, the intent direction is determined by two factors: the pathway direction and its probability. We

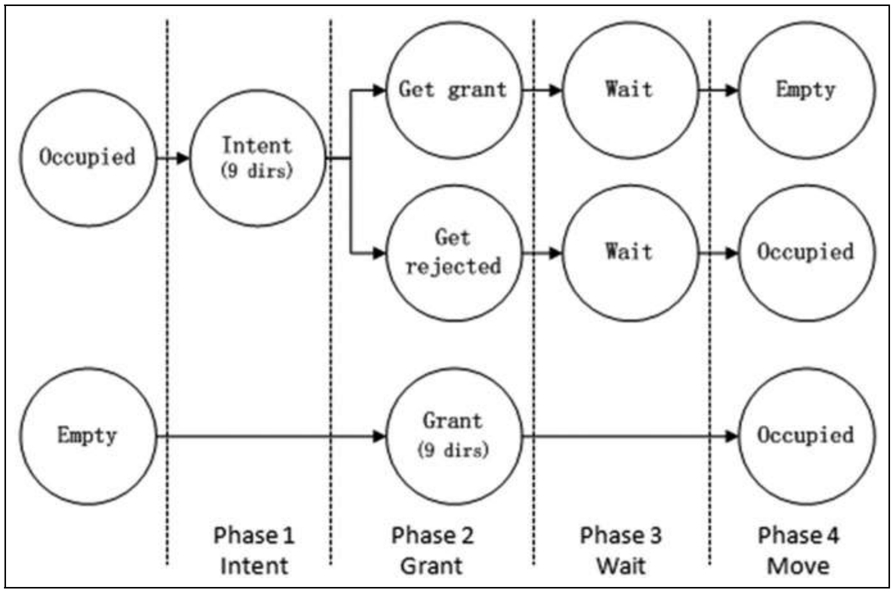

Figure 15. State diagram of movement.

first check if we are in the intent phase (`remainder(-time,4)=0`), and if the cell is a stair, it gets the intent direction `intentDown` as seen in the following rule.

```
rule : { intentDown} 1{ remainder(time,4)=0
  and (0,0,0,0)=occupied and
  (0,0,0,2)=stair to floor 1}
```

If the cell is not a stair, we find the probability of moving in different directions according to the value stored in the pathway cuboid. For example, if the pathway is N, we would move according to the first chart of Figure 16. Then, we check in which direction that random number is located. For example, if the random number is 58, it is in *%F (0-69)*, so we get the intent direction to go *N* (or ''Front'').

The first step is to generate a random number between 0 and 100 on each cell, as follows:

```
rule : { uniform(0,1)} 1{ remainder(time,4)=0
  and (0,0,0,0)=1 }
```

Finally, the cell state is changed to a value corresponding to the intent direction, as defined in the following rules (one rule for each direction). Note here that we do not care whether the target cell is available; this will be checked in the following phases.

```
rule : { E} 0{ (0,0,0,1)=5 and (0,0,0,0) > 0.0
  and (0,0,0,0) < = #Macro(Front)
...
rule : { NW} 0{ (0,0,0,1)=6 and (0,0,0,0)
   > #Macro(Front) and (0,0,0,0)
   < = #Macro(Front)
   + #Macro(Left-Front) ...
...
rule : { SW} 0{ (0,0,0,1)=8 and (0,0,0,0) >
  #Macro(Front) + ... + #Macro(Right-Front) }
```

Phase 2: Grant

After we chose our intended direction, more than one individual may want to enter into a same cell (which would result in a collision). To handle this problem, each empty

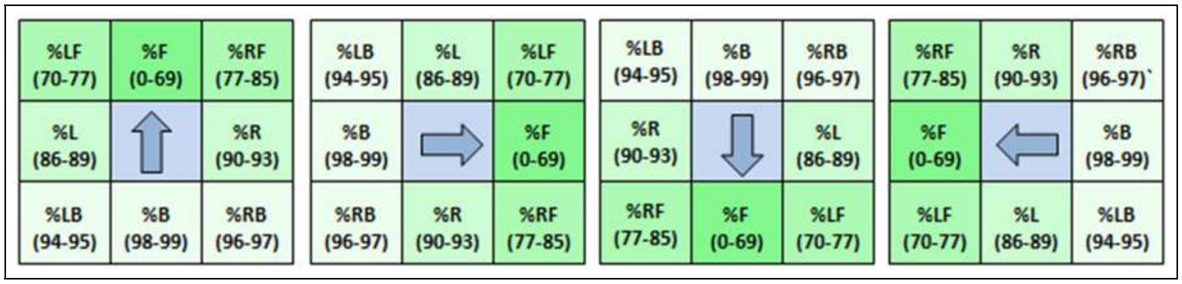

Figure 16. Intent probability distribution with the pathway.

cell will choose only one neighbor between all the candidates, and change its state accordingly. We first check if the cell above cell above wants to come down (`(0,0,-1,0)=wantDown`), and in that case, we grant it. We need to check the phase (`remainder(time, 4)=1`).

```
rule : { GrantDown} 1{ remainder(time, 4)=1
  and (0,0,0,0)=0 and (0,0,-1,0)=wantDown}
```

Otherwise, we choose an intent cell in the neighborhood (in a predefined order), and we change to the *Grant Target* state with the corresponding direction, whose value corresponds with one of the reverse eight directions. For example, in the rules below GTW means that the current cell accepts the left neighbor to come in (Grant Target West). We also need to ensure that the cell is empty, and this is not a building wall (found on the layout plane). Finally, we need to ensure that we are in the Grant phase (`remainder(time,4)=1`). The rules for doing this are summarized below.

```
rule : { GTW} 1{ remainder(time,4)=1 and
  (0,0,0,0)=0 and (0,-1,0,0)=IntentE and
  (0,0,0,2)!=Wall}
...
rule : { GTSW} 1{ remainder(time,4)=1 and
  (0,0,0,0)=0 and (-1,-1,0,0)=IntentNE and
  (0,0,0,2)!=Wall}
```

Phase 3: Wait

We use this phase to mimic the behavior of random wait in the building. If an individual has been granted to move by the target cell (they have an intent direction and the target cell choses them), we do not move until a given random amount of time has passed, in order to better represent the movement delays in the building. The visitors will stop in different areas with different probability. These delays are generated according to the hot zone value. We implement this by using different delays for each cell. We first get a random number between 0 to the hot zone value. As the cycle is 4 time units long, plus 1 time unit for the fourth *Move* phase; we need to wait 4*N + 1 (if the random result N is, for instance 2, then we wait 9 time units). After that, a cell's value will change. The stairs do not use this delay, as seen in the following rule.

```
rule : { CanMoveStair} 1{ remainder(time,4)=2
  and (0,0,0,0)=GrantStair}
```

