Modeling of Maze-Solving Systems using Cell-DEVS

Kevin Lam Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON. K1S 5B6. Canada.

Keywords: discrete event simulation, DEVS, cellular automata, maze solving, path planning

Abstract

The Cell-DEVS formalism was created as an extension of cellular automata for modeling complex systems using a discrete event formal approach. We examine the application of the Cell-DEVS formalism to a maze-solving application. The model uses the CD++ toolkit to model and simulate the proposed maze applications, showing that this approach permits solving complex applications by implementing them in a simple fashion.

1. INTRODUCTION

In recent years, computer simulation has played a key role in the study of artificial system. The cellular Automata (CA) formalism has recently gained popularity to describe some of these applications [1]. CA ~~are~~are defined as infinite n-dimensional lattices of cells whose values are updated according to a local rule. This is done simultaneously and synchronously using the current state of the cell and the state of a finite set of nearby cells (known as the neighborhood).

A popular application for cellular automata algorithms is maze solving, whether it is optimal path planning or obstacle avoidance. Nayfeh [2] describes a simple algorithm for finding solution paths through a two-dimensional maze. Tzionas, Thanailakis and Tsalides describe an algorithm (and hardware implementation) for collision-free path planning for a diamond-shaped robot [3]. Neither algorithm requires very complex computation or backtracking. Cellular automata lend themselves easily to fast, simple, and scalable implementations. The mazes effectively "solve themselves", in a linear time and with multiple parts of the maze solved simultaneously in parallel.

Typically, the maze, or space to be mapped, is described by a two-dimensional bitmap. Cells are marked to represent walls or free space. The cellular automata algorithm processes the bitmap and transforms it into one containing the solution set. Unfortunately, CA ~~have~~has showed to have different problems to model physical systems: they usually require large amounts of compute time, mainly due to their synchronous nature.

The Cell-DEVS formalism [4] solve these problems by using the DEVS (Discrete EVents Systems specifications) formalism [5] to define a cell space where each cell is defined as a DEVS model. This technique allows modeling of discrete-event cell spaces, improving their definition by making the timing specification more expressive. Besides this, discretizing the model into a bidimensional grid poses constraints on the precision that can be achieved by the model. Finite element analysis, instead, is able to provide higher precision due to the characteristics of the technique.

Here we present a definition of maze-solving algorithms using Cell-DEVS and their implementation using the CD++ toolkit. We show that complex applications like these ones can be easily implemented in our environment, permitting the user to focus on the modeling activities and letting them approach more complex applications using simple specification techniques.

2. BACKGROUND

The Cell-DEVS formalism is based on the use of the DEVS (Discrete EVents Systems specifications) formalism to define a cell space where each cell is defined as a DEVS model. The goal is to build discrete-event cell spaces, improving their definition by making the timing specification more expressive. DEVS formalism was proposed to model discrete events systems. A DEVS model is built using a set of behavioral models called **atomic** models, which can be combined to form **coupled** ones. In Cell-DEVS, each cell of a cellular model is defined as an atomic DEVS using transport or inertial delays.

Each cell is seen as having a set of N inputs to compute its future state. Each input (generally received from the neighboring cells) is received through the model's interface, and is used to activate the local function. A delay can be associated with each cell, allowing the deferred transmission of the execution results. A **transport** delay allows us to model a variable commuting time for each cell with anticipatory semantics (every scheduled event is executed). Using **inertial** delays, the semantics is preemptive: some scheduled events are not executed due to a small interval between two input events. Therefore, the outputs of a cell are not transmitted instantaneously, but after the consumption of the delay. The model advances through the activation of the internal, external, output and state's duration functions, as in other DEVS models.

Cell-DEVS atomic models are specified as:

$$TDC = < X, Y, S, N, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >.$$

Each cell will use the **N** inputs to compute the future state **S** using the function $\boldsymbol{\tau}$. The new value of the cell is transmitted to the neighbors after the consumption of the delay function. **Delay** defines the kind of delay for the cell, and **d** its duration. This behavior is defined by the $\boldsymbol{\delta_{int}}$, $\boldsymbol{\delta_{ext}}$, $\boldsymbol{\lambda}$ and **D** functions.

