Improved Cell-DEVS model definition in CD++

Alejandro López [1], Gabriel Wainer [2]

[1]Computer Science Department. Universidad de Buenos Aires
Ciudad Universitaria (1428). Buenos Aires. Argentina.
[2]Department of Systems and Computer Engineering. Carleton University
1125 Colonel By Dr. Ottawa, ON. K1S 5B6. Canada.

Abstract. We describe two improvements made to CD++, a tool for modeling and simulation of cellular models based on the Cell-DEVS formalism. The modifications described in this work remove some limitations existing in the previous implementation. These modifications allow the cells to use multiple state variables and multiple ports for inter-cell communication. The cellular model specification language has been extended to cover these cases, making CD++ a more powerful tool.

Introduction

The Cell-DEVS formalism [1] was defined as an extension to Cellular Automata combined with DEVS (Discrete Event systems Specification) [2], a formalism for specification of discrete-event models. The DEVS formalism provides a framework for the construction of hierarchical modular models, allowing for model reuse, reducing development and testing times. 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. A DEVS atomic model defined as:

$$DEVS = < X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta>$$

In the absence of external events, a DEVS model will remain in state $s \in S$ during $ta(s)$. Transitions that occur due to the expiration of $ta(s)$ are called internal transitions. When an internal transition takes place, the system outputs the value $\lambda(s) \in Y$, and changes to the state defined by $\delta_{int}(s)$. Upon reception of an external event, $\delta_{ext}(s, e, x)$ is activated using the input value $x \in X$, the current state s and the time elapsed since the last transition e. Coupled models are defined as:

$$DN = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}>$$

Coupled models consist of a set of basic models (*Mi*, atomic or coupled) connected through the models' interfaces. Component identifications are stored into an index (*D*). A translation function (*Zij*) is defined by using an index of influencees created for each model (*Ii*). The function defines which outputs of model *Mi* are connected to inputs in model *Mj*.

Cell-DEVS defines a cell as a DEVS atomic model and a cell space as a coupled model. Each cell of a Cell-DEVS model holds a state variable and a computing function, which updates the cell state by using its present state and its neighborhood. A Cell-DEVS atomic model is defined as:

$$TDC = < X, Y, S, N, delay, d, \delta_{INT}, \delta_{EXT}, \tau, \lambda, D >$$

A cell uses a set of *N* input values to compute its future state, which is obtained by applying the local function τ. A delay function is associated with each cell, after which, the new state value is sent out. There are two types of delays: inertial and transport. When a transport delay is used, every value scheduled for output will be transmitted. Inertial delays use a preemptive policy: any previous scheduled output will be preempted unless its value is the same as the new computed one. After the basic behavior for a cell is defined, a complete cell space can be built as a coupled Cell-DEVS:

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

A coupled Cell-DEVS is composed of an array of atomic cells (C), each of which is connected to the cells in the neighborhood (N). The border cells (B) can have a different behavior than the rest of the space. The Z function defines the internal and external coupling of cells in the model. This function translates the outputs of mth output port in cell C_{ij} into values for the mth input port of cell C_{kl}. Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood. The Xlist and Ylist are used for defining the coupling with external models.

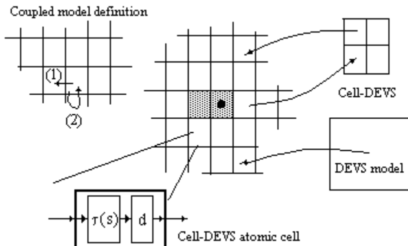

Fig. 1. Informal definition of a Cell-DEVS model [3]

Using Cell-DEVS has different advantages. First, we have asynchronous model execution, which, as showed in [3], results in improved execution times. Timing constructions permit defining complex conditions for the cells in a simple fashion, as showed in [4, 5]. As DEVS models are closed under coupling, seamless integration with other types of models in different formalisms is possible. The independent simulation mechanisms permit these models to be executed interchangeably in single-processor, parallel or real-time simulators without any changes.

The CD++ tool [6] was created for the simulation of DEVS and Cell-DEVS models based on the formal specifications in [1]. This version of the tool and the formalism was used to study a variety of models including traffic, forest fires, biological systems and experiments in chemistry [4, 5, 7, 8]. While developing these models, we found that Cell-DEVS formal specifications and CD++ implementation lacked on expressiveness when defining very complex applications. We were able to solve some

of these problems with the n-dimensional Cell-DEVS definition presented in [9], which permitted us to store different state variables in each dimension of a cell space. Nevertheless, as this organization require extra work from the modelers, and is time-consuming, we decided to add new facilities to CD++. We will show how to define different state variables for each cell, and to declare and use multiple inter-cell ports to communicate with the neighbors, which permitted to improve the definition of complex models.