The following rules summarize the rest of the movement behavior. We check that we are in phase 3, and that the movement we intent to do has been granted. Then, we change the state to reflect that we can move in the chosen direction. We inform this after the random delay based on the information found on the hotspot cuboid.

```
rule : { CanMoveE} { 1 + 4*randInt
  ((0,0,0,3) + 1) } { remainder(time,4)=2
  and (0,0,0,0)=GrantE
  and (0,1,0,0) = IntentW}
...
rule : { CanMoveSW} { 1 + 4*randInt
  ((0,0,0,3) + 1) } { remainder(time,4)=2 and
  (0,0,0,0)=GrantSW and (1,1,0,0)=IntentNE}
```

If not granted, the person cannot move, and they should try again during the next moving cycle, and will only wait a 1 time unit until the next *Move* phase.

```
rule : { Wait} 1{ remainder(time,4)=2 and
  (0,0,0,0)=IntentNotGranted}
```

Phase 4: Move

Now, every cell that intended to change has been *granted* the desired movement or it is *waiting*; therefore, the granted individual can move to the target cell. To finish the move, we empty the intended cells that are granted, and the rejected ones keep the individuals that have not been granted to move. As in the previous cases, we need to check in which the phase we (`remainder(time,4)=3`).

```
rule : { 0} 1{ remainder(time,4)=3 and
  (0,0,0,0)=CaMove} // Empty the cell
rule : { 1} 1{ remainder(time,4)=3 and
  (0,0,0,0)=CannotMove} // Wait
rule : { 1} 1{ remainder(time,4)=3 and
  (0,0,0,0)=GrantedTargetCell} // Receive
person
```

So far, we have discussed entrances, stairs and normal cells. The last thing is to consider the exit; we need to empty it if it is occupied, using the following the rule:

```
rule : { 0} 1{ (0,0,0,2)=exit and (0,0,0,0)=1}
```

As we can see, using multiple 4D rules is useful but it might be complex to read and debug. To reduce the complexity of the model definition, we have used different macros to make the rules more readable. Nevertheless, having a single value per cell makes increases the rules complexities. Using a 4D model with interconnected cuboids also increases the memory footprint and the number of objects required to build the cell space, which can slow down the simulation. Likewise, the size of the cell space is higher, and the definition of the neighborhood is more complex.

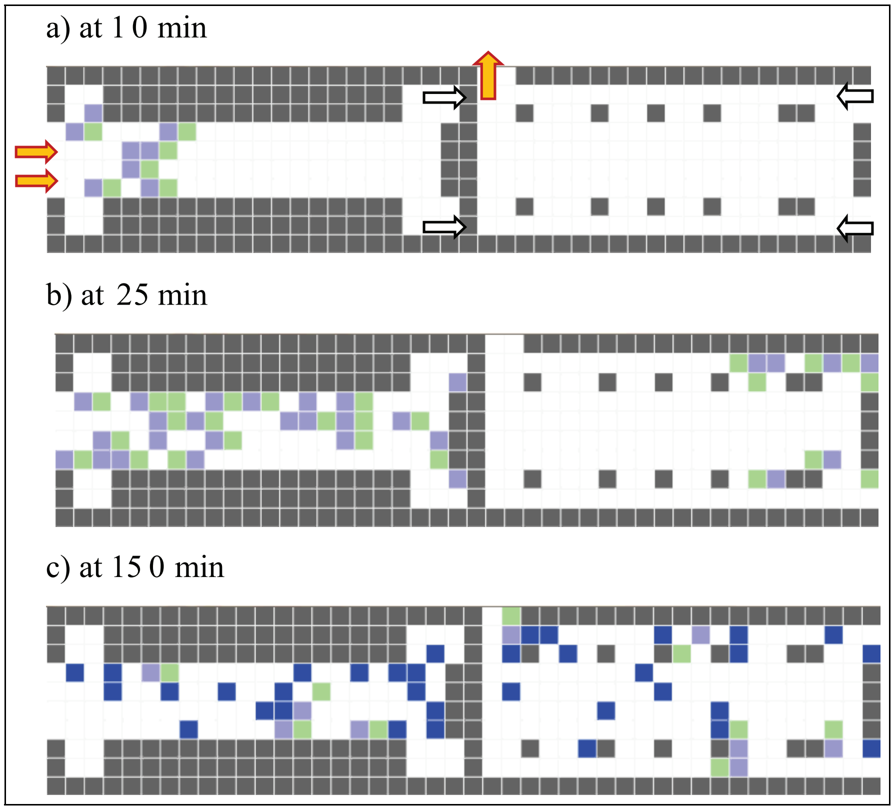

Figure 17. Simulation results of basic properties at different simulation times.

Instead, building advanced Cell-DEVS models multi-variable and multi-port Cell-DEVS models, such as those in CD ++ , allow us to save memory, execution time, and make the models easier to read and debug. For instance, by using different state variables in a model like this, we simplify the neighborhood shape. For instance, we would only need the shape defined in the Movement plane shown in Figure 14. The 4D model can now be represented as a 3D model, and the rules can be simpler. For instance, the rule we defined above:

```
rule : { CanMoveSE} { 1 + 4*randInt
  ((0,0,0,3) + 1)} { remainder(time,4)=2 and
  (0,0,0,0)=GrantSE
  and (0,1,0,0) = IntentNW}
```

is, in fact, encoded using real numbers as in the formal definition of CA and Cell-DEVS as follows:

```
rule : { 38} { 1 + 4*randInt((0,0,0,3) + 1) }
 { remainder(time, 4)=2 and (0,0,0,0) = 28
  and (1,1,0,0)=48 }
```

As in CA the cells can have only one value, we need to encode all the information in an integer value. In this case particular case, `CanMoveSE` is encoded as 38, where the code means that we are moving to the East (encoded as the digit 8 in the units), and the fact that we are in the third phase (represented by the tenth 3 in number 38). We need to check if our cell in phase 2 has been granted to move to the East (encoded as 28 and checked by the condition `(0,0,0,0) = 28`) and we need to check that in the cell to the NW has the intent to move in the SE direction (`(1,1,0,0)=48`). Likewise, we generate a random number and we need to remember that the hot zone information is stored in cuboid 3, therefore we check cell `(0,0,0,3)` and generate a random number based on the value found there.