Once each cell is defined, they can be put together to form a coupled model composed of an array of atomic cells. Each of them is connected to its neighborhood. As the cell space is finite, the borders should be provided with a different behavior than the rest of the space. Otherwise, the space can be defined as wrapped, meaning that cells in a border are connected with those in the opposite one.

A Cell-DEVS coupled model is defined by:

$$GCC = < X_{list}, Y_{list}, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z >.$$

A cell space **C** defined by this specification is a coupled model composed by an array of atomic cells with size $\{\mathbf{t_1} \textbf{ x...x } \mathbf{t_n}\}$. Each cell in the space is connected to the cells defined by the neighborhood **N**. The cell space can be "wrapped", meaning that cells in a border are connected with those in the opposite one. Otherwise, the borders **B** should have a different behavior than the remaining cells. The **Z** function allows one to define the internal and external coupling of cells in the model. This function translates the outputs of output port m in cell Cij into values for the m input port of cell Ckl. The input/output coupling lists can be used to interchange data with other models.

The **CD++** tool [9] was developed following the definitions of the Cell-DEVS formalism CD++ is a tool to simulate both DEVS and Cell-DEVS models. Cell-DEVS models are described using a built-in specification language. The language provides a set of primitives to define the size of the cell-space, the type of borders, a cell's interface with other DEVS models and a cell's behavior. The behavior of a cell (the τ function of the formal specification) is defined using a set of rules of the form:

VALUE DELAY CONDITION

When an external event is received, the rule evaluation process is triggered to calculate the new cell value. Starting with the first listed rule, the CONDITION is evaluated. If it is satisfied, the new cell state is obtained by evaluating the VALUE expression. The cell will change to this new state after a DELAY time, and when it changes, it sends output messages to all its neighbors. If the condition is not valid, the next rule is evaluated repeating this process until a rule is satisfied. If no rule CONDITION statement is satisfied, the simulation is aborted.

3. A MAZE SOLVING ALGORITHM IN CELL-DEVS

In this section, we present a model simulating the algorithm proposed by Nayfeh [2]. The maze is represented as a two-dimensional cell array, with values of "1" and "0" representing walls and hallways (free cells). Each cell's neighbors consist of cells in the four cardinal directions North, East, South and West. Figure 1 represents a simple maze in a 10x10 cellular array, and an illustration of the cell's neighbors.

The maze is solved using cellular automata with the following rules for updating the cell's states:

- Wall cells always remains unchanged
- Free cells becomes a wall cell if its neighborhood

1	1	1	1	1	1	1	1	1	1
0	0	1	0	0	0	0	0	0	1
1	0	1	0	1	1	1	1	0	1
1	0	1	0	0	0	0	1	0	1
1	0	1	0	1	1	0	1	0	1
1	0	0	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1	1	1
1	0	1	0	0	0	0	0	0	1
1	0	1	0	1	0	1	1	0	1
1	1	1	0	1	1	1	1	1	1

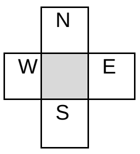

Figure 1: A 10x10 maze. A cell's neighbors consist of cells to the North, South, East and West.

includes three or more wall cells

- Free cells remains a free cell if its neighborhood includes less than three wall cells.

When this set of rules is processed, the algorithm effectively blocks off every dead-end path in the maze. Every free cell that is accessible from only one direction (i.e. three wall cells around it) must be a dead end and therefore cannot be part of the solution. These cells become new wall cells, and this procedures is repeated until the system remains in a steady state. In this state the only remaining free cells represent the solution(s) to the maze. If there is no solution, the entire array of cells will be wall cells.