Other cellular automata languages as CARPET [10] and Cellang [11] include these features which are new to CD++. However, these languages use different approaches from the one used in CD++. CARPET is a procedural language while CD++ is logical (as Prolog is) and Cellang is a mix. With the new extensions to CD++, it is leveraged in capabilities to these two languages.

The CD++ toolkit

CD++ [6] is a tool built to implement DEVS and Cell-DEVS theory. The tool allows defining models according to the specifications introduced in the previous section. DEVS atomic models can be incorporated into a class hierarchy in C++, while coupled models are defined using a built-in specification language. The tool also includes an interpreter for a specification language that allows describing Cell-DEVS models.

The behavior specification of a Cell-DEVS atomic model 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. The behavior of the local computing function is defined using a set of rules with the form: `VALUE DELAY { CONDITION }`. These indicate that when the `CONDITION` is satisfied, the state of the cell changes to the designated `VALUE`, and its output is `DELAY`ed for the specified time. The main operators available to define rules and delays include: boolean, comparison, arithmetic, neighborhood values, time, conditionals, angle conversion, pseudo-random numbers, error rounding and constants (i.e., gravitation, acceleration, light, Planck, etc.) [12].

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

[tau-function]
rule: 1 100 {(0,0)=1 and (truecount=8 or truecount=10)}
rule: 1 200 {(0,0) = 0 and truecount >= 10 }
rule: (0,0) 150 { t }
```

Fig. 2. A Cell-DEVS specification in CD++

Figure 2 shows the specification of a Cell-DEVS model in CD++. The specification follows Cell-DEVS coupled model's formal definitions. In this case, *Xlist* = *Ylist*

= { ∅ }. The set {m, n} is defined by *width-height*, which specifies the size of the cell space (in this example, m=20, n=40). The N set is defined by the lines starting with the *neighbors* keyword. The border B is wrapped. Using this information, the tool builds a cell space, and the Z translation function following Cell-DEVS specifications. 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 in N is 8 or 10, the cell state remain in 1. This state change is spread to the neighboring cells after 100 ms. The second rule states that, whenever a cell state is 0 and the sum of the inputs is larger or equal to 10, the cell value changes to 1. In any other case (t = true), the result remains unchanged, and it is spread to the neighbors after 150 ms.

Expanding CD++ Architecture

CD++ only supported one state variable per cell. To work around this problem, modelers usually defined extra planes in their cell space, creating as many layers as state variables needed. For instance, when one state variable was needed in a planar cell space, the solution was to create a three-dimensional cell space with two planar layers [7]. State variables are declared as follows:

```
StateVariables: pend temp vol
StateValues: 3.4 22 -5.2
InitialVariablesValue: initial.var
```

The first line declares the list of state variables for every cell, the second one the default initial values, and the last one provides the name of a file where the initial values for some particular cells are stored, using the following format:

```
(0,0,1) = 2.8 21.5 -6.7
(2,3,7) = 6 20.1 8
```

The values are assigned to the state variables following the order in which they are listed in the sentence *StateVariables*. Here, the first line assigns 2.8 to *pend*, 21.5 to *temp*, and -6.7 to *vol* in the cell (0,0,1). The second line will assign respectively the values 6, 20.1 and 8, to the variables *pend*, *temp* and *vol* in the cell (2,3,7).

State variables can be referenced from within the rules that define the cells' behavior by using its name preceded by a **$**.

```
rule: {(0,0,0)+$pend} 10 { (0,0,0)>4.5 and $vol<22.3 }
```

The identifier '**:=**' is used to assign values to a state variable. Assignments must be placed in a new section in the rules (a list of assignments, separated by semi-colons).

```
% <value> [ { <assignments> } ] <delay> <condition>
rule: { (0,0,0)+1 } { $temp:=$vol/2; $pend:=(0,1,0); }
              10 { (0,1,0) > 5.5 }
```

In the example, if the condition *(0,1,0)>5.5* is true, the variable *temp* will be assigned half of *vol* value, and *pend* will be assigned the value of the neighbor cell (0,1,0). These assignments are executed immediately (they are not delayed).

A second limitation was that the previous implementation of CD++ only used one port for inputs (*neighborChange*) and one for outputs (*out*), which are automatically created with the cell. The use of different I/O ports provides a more flexible definition of the cell behavior, as the rules can react in different ways according to inputs arriving in different ports. Therefore, our second extension supports the use of multiple I/O ports, which are defined as follows:

```
NeighborPorts: alarm weight number
```

The input and output ports share the names, making possible to automatically create the translation function: an output from a cell will influence exclusively the input port with the same name in every cell in its neighborhood. When a cell outputs a value through one of the ports, it will be received by all its neighbors through the input ports with the same name. A cell can read the value sent by one of its neighbors, specifying both the cell and port name, separated by a tilde (~), as follows:

```
rule : 1 100 { (0,1)~alarm != 0 }
```

In this case, if the cell receives an input in the *alarm* port from the cell to the right, and that value is not 0, the cell status will change to 1, and this change will be sent through the default output port 100 ms after. 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 semi-colon), as follows:

```
% <port_assigns> [ <assignments> ] <delay> <condition>
rule: { ~alarm := 1; ~weight := (0,-1)~weight; } 100
            { (0,1)~number > 50 }
```

In this example, if we receive a value larger than 50 in the port *number* on the cell to the right, we wait 100 ms, we generate an output of 1 in the *alarm* port, and we copy the *weight* value received from the cell to the left to the *weight* output port.

The rules defining the models are translated into an executable definition. Each of the rules is represented with a structure <value, assignations, delay, condition>, each represented by a tree. Rules are evaluated recursively form the tree that represents the condition. If the result of the evaluation is *True*, it then evaluates the trees corresponding to the value and the delay, and the result of these evaluations are the values used by the cell. The complete language grammar and details about the tool implementation are described in [13].

Using the extensions in a model of fire spread

In [14], we described a fire model using Cell-DEVS. Figure 3 represents the specification of this model in CD++.

```
[ForestFire]
dim : (100,100,2)    border : nowrapped
neighbors : (-1,0,0) (0,-1,0) (1,0,0)
neighbors : (0,1,0)(0,0,0)(0,0,-1)(0,0,1)
zone : ti { (0,0,1)..(99,99,1) }
localTransition : FireBehavior

[ti]
rule:{ time/100 } 1 { cellpos(2)=1 AND (0,0,-1)>=573
             AND (0,0,0) = 1.0 }

[FireBehavior]
rule: {#unburned} 1 {(0,0,0)<300 AND (0,0,0)!=26
     AND (#unburned>(0,0,0) OR time<=20)} %Unburned
rule: {#burning} 1 {cellpos(2)=0 AND ( ((0,0,0) >
    #burning AND (0,0,0)>333) OR (#burning> (0,0,0)
       AND (0,0,0)>=573) ) AND (0,0,0)!=209 } %Burning
rule: {26} 1 { (0,0,0)<=60 AND (0,0,0)!=26 AND
         (0,0,0)>#burning } %Burned
rule : { (0,0,0) } 1 { t }   %Stay Burned or constant

#BeginMacro(unburned)
(0.98689 * (0,0,0) + 0.0031 * ( (0,-1,0) + (0,1,0) +
(1,0,0) + (-1,0,0) ) + 0.213 )
#EndMacro

#BeginMacro(burning)
(0.98689*(0,0,0)+.0031*((0,-1,0)+(0,1,0)+(1,0,0)+
  (-1,0,0))+2.74*exp(-.19*((time+1)*.01-(0,0,1)))+.213)
#EndMacro
```

Fig. 3. Fire spread model specification.

We first define the Cell-DEVS coupled model (including neighborhood, dimension, etc.). Then, the *ti* rules show how to store ignition times: if a cell in plane 0 starts to burn, we record the current simulation time in plane 1. To make this happen, we include a clause specifying to identify the layer in which the current cell is located (*cellpos(2)=x*). Then, we show the rules used to compute the cells' temperatures. The model specification was simplified using macros containing rules corresponding to the temperature calculus when the cell is in unburned or burning phase.

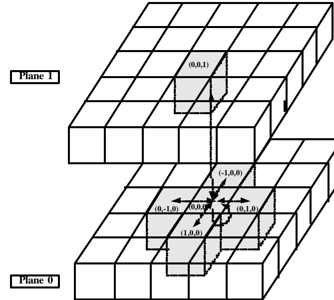

Fig. 4.. Cell's neighborhood specification

As described in Figure 4, we used two planes representing the different variables in our model. The first plane stores the cell's temperature. The second plane stores the ignition time of the cells in the propagation plane.

In the example, we need n *x* m *x* 2 cells (double the size of simulated field). Using the new simulator, the *temperature* is stored as the cell's value and the *ignition time* in a state variable (*temperature* is stored in the cell's value because it must be passed to the neighbor cells, while the *ti* value is only used internally to the cell). The first step was to add a state variable *ti*, to remove the higher layer of cells and to replace all the references to this layer by references to the state variable. The original *burning* and *ti* rules in Figure 5 were replaced as follows:

```
rule: {#burning} 1 {((0,0)>#burning AND (0,0)>333) OR
      (#burning>(0,0) AND (0,0)>=573) AND (0,0)!=209}

rule : { (0,0) } { $ti := time/100; } 1 { (0,0)>=573
           AND $ti = 1.0 }
```

The direct translation has a problem: in some cases, both conditions can be true at the same time. For instance, when \$ti = 1.0, (0,0) >=573 and #burning > (0,0), both rules apply. In order to solve this problem, the *burning* rule was factorized into two simpler rules, as follows:

```
rule : { #burning } 1 { (0,0) > #burning AND (0,0)> 333
           AND (0,0) != 209 }
rule : { #burning } 1 { #burning > (0,0) AND (0,0)>=
           573 AND (0,0)!= 209 }
```

We can see that in both rules we need (0,0) != 209, and (0,0) > 333 and $(0,0) \geq 333 \Rightarrow (0,0) \neq 209$. Hence, (0,0) != 209 is redundant, and so it can be removed. Rule *ti* overlaps with the second part of the rule *burning*, so they were merged. To shorten the execution time, the number of rules was reduced and the clauses in the rules' condition reordered. The two rules were merged in one rule that will assign the new value to *\$ti* depending on *\$ti*'s original value:

```
rule : { #burning } 1 { (0,0)> 333 AND  ( (0,0)< 573 OR
```

```
            $ti != 1.0) AND (0,0)>#burning }
rule : { #burning }{ $ti := if($ti = 1.0, time/100,
            $ti); } 1{ (0,0)>=573 AND #burning>=(0,0) }
rule : { #burning } { $ti := time / 100; } 1 { $ti=1.0
            AND (0,0)>=573 AND #burning<(0,0) }
```

A second step optimization is based on the fact that CD++ is capable of using short-cut evaluation (in the same style as the C programming language). When the left expression of an **and** operation evaluates to false, the whole operation will evaluate to *false*, so it is useless to evaluate the right expression. Similarly, when the left expression of an **or** operation evaluates to *true*, the whole operation will evaluate to true, and so there is no need to evaluate the right expression of the operation. By simply reordering the operations and their parameters, we can save execution time. The idea is to execute the simplest conditions first, while leaving the more complex ones to the end.

```
%Unburned
rule : { #macro(unburned) } 1
       { (0,0) != 209 AND (0,0) < 573 AND
         ( time <= 20 OR #macro(unburned) > (0,0) ) }
%Burning and ti
rule : { #macro(burning) } 1
       { (0,0) > 333 AND ( (0,0) < 573 OR $ti != 1.0 )
         AND (0,0) > #macro(burning) }
rule : { #macro(burning) }
       { $ti := if($ti = 1.0, time / 100, $ti); } 1
       { (0,0) >= 573 AND #macro(burning) >= (0,0) }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { $ti = 1.0 AND (0,0) >= 573 AND
         #macro(burning) < (0,0) }
%Burned
rule : { 209 } 100
       { (0,0) != 209 AND (0,0) <= 333 AND
         (0,0) > #macro(burning) }
%Stay Burned or constant
rule : { (0,0) } 1 { t }
```

In the FireSpread model the cells can be in one of four phases: inactive, unburned, burning and burned. An unburned cell's temperature is lower than 573 degrees. A cell starts burning at 573 degrees and its temperature increases for a while and then start decreasing as the fuel mass is consumed. When the temperature gets lower than 333 degrees, the cell enters the burned phase. In the simulation this is signaled by a constant temperature of 209 degrees.

The first rule applies to unburned cells, whose temperature in the next step will be higher than its current one or the simulation time is smaller than twenty (transient period). The second rule is based on the same principles but applies to burning cells. The third rule is used when the cells start burning to modify the temperature and the ignition time (depending on its value). Fourth rule updates the ignition time and the

temperature of burning cells when their temperature is decreasing. The fifth rule sets the burned flag (temperature equals 209 degrees) when a burning cell crosses down the 333-degrees threshold, and the sixth rule keeps the burned cells constant.

This problem can also be solved using multiple ports to replace the extra plane. When we use multiple ports we do not need to store internally the values, but to transmit them through the ports. So, there is not need to set values, but just send them out though the port. In this case, two ports are declared: temp and ti. The port temp exports the cell s temperature (the old lower layer), while the port ti exports the ignition time (the higher layer).