As we can see, using a single state value like in CA can result in complex rules, in particular when we need to represent varied information on each of the cells. Instead, using specific state variables and individual ports makes the development of the applications easier, as we have seen in the section before and we will see in the following section. For instance, the rule above can be defined as:

```
rule : { ~movement:=SE; ~phase:=4;}
  { 1 + 4*randInt($hotzone)} {
  (0,0,0)~phase=3 and
  (0,0,0)~movement=SE and (1,1,0)~movement =
SE }
```

which is simpler to read and debug. As we can see, now we only use a 3D model, in which each of the layers represent the individuals on each of the two floors. The hot zone information, the movement information, and the phases for conducting the movement without risk are represented using state variables and input/output ports, making the rules more readable and the model smaller.

The version of the model using extended variables and simpler was originally presented elsewhere in detail.[30,34,52] The simulation software generates exactly the same log files for both versions of the model, as both models represent exactly the same phenomena but using 3D or 4D models.

Here, in Figure 17, we show one of the tests we conducted, which shows the basic behavior of visitors under normal properties during rush hour. The simulation animation shows the two floors in the building, with different individuals (in blue) arriving and leaving the building through the doors (white cells, orange arrows) and moving downstairs (white cells, white arrows) from the top floor (in the left part of the graph), where the entrance to the building is. At rush hour, the house opens four entrances, and each entrance cell generates one visitor during each cycle as discussed earlier. As we explained, the blue cells represent visitors (light blue cells are waiting visitors, and green cells are destination cells). We conducted various studies, presented by Wang and Wainer,[52] in which we analyzed the impact of door location/stairs for occupancy. When we changed the entrance positions, there was a small change in terms of building occupancy, and no conflicts between individuals. We also changed the number of stairs. In this case, the occupancy levels changed meaningfully, except in the cases where there was congestion at the stairs. We conducted numerous tests, changing the values in hot zones, entrances location, movement direction probabilities, coming rate, etc. These changes are straightforward by doing simple modifications in the rules above and in the initial conditions of the model, and it allows a designer to do varied analysis with ease.

We built a plugin for the 3dsMax tool in order to be able to include the simulation results into a 3D scene, as presented in Figure 18. As we can see, the floor plan includes the simulation results obtained from the model above, in which we show the two floors and the direction of each of the individuals visiting the building at different times of the simulation. All the results presented are based on the results provided by CD + + .

7. Modeling computer malware

Nowadays, computer networks are ubiquitous, and these new services bring security issues, as the vulnerability of these distributed systems is high. In this section, we show how to model one kind of such attacks, namely computer malware, using a 2D Cell-DEVS model. We follow the recent research by various authors who recently proposed modeling malware using CA,[82–85] and here we show how to extend those concepts in a Cell-DEVS model, which provides as a better and simpler way to model those kinds of applications. Among the different kinds of malware and viruses, we will show how to model *worms*, self-replicating computer viruses that can propagate through the network without any human intervention.

Mathematical epidemiology has existed for over a hundred years. Epidemic modeling has been used to imitate the spreading of infectious diseases for a given population, such as H1N1, SARS, and influenza.[84] Infected individuals spread the virus to healthy individuals that they contact. Because worms are very similar to biological viruses in replication and spread behaviors, existing epidemiological models have been used to study how the computer viruses spread. In particular, we are interested in cellular models to study epidemics as well as worm propagation in networks. For instance, Martín del Rey presented a CA model in which each cell represents a computer in the network.[82,84] Each computer can be in one of three possible states: S (susceptible – not infected by the virus); E (exposed – infected by the virus but not activated); or I (infected – the virus is activated and it is able to propagate). Similar results have been presented to study worm propagation in smartphones.[83,85]

Instead of using a uniform model with identical cells, we assigned different roles to different cells. We introduce an *attacker*, whose objective is to deploy malware that will spread to the computers that do not have anti-virus software (or have obsolete versions). The attacker can control these infected computers.

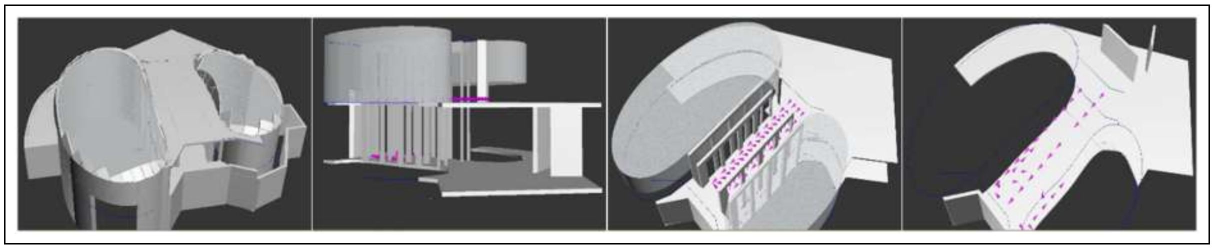
Figure 18. 3D visualization of the simulation of the occupancy model.

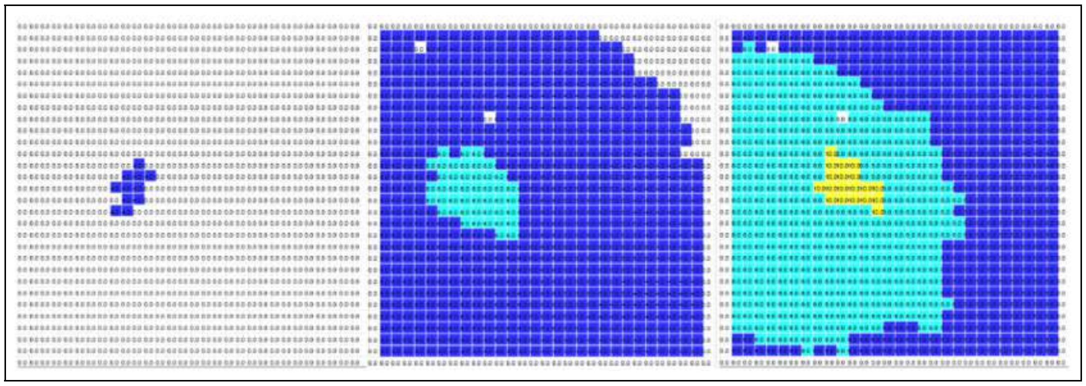

Figure 19. Simulation: one attacker.