The following is the specification for the maze-solving model in Cell-DEVS:

CD = < X, Y, I, S, θ, N, d, δ_{int}, δ_{ext}, τ, λ, D >
X = Ø
Y = Ø
S = { 0, 1 }
N = neighborhood = { (-1, 0), (0, -1), (0, 1), (1, 0), (0,0) }
d = 100 ms
τ: N→S is defined by the rules described in the previous section, i.e.:

S = 1 if cell(0,0) = 1
S = 1 if cell(0,0) = 0 and # of "wall"neighbors $\geq$ 3
S = 0 if cell(0,0) = 0 and # of "wall"neighbors <3

The formal specification translates into the following .ma (model definition) in Cell-DEVS:

```
[maze]
type : cell
dim : (20, 20)
delay : transport
border : nowrapped
neighbors :    maze(-1,0)
neighbors : maze(0,-1)  maze(0,0)  maze(0,1)
neighbors :     maze(1,0)
localtransition : maze-rule

[maze-rule]
rule : 1 100 { (0,0) = 0 and (truecount = 3 or
truecount = 4) }
rule : 0 100 { (0,0) = 0 and truecount < 3 }
rule : 1 100 { t }
```

Figure 2: Defining the Cell-DEVS specification in CD++.

This model was executed using the CD++ simulation toolkit, and the results were visualized using the tool's visual output facilities. The results are showed in figure 3, which include the graphical displays of a maze with a given initial state (a) and the results of solving the maze (b).

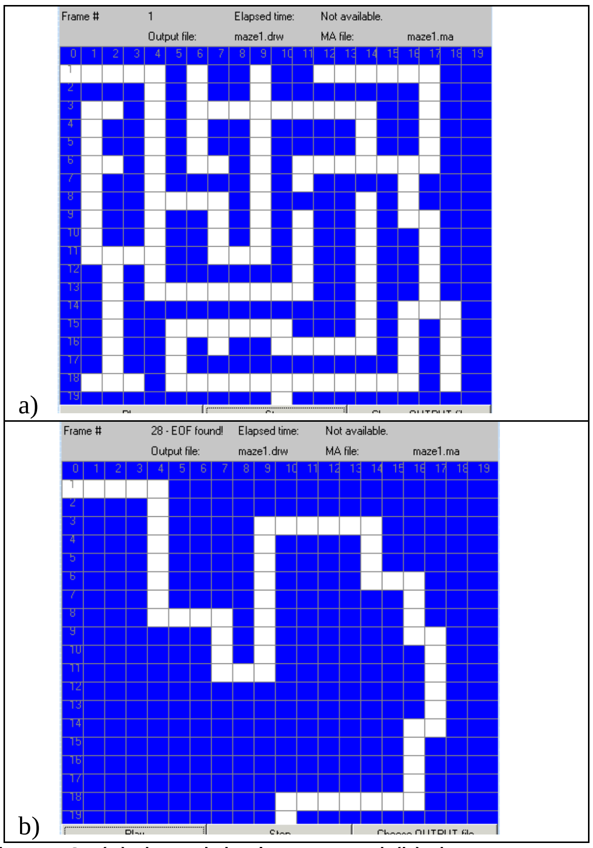
a)
Frame # 1 Elapsed time: Not available.
Output file: maze1.drw MA file: maze1.ma
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
b)
Frame # 28 - EOF found! Elapsed time: Not available.
Output file: maze1.drw MA file: maze1.ma
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Play
Stop
Choose OUTPUT file

Figure 3: (a) the original maze and (b) the maze after processing in Cell-DEVS.

If a maze has no solution, or many solutions, the cell space will generate a result without any cell in the final path, that is, a solid block of wall cells (see Figure 4a). Likewise, if the maze has different solutions, the cell space will stop evolving when all the solution paths are revealed in mazes where more than one path belongs to the solution (see Figure 4b). Further processing would be required for a complete solution to be made available.