```
%Unburned
rule : { ~temp := #unburned; } 1
        { (0,0)~temp!=209 AND (0,0)~temp<573 AND
          (time<=20 OR #unburned>(0,0)~temp ) }
%Burning and ti
rule : { ~temp := #burning; } 1 { (0,0)~temp>333 AND
          ( (0,0)~temp<573 OR (0,0)~ti!=1.0 ) AND
          (0,0)~temp > #burning }
rule : { #burning } 1
        { (0,0)> 333 AND ( (0,0)< 573 OR $ti != 1.0) AND
          (0,0)>#burning }
rule : { #burning }
        { $ti := if($ti = 1.0, time/100, $ti); } 1
        { (0,0)>=573 AND #burning>=(0,0) }
rule : { #burning } { $ti := time / 100; } 1
        { $ti=1.0 AND (0,0)>=573 AND #burning<(0,0) }
%Burned
rule : { ~temp := 209; } 100
        { (0,0)~temp > #macro(burning) AND
          (0,0)~temp <= 333 AND (0,0)~temp != 209 }
%Stay Burned or constant
rule : { } 1 { t }
```

This model behaves exactly the same as the previous one. As the initial value for both ports is the same and this model needs different values, it can solved by assigning negative initial value that will never appear during the simulation and adding two rules that generate the real initial state when the cell has this special values.

Conclusions

A new implementation of CD++ was presented, which allows using state variables and multiple neighbor ports in each cell of a cellular model. These new features add great power to the specification language to the simulator, simplifying the modeling task.

Cell-DEVS simplifies the construction of complex cellular models by allowing simple and intuitive model specification. The CD++ logic rules facilitate the debugging phase and, consequently, reduces development time. Complex model modifica-

tions can now be easily and quickly integrated to the current model of fire spread even by a non-computer specialist.

Models can now be written more clearly, and their simulation consume less memory and file descriptors (than those models with extra cell layers), which allow for larger cell spaces to be simulated.

References

1. Wainer, G.; Giambiasi, N., 2000. "Timed Cell-DEVS: modelling and simulation of cell spaces". *Discrete Event Modeling & Simulation: Enabling Future Technologies*, to be published by Springer-Verlag.
2. Zeigler, B.; Kim, T.; Praehofer, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.
3. Wainer, G.; Giambiasi, N. "Application of the Cell-DEVS paradigm for cell spaces modelling and simulation.". *Simulation*; Vol. 76, No. 1. January 2001.
4. Ameghino, J.; Troccoli, A.; Wainer, G. "Modeling and simulation of complex physical systems using Cell-DEVS". *Proceedings of 34th IEEE/SCS Annual Simulation Symposium*. Seattle, U.S.A. 2001.
5. Muzy, A.; Wainer, G.; Innocenti, E.; Aiello, A.; Santucci, J.-F. "Dynamic and discrete quantization for simulation time improvement: fire spreading application using the CD++ tool". *Proceedings of 2002 Winter Simulation Conference*. San Diego, U.S.A. 2002.
6. Wainer, G. "CD++: a toolkit to define discrete-event models". *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.
7. Ameghino, J.; Wainer, G. "Application of the Cell-DEVS paradigm using CD++". *Proceedings of the 32nd SCS Summer Computer Simulation Conference*. Vancouver, Canada. 2000.
8. Ameghino, J.; Troccoli, A.; Wainer, G. "Applying Cell-DEVS Models of Complex Systems". *Proceedings of Summer Simulation Multiconference. Montreal,* QC. Canada. 2003.
9. Wainer, G.; Giambiasi, N. "N-dimensional Cell-DEVS". *Discrete Events Systems: Theory and Applications*, Kluwer, Vol. 12 N° 1, January 2002. pp 135-157.
10. Spezzano, G.; Talia, D. "CARPET: A Programming Language for Parallel Cellular Processing". In *Proceedings of the European School on Parallel Programming Environments for HPC'96*. April 1996.
11. Eckart, D. "A Cellular Automata Simulation System". SIGPLAN Notices, 26(8):80--85, August 1991.
12. D. Rodríguez, G. Wainer. "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Multiconference on Computer Simulation*. 1-7. 1999.
13. López, A.; Wainer, G. "Extending CD++ for Cell-DEVS model definition". Technical Report SCE-04-11. Dept. of Systems and Computer Engineering. Carleton University. 2004.
14. Muzy, A.; Wainer, G.; Innocenti, E.; Aiello, A.; Santucci, J.F. "Cellular Discrete-event modeling and simulation of fire spreading across a fuel bed". Accepted for publication in Simulation: Transactions of the Society for Modeling and Simulation International (Accepted: September 2003).