The following rules show the basic behavior for spreading the virus. We use a timer that is used to decide when we attack computers; we also keep and update the version number of the current malware (*virusvers*).

```
rule : { ~virusvers := $virusvers; } { $timer
  := $timer + uniform(4,6); } 100
{ $identity = 100 and $timer < 60 }
rule : { ~virusvers := $virusvers;} { $timer
  := 0; $virusvers := $virusvers + 2;} 100
  { $identity = 100 and $timer > 60 }
```

The attacker is on a cell with *identity* 100. The attacker cell uses the timer to implement a random interval between issuing malwares (*timer*). Every update will make the version number increase to a new value. The goal of the attacker is to spread it as widely as possible. Then, it can intrude and even control these infected computers. This process can be divided to two stages. In the first stage, an unprotected cell is infected. Then, its state changes from ''unprotected'' to ''infected.'' In the second stage, a new version of the malware (which is represented as one with a higher value of the variable *virusvers*) will replace the current version of malware in the cell. The state of cell is still ''infected,'' but at this time by more advanced malware. Each cell checks to see if its version of malware is higher than their neighbors' current versions. If so, the neighbors are also infected or they update their version of malware. The rules for doing this are as follows:

```
rule : { ~virusvers :=
  #Macro(virusspreading); ~state := -1; }
  { $virusvers := #Macro (virusspreading); }
  100 { #Macro(virusspreading) > $virusvers
  and #Macro
(virusspreading) > $myAntiVirusVers and
  $identity = 0 and random <
  ( #Macro(PossibilityOfSpreading)) }
```

Here, we use the macro *virusspreading* to find out the highest version number among a cell's 8 neighbors. We use a random function to take into account the probability of spreading (i.e., the cells are infected conditionally). The following figure shows the simulation results for one attacker. As we can see from Figure 19, the initial attacker infects a few cells, which in turn will end up infecting the whole network quickly.

Our model includes the modeling of antivirus software (or technical support that can clean the malware). These ''clean'' cells are exempt from being infected. Clean cells monitor their eight neighbors and check out if they are infected. An infected neighbor will request the clean cell to transfer anti-malware software (which also has a version number). If a clean cell finds malware whose version number is lower when compared with the newest anti-malware software's, then the clean cell will ignore the request (because anti-malware software has been deployed to eliminate this malware version in the network).

The rule for producing and updating a new version of anti-malware software is shown below:

```
rule : { ~anvirusvers :=
  #Macro(virusspreading) + 1; ~virusvers :=
  0; ~state := 2; }
{ $virusvers := 0; $myAntiVirusVers :=
  #Macro(virusspreading) + 1; } 100
{ ( #Macro(virusspreading) >
  $myAntiVirusVers or
  #Macro(anvirusspreading) >
  $myAntiVirusVers ) and $identity = 200 }
```

Every time the clean cell encounters a new version malware, it updates the anti-malware software (the version number of malware plus 1). This ensures that the malware can be wiped out by the anti-malware software. The clean cell outputs the current version number of anti-malware to its neighbor, using a port called *~anvirusvers*. The state variable *$myAntiVirusVers* records the current version of anti-malware software. The following graphs in Figure 20 show two clean cells producing anti-malware and updating it to a new version.

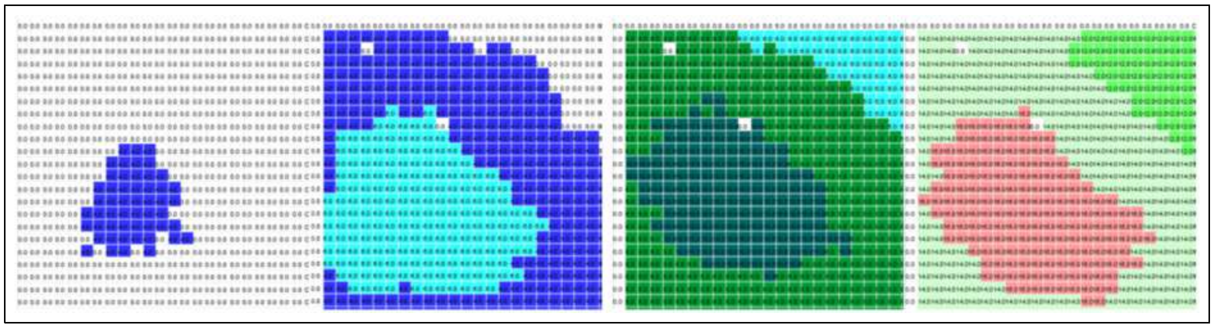

Figure 20. Simulation results of malware and antimalware interacting.

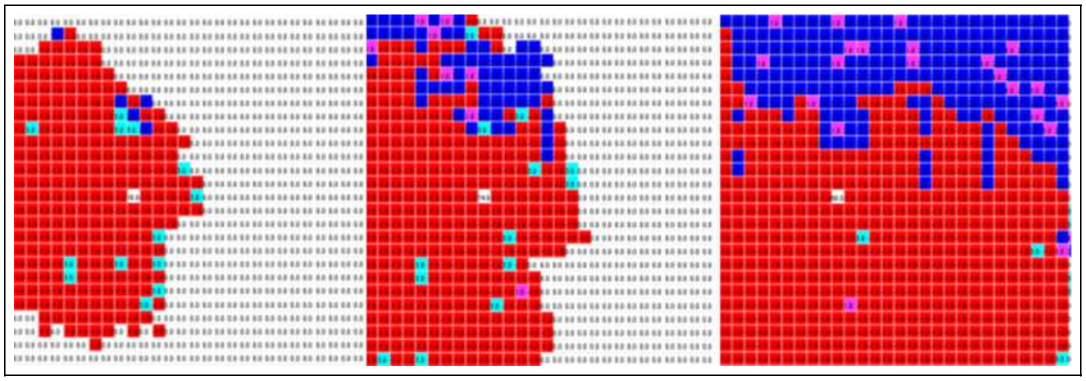

Figure 21. Simulation of malware with automated network updates for the anti-malware software.

In the left graph, we can see that the malware has begun to spread. Several cells have been infected. In the next figure, we can see that the clean cells (white cells) have not responded to the malware, which has updated its version number to 6 (light blue cells), and the clean cells have detected the malware and have updated its version of anti-malware to 7 to fight the malware. As we can see in the following figures, the anti-malware is transmitted to the neighboring cells, the malware is cleaned (black cells), and eventually most of the cells are clean.

A clean cell needs to spread its anti-malware software to its neighbors and eventually the whole network in order to fight against the malware made by attacker. Typical anti-malware software cannot spread in the network automatically. Therefore, compared with the spread of malware, the spread of anti-malware software is more passive, which leads to a poor efficiency to fight against the malware in the network. In order to consider this feature, we decide to design a pattern of anti-malware, which can also spread from one computer to others automatically using a similar process to the one used for malware.

In this case, the attacker in the network updates malware frequently. Thus, the clean cells have to update their anti-malware frequently as well and they have little chance to wipe out the attacker finally. In order to satisfy this setting, we changed the timer frequency. The simulation results can be seen in Figure 21.

In the first graph, the clean cell has detected the malware in their neighboring cells, and it produces anti-malware. In the second graph, we can see the malware spread fast, and the anti-malware has only secured a small part of computers in the upper-left corner of this scene. Finally, every cell in the network has been involved but the malware still controls most of the cells. The clean cell has no chance to eliminate the attacker because of the frequent update of malware.

Instead, we now assume that every cell in the network has a better anti-malware strategy, which will make the spread of malware more difficult and the spread of anti-malware easier. To do so, we just adjust some basic settings in the model. We lower the possibility of spread of malware whereas we increase the possibility of anti-malware, while keeping all other settings unchanged. The simulation results are shown in Figure 22.

Initially, the malware has infected a few cells and the anti-malware is defending many others. The malware expands slowly, therefore giving enough time to update the anti-malware software. In the second graph, we can see that the anti-malware has spread out quickly and the malware is advancing. Nevertheless, at the end, the anti-

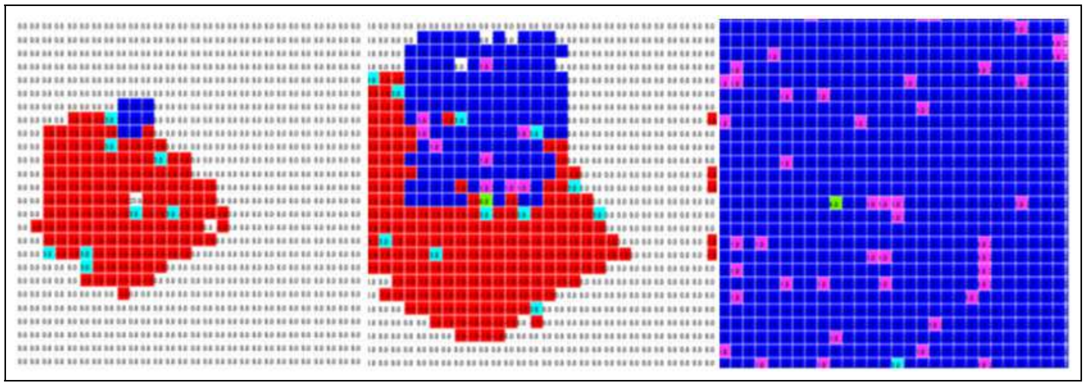

Figure 22. Simulation results of automated update of anti-malware in every computer in the network.

malware has been distributed to all the cells, getting rid of the malware, which has successfully secured or protected all the cells in the network.

8. Conclusion

We introduced different modeling applications of advanced Cell-DEVS models, a modeling formalism Norbert Giambiasi and I defined in the late 1990s. This work, which started in 1995, focused on new methods for studying complex systems with emergent dynamic behavior. Our focus was on formal models, which can improve the definition of the model and making easier their execution, as we could see in the different examples presented here. As shown in the paper, the executable models can be verified against a formal specification. The same models can be run in centralized, distributed or parallel simulators without change.

We extended the original definition of cellular models and defined Cell-DEVS, a timed cellular model specification based on DEVS with explicit timing delays, using Norbert's ideas on circuit design. Norbert's previous experience in digital circuits was fundamental for this new formal specification. These ideas have resulted in numerous research projects and interesting results in different domains. We discussed improved versions of these models built using the CD ++ toolkit, which was built in order to study, model and simulate such cellular models. The models have removed some limitations that standard cellular models have, which allow each cell to use multiple state variables multiple ports for inter-cell communications. We showed the application of the formalism in social models, pedestrian analysis, and computer networks.

Cell-DEVS was extended and combined with different methods to improve the modeling further. One of the first contributions in the year 2000 was the definition of activity-based models based on the dynamic quantization of the cell's value. We also combined Cell-DEVS with G-DEVS, another contribution of Dr. Giambiasi, and we combined Cell-DEVS and quantized DEVS with hysteresis, as well as computational fluid dynamics and finite elements and finite differences methods.

Several types of models can be integrated in an efficient fashion, allowing multiple points of view to be analyzed using the same model. The tools are public domain and can be obtained at `http://cell-devs.sce.carleton.ca`. The new implementation of CD ++ runs on the Cloud, allowing users around the world to run distributed experiments with ease. The simulator's features add power to the specification language, simplifying the modeling task at a cost of increased overhead required for the management of these features. In order to reduce this overhead, the modelers must pay more attention to model optimization. Cell-DEVS simplifies the construction of the models, allowing intuitive specification. The CD ++ logic rules facilitated the debugging phase and reduced development time, and the modeler does not need to focus on the simulation algorithms, which are handled internally by the CD ++ engine. Complex model modifications can be integrated easily and quickly. The numerous results provided by Cell-DEVS models are a testimony to the mentorship and legacy of Dr. Giambiasi and his contribution to different areas of knowledge.

Acknowledgements

This work is based on the research of numerous students that participated throughout the years: Alejandro López built a new CD ++ simulator, Baha Uddin Kazi and Wubing Ouyang defined malware models; Qi Liu defined the duopolistic market model, Sixuan Wang, Michael Van Schyndel, and Victor Freire worked on the evacuation model and visualization.

Funding

This work was partially supported by NSERC.

Appendix

CD++ rules grammar

```
RuleList = Rule | Rule RuleList
Rule = AssignResult Result{BoolExp}
    | AssignResult{AssignSet}Result{BoolExp}
AssignResult = Result |{PortSendSet}
Result = Constant | UNDEF |{RealExp}
BoolExp = BOOL |(BoolExp)| RealRelExp | NOT BoolExp
      | BoolExp BOOL_OP
RealRelExp = RealExp REL_OP RealExp
        | COND_REAL_FUNC(RealExp)
RealExp = IdRef |(RealExp)| RealExp OPER RealExp
IdRef = CellRef OptPortName | Constant | Function
    | UNDEF | PORTREF(PORTNAME)| STVAR_NAME
    | CELLPOS(RealExp)| SEND(PORTNAME,RealExp)
OptPortName = /* Empty */ |~PORTNAME
AssignSet = /* Empty */ | Assign AssignSet
Assign = STVAR_NAME ASSIGN_OP RealExp;
PortSendSet = /* Empty */ | PortSend PortSendSet
PortSend = SEND(PORTNAME,RealExp) ;
      |~PORTNAME ASSIGN_OP RealExp;
Constant = INT | REAL | CONSTFUNC
Function = COUNT
    | STATECOUNT(RealExp OptParamPort)
    | UNARY_FUNC(RealExp)
    | BINARY_FUNC(RealExp,RealExp)
    | WITHOUT_PARAM_FUNC_TIME
    | WITHOUT_PARAM_FUNC_RANDOM
    | UNARY_FUNC_RANDOM(RealExp)
    | BINARY_FUNC_RANDOM(RealExp,RealExp)
    | COND3_FUNC(BoolExp,RealExp,RealExp)
    | COND4_FUNC(BoolExp,RealExp,RealExp,RealExp)
OptParamPort = /* Empty */ |, ~PORTNAME
CellRef =(Tuple
Tuple = INT,INT Rest_nTuple
Rest_nTuple =,INT Rest_nTuple |)
BOOL =t|f|?
REL_OP =!=|=| > | < | >=| <=
BOOL_OP =and|or|xor|imp|eqv
ASSIGN_OP =:=
OPER = + |-|*|/
INT =[ SIGN] DIGIT{ DIGIT}
REAL = INT | [ SIGN] { DIGIT} .DIGIT{ DIGIT}
SIGN = + |-
DIGIT =0|1|2|3|4|5|6|7|8|9
PORTNAME =thisPort| STRING
STVAR_NAME =$STRING
STRING = LETTER{ LETTER}
LETTER =a|b|c|...|z|A|B|C|...|Z
CONSTFUNC = pi | e | inf | grav | accel | light
    | planck | avogadro | faraday | rydberg | golden
    | euler_gamma | bohr_radius | boltzmann | amu
    | bohr_magneton | catalan | electron_charge | pem
    | ideal_gas | stefan_boltzmann | proton_mass
    | electron_mass | neutron_mass
WITHOUT_PARAM_FUNC = truecount | falsecount
    | undefcount | time | random | randomSign
UNARY_FUNC= abs | acos | acosh | asin | asinh | atan
    | atanh | cos | sec | sech | exp | cosh | fact | ln
    | fractional | log | round | cotan | cosech | cosec
    | sign | sin | sinh | statecount | sqrt | tan | tanh
    | trunc | truncUpper | poisson | chi | exponential
    | randInt | acotan | acosech | acotanh | asech
    | asec | nextPrime | radToDeg | degToRad | nth_prime
    | CtoF | CtoK | KtoC | KtoF | FtoC | FtoK
BINARY_FUNC = comb | logn | max | min | power | root
    | remainder | beta | gamma | lcm | gcd | uniform
    | normal | f | binomial | rectToPolar_r | hip
    | rectToPolar_angle | polarToRect_x | polarToRect_y
COND_REAL_FUNC = even | odd | isInt | isPrime | isUndefined
```

9. References

1. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation*. Academic Press, 2001.
2. Von Neumann J and Burks AW. *Theory of self-reproducing automata*. University of Illinois Press, 1966.
3. Wainer G and Giambiasi N. Application of the Cell-DEVS paradigm for cell spaces modeling and simulation. *Simulation* 2001; 71: 22–39.
4. Wainer G and Giambiasi N. N-Dimensional Cell-DEVS. *Discrete Event Syst Theory Appl* 2002; 12: 135–157.
5. Wainer G and Giambiasi N. Timed Cell-DEVS: modeling and simulation of cell spaces. In: Sarjoughian H and Cellier F (eds) *Discrete event modeling & simulation: enabling future technologies*. Springer-Verlag, 2001.
6. Wainer G and Giambiasi N. Cell-DEVS models with transport and inertial delays. In: *Proceedings of the SCS European simulation symposium*, Passau, Germany 1996.
7. Goldstein R, Khan A, Dalle O, et al. Multiscale representation of simulated time. *Simulation*. Epub ahead of print 28 September 2017. DOI: 10.1177/0037549717726868.
8. Wainer G. *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2009.
9. Wainer G and Castro R. A survey on the application of the Cell-DEVS formalism in cellular models. *J Cell Automata* 2010; 5: 509–524.
10. Wainer G and Fernández J. Modeling and simulation of complex cellular automata using Cell-DEVS. *Simulation* 2016; 92: 101–115.
11. Wainer G, Liu Q, Dalle O, et al. Applying cellular automata and DEVS methodologies to digital games. *Simul Gaming* 2010; 41: 796–823.
12. Lo Tártaro M, Torres C and Wainer G. Defining models of urban traffic using the TSC tool. In: *Proceedings of 2001 winter simulation conference*, Arlington, VA, 2001. New York: IEEE Press.
13. Wainer G. ATLAS: a specification language for traffic modelling and simulation. *Simul Model Pract Theory*. 2006; 14: 317–337.
14. Wainer G and Edwards K. GATLAS: Google Earth visualization for atlas. In: *Proceedings of 2011 SCS/ACM symposium on theory of modeling and simulation (TMS/DEVS'11)*, Boston, MA, 2011.
15. Wainer G and Davidson A. Defining a traffic modeling language using cellular discrete-event abstractions. *J Cell Automata* 2007; 2: 291–343.
16. Wainer G. Developing a software tool for urban traffic modeling. *Software Pract Experience* 2007; 37: 1377–1404.
17. Ameghino J, Troccoli A and Wainer G. Models of complex physical systems using Cell-DEVS. In: *Proceedings of 34th IEEE/SCS annual simulation symposium*, Seattle, WA, 2001.
18. Saadawi H and Wainer G. Modeling complex physical systems using 2D finite element Cell-DEVS. In: *Proceedings of spring simulation conference 2004 (ASTC'04)*, Arlington, VA, 2004.
19. Ding W, Lin C, Chechiu L, et al. Definition of Cell-DEVS models for complex diffusion systems. In: *Proceedings of the 2005 SCS summer computer simulation conference*, Philadelphia, PA, 2005.
20. Kazi B and Wainer G. Integrated cellular framework for modeling ecosystems: theory and applications. *Simulation*. Epub ahead of print 11 May 2017. DOI: 10.1177/0037549717706007.
21. MacLeod M, Chreyh R and Wainer G. Improved Cell-DEVS models for fire spreading analysis. *Lect Notes Comput Sci* 2006; 4173: 472–481.
22. Harzallah Y, Michel V, Liu Q, et al. Distributed simulation and web map mash-up for forest fire spread. In: *Proceedings of IEEE international conference on web services*, Honolulu, HI, 2008. New York: IEEE Press.
23. Wainer G. Applying Cell-DEVS methodology for modeling the environment. *Simulation* 2006; 82: 635–660.
24. Dahmani Y and El-Amine Hamri M. Event triggering estimation for Cell-DEVS: wildfire spread simulation case. In: *Fifth UKSim European symposium on computer modeling and simulation*, Madrid, 16–18 November 2011. Piscataway, NJ: IEEE.

25. Ntaimo L and Zeigler B. Expression of a forest cell model in parallel DEVS and timed Cell-DEVS formalisms. In: *Proceedings of the 2004 summer computer simulation conference*, San Jose, CA, 25–29 July 2004. San Diego, CA: Society for Computer Simulation.
26. Qela B, Wainer G and Mouftah H. Simulation of large wireless sensor networks using Cell-DEVS. In: *Proceedings of the winter simulation conference*, Austin, TX, 2009. New York: IEEE Press.
27. Moallemi M, El-Shabani M and Wainer G. Cellular simulation of asymmetric energy requirements in wireless sensor networks. In: *Proceedings of 2013 SCS/ACM summer computer simulation conference (SCSC'13)*, Toronto, ON, 2013.
28. Broutin E, Tavanpour M and Wainer G. Modelling wireless networks with the DEVS and Cell-DEVS formalisms. In: *Proceedings of winter simulation conference 2013*, Washington, DC, 2013.
29. Farooq U, Balya B and Wainer G. DEVS modeling of mobile wireless ad hoc networks. *Simul Model Pract Theory* 2007; 15: 285–314.
30. Freire V, Wang S and Wainer G. Visualization in 3ds Max for Cell-DEVS models based on moving entities. In: *Proceedings of the SCS/ACM symposium on simulation for architecture and urban design*, San Diego, CA, 2013.
31. Zhang C, Hammad A, Zayed T, et al. Cell-based representation and analysis of spatial resources in construction simulation. *J Autom Constr* 2007; 16: 436–448.
32. Zhang C, Hammad A, Zayed TM, et al. Simulation of the redecking of Jacques Cartier Bridge considering spatial constraints. In: *Proceedings of 7th international conference on short and medium span bridges 2006*, Montreal, QC, 2006.
33. Pang H, Zhang C and Hammad A. Sensitivity analysis of construction simulation using Cell-DEVS and microcyclone. In: *Proceedings of the winter simulation conference*, 2006. New York: IEEE.
34. Wang S, Wainer G, Rajus V, et al. Occupancy analysis using building information modeling and Cell-DEVS simulation. In: *Proceedings of 2013 SCS/ACM/IEEE symposium on theory of modeling and simulation (TMS/DEVS'13)*, San Diego, CA, 2013.
35. Shang H and Wainer G. A model of virus spreading using Cell-DEVS. *Lect Notes Comput Sci* 2005; 3515: 373–377.
36. Boiko Y and Wainer G. Modeling a neural decoder based on spiking neurons in DEVS. In: *Proceedings of SCS/ACM Springsim 2009 (DEVS symposium)*, San Diego, CA, 22–27 March 2009. San Diego, CA: Society for Computer Simulation International.
37. Goldstein R, Wainer G, Cheetham J, et al. Vesicle–synapsin interactions modeled with Cell-DEVS. In: *Proceedings of winter simulation conference*, Miami, FL, 2008. New York: IEEE Press.
38. Wainer G and Goldstein R. Modelling tumor-immune systems with Cell-DEVS. In: *Proceedings of the European modeling and simulation conference 2008*, Nicosia, Cyprus, 2008.
39. Goldstein R and Wainer G. Hierarchical biological DEVS model design. *IEEE/AIP Comput Sci Eng* 2016; 17: 72–82.
40. Holman K, Kuzub J and Wainer G. UAV search strategies using Cell-DEVS, In: *Proceedings of 2010 annual simulation symposium*, Orlando, FL, 11–15 April 2010. San Diego, CA: Society for Computer Simulation International.
41. Castonguay P and Wainer G. Aircraft evacuation DEVS implementation & visualization. In: *Proceedings of SCS/ACM Springsim 2009 (DEVS symposium)*, San Diego, CA, 22–27 March 2009. San Diego, CA: Society for Computer Simulation International.
42. Ha S. Cell-based evacuation simulation considering human behavior in a passenger ship. *Ocean Eng* 2012; 53: 138–152.
43. Lawler R and Jafer S. Egress modeling with cellular discrete event system. In: *SoutheastCon 2015*, Fort Lauderdale, FL, 9–12 April 2015. Piscataway, NJ: IEEE.
44. Jafer S and Lawler R. Emergency crowd evacuation modeling and simulation framework with cellular discrete event systems. *Simulation* 2016; 92: 795–817.
45. Madhoun R and Wainer G. Creating spatially-shaped defense models using DEVS and Cell-DEVS. *J Defense Model Simul* 2005; 2: 121–143.
46. Hosang E and Wainer G. An architecture to facilitate interoperability of DEVS and C-BML simulation models. *J Defense Model Simul* 2016; 13: 1–23.
47. Cao Q, He ZS and Yu L. Research into simulation and modeling of material supply in emergent disaster based on DEVS/CD ++ . *J Comput Appl* 2008; 28: 2967–2969.
48. Bouanan Y and Zacharewicz G. Using DEVS and Cell-DEVS for modelling of information impact on individuals in social network. In: *IFIP international conference on advances in production management systems (APMS)*, 2014. Berlin: Springer.
49. Quesnel G, Duboz R and Ramat E. The virtual laboratory environment: an operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simul Modell Pract Theory* 2009; 17: 641–653.
50. Quesnel G, Vermisse D and Ramat E. Coupling of physical models and social models: multi-modeling and simulation with VLE. In: *Joint conference on multi-agent modelling for environmental management (CABM-HEMA-SMAGET'05)*, Bourg Saint Maurice, France, 21–25 March 2005.
51. Al-Habashna A and Wainer G. Modeling pedestrian behavior with Cell-DEVS: theory and applications. *Simulation* 2016; 92: 117–139.
52. Wang S and Wainer G. A simulation as a service methodology with application for crowd modeling, simulation and visualization. *Simulation* 2015; 91: 71–95.
53. Van Schyndel M, Hesham O, Wainer G, et al. Crowd modeling in the Sun Life building. In: *Proceedings of symposium on simulation in architecture and urban design (SimAUD 2016)*, London, UK, 4–7 June 2016. San Diego, CA: The Society for Modeling and Simulation International.
54. Farrell R, Moallemi M, Wang S, et al. Modeling and simulation of crowd using cellular discrete event systems theory. In: *Proceedings of ACM sigsim conference on principles of advanced discrete simulation (PADS)*, Montréal, QC, 19–22 May 2013. New York, NY: ACM.
55. Morris H and Wainer G. Music generation using cellular models. In: *Proceedings of 2012 SCS/ACM/IEEE symposium on theory of modeling and simulation*, Orlando, FL, 26–29 March 2012. San Diego, CA: Society for Computer Simulation International.

56. Moallemi M and Wainer G. Content-based image recognition with cellular discrete event system specifications. In: *Proceedings of annual simulation symposium (SCS/ACM)*, Orlando, FL, 26–29 March 2012. San Diego, CA: Society for Computer Simulation International.
57. Liu Q and Wainer G. Simulating market dynamics with CD ++ . *Lect Notes Comput Sci* 2005; 3515: 368–372.
58. Wainer G and Zeigler B. Experimental results of timed Cell-DEVS quantization. In: *Proceedings of AIS'2000 artificial intelligence, simulation and planning*, Tucson, AZ, 2000.
59. Wainer G. The Cell-DEVS formalism as a method for activity tracking in spatial modeling and simulation. *Int J Process Model Simul* 2015; 10: 19–38.
60. Wainer G and Giambiasi N. Using G-DEVS and Cell-DEVS to model complex continuous systems. *Simulation* 2005; 81: 137–151.
61. Kofman E. A second-order approximation for DEVS simulation of continuous systems. *Simulation* 2002; 78: 76–89.
62. Van Schyndel M, Wainer G, Goldstein R, et al. On the definition of a computational fluid dynamic solver using cellular discrete-event simulation. *J Comput Sci* 2014; 5: 882–890.
63. Saadawi H and Wainer G. Modeling physical systems using finite element Cell-DEVS. *Simul Modell Pract Theory* 2007; 15: 1268–1291.
64. Jafer S and Wainer G. Conservative DEVS: a novel protocol for parallel conservative simulation of DEVS and Cell-DEVS models. In: *Proceedings of 2010 SCS/ACM symposium on theory of modeling and simulation (DEVS'10)*, Orlando, FL, 12–15 April 2010. San Diego, CA: The Society for Modeling and Simulation International.
65. Jafer S and Wainer G. Global lookahead management (GLM) protocol for conservative DEVS simulation. In: *Procedings of IEEE/ACM international symposium on distributed simulation and real-time applications (DS-RT 2010)*, Washington, DC, 2010. New Yotk: IEEE Press.
66. Liu Q and Wainer G. Parallel environment for DEVS and Cell-DEVS models. *Simulation* 2007; 83: 449–471.
67. Liu Q and Wainer G. Multicore acceleration of discrete event system specification systems. *Simulation* 2012; 88: 801–831.
68. Liu Q and Wainer G. Exploring multi-grained parallelism in compute-intensive DEVS simulations. In: *Proceedings of the 24th ACM/IEEE/SCS workshop on principles of advanced and distributed simulation (PADS 2010)*, Atlanta, GA, 17–19 May 2010. Piscataway, NJ: IEEE.
69. Liu Q and Wainer G. A Performance evaluation of the lightweight time warp protocol in optimistic parallel simulation of DEVS-based environmental models. In: *Proceedings of 23rd workshop on principles of advanced and distributed simulation*, Lake Placid, NY, Date? 2009. New York: IEEE Press.
70. Mancini E, Wainer G, Al-Zoubi K, et al. Simulation in the cloud using handheld devices. In: *MSGC'12 workshop (CCGRID 2012)*, Ottawa, ON, 13–16 May 2012. Piscataway, NJ: IEEE.
71. Al-Zoubi K and Wainer G. Using REST web-services architecture for distributed simulation. In: *Proceedings of 23rd workshop on principles of advanced and distributed simulation*, Lake Placid, NY, 22–25 June 2009. New York: IEEE Press.
72. Al-Zoubi K and Wainer G. RISE: a general simulation interoperability middleware container. *J Parallel Distrib Comput* 2013; 73: 580–594.
73. Bonaventura M, Wainer G and Castro R. A graphical modeling and simulation environment for DEVS. *Simulation* 2013; 89: 4–27.
74. López A and Wainer G. Improved Cell-DEVS model definition in CD ++ . *Lect Notes Comput Sci* 2004; 3305: 803–812.
75. Goldstein R, Breslav S and Khan A. Practical aspects of the DesignDEVS simulation environment. *Simulation*. Epub ahead of print 18 August 2017. DOI: 10.1177/0037549717718258.
76. Cardoen B, Manhaeve S, Van Tendeloo Y, et al. A PDEVS simulator supporting multiple synchronization protocols: implementation and performance analysis. *Simulation*. Epub ahead of print 23 February 2017. DOI: 10.1177/0037549717690826.
77. Risco-Martín J-L, Mittal S, Fabero Jiménez J-C, et al. Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simulation* 2017; 93: 459–476.
78. Seo K-M, Hong W and Kim T-G. Enhancing model composability and reusability for entity-level combat simulation: a conceptual modeling approach. *Simulation* 2017; 93: 825–840.
79. Djitog I, Aliyu H and Traoré MK A model-driven framework for multi-paradigm modeling and holistic simulation of healthcare systems. *Simulation*. Epub ahead of print 8 December 2017. DOI: 10.1177/0037549717744888.
80. Gu F. Localized recursive spatial-temporal state quantification method for data assimilation of wildfire spread simulation. *Simulation* 2017; 93: 343–360.
81. Oda S, Iyori K, Ken M, et al. The application of cellular automata to the consumer's theory: simulating a duopolistic market. In: *Asia-Pacific conference on simulated evolution and learning*, 1999. Berlin: Springer-Verlag.
82. Martín del Rey A. A computer virus spread model based on cellular automata of graphs. *Lect Notes Comput Sci* 2009; 5518: 503–506.
83. Peng S, Wang G and Yu S. Modeling the dynamics of worm propagation using two-dimensional cellular automata in smartphones. *J Comput Syst Sci* 2013; 79: 586–595.
84. Martín del Rey A. A SIR e-Epidemic model for computer worms based on cellular automata. *Lect Notes Artif Intell* 2013; 8109: 228–238.
85. Huang G and Liu X. Simulation of worm viruses spread in network based on cellular automata. *Comput Eng* 2009; 35; 168–169.

Author biography

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 the 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 the author of three books and over 360 research articles; he edited four other books, and helped organizing numerous conferences, including being one of the founders of the Symposium on Theory of Modeling and Simulation, SIMUTools and SimAUD. Prof. Wainer was Vice-President Conferences, Vice-President Publications, and 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.