4. COLLISION-FREE PATH PLANNING IN CELL-DEVS

This section describes a Cell-DEVS model to simulate a path-planning algorithm described in [3]. The paper proposes the use of cellular automata to process a "top down" bitmap of an area to be traveled by a robot, assuming that a robot is defined as a diamond that can enclose a robot of arbitrary shape. The algorithm produces a Voronoi diagram, which can be used to determine a path equidistant from any obstacles in the space. The paths are calculated by marking the intersections of expanding "wavefronts" propagated by cellular expansion from given starting points.

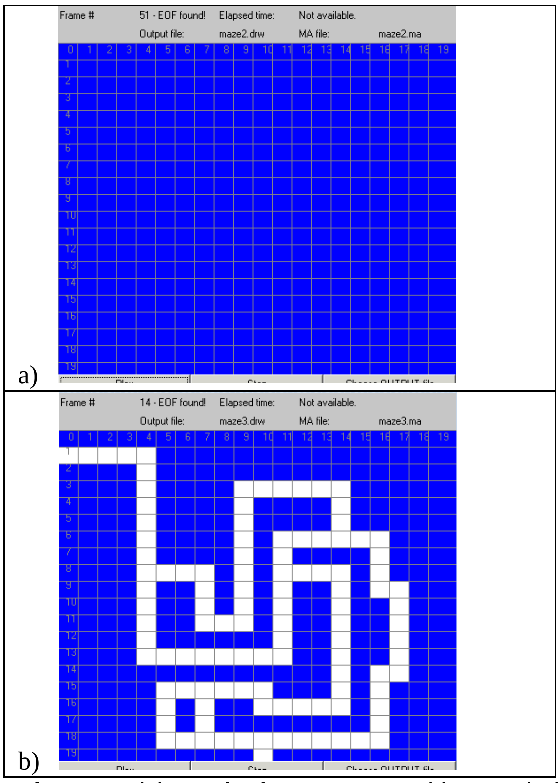

Figure 4: (a) results from a maze with no solution and (b) a maze with multiple solution paths.

As with Nayfeh's algorithm, cells have a value of "1" or "0" for wall and free cells, and neighbors consist of the cardinal directions North, East, South and West (i.e. the von-Neumann neighborhood). The rules defining the model's behavior can be characterized by a two-stage procedure:

- In the first stage, cells and their neighborhoods are examined and compared to a set of 12 "edge code" templates. For each cell that matches a configuration in the template, a corresponding edge code from 1-12 is used in the second stage. This is an "object boundary detection" stage.
- In the second stage, cells containing edge codes (derived from object boundary detection) are expanded in free space. Where expansions intersect, the cell of intersection is given a timestamp and considered part of the final Voronoi diagram.

This is a closed cellular model with no external inputs or outputs. The final state of the cellular array contains the Voronoi diagram describing a collision-free path.

The authors presented their original path-finding algorithm as C-like pseudocode, broken into two distinct stages (Object Boundary Detection followed by Voronoi Diagram Construction). The second stage is further divided into sub-stages. In addition, the algorithm references not only the actual cell value at any given cell (i, j), but also defines additional variables corresponding to each cell. The algorithm requires the following set of data for every cell:

z(i, j) – the original encoding of detected obstacles (0 or 1)
edge_code(i, j) – calculated edge code for the cell (1-12)
Flag(i, j) – value used during "wavefront expansion"
Vor(i, j) – the point on the Voronoi diagram representing this cell's position

Because a Cell-DEVS model only supports one value for any given cell, and the algorithm requires four, a single two-dimensional model does not suffice. Thus, the algorithm is implemented using a 3D Cell-DEVS model, in which each plane represents a set of state variables. The x and y dimensions are dependent on the input values and represent the two-dimensional space being considered. The model consists of four such planes of size $x \cdot y$, i.e. the dimension of the z-axis is 4. Each plane contains the data represented in each of the four variables discussed previously:

plane 0 (x, y, 0)	original bitmap representing the space
plane 1 (x, y, 1)	edge codes
plane 2 (x, y, 2)	propagation of edge codes over time
plane 3 (x, y, 3)	final Voronoi diagram

In the original specification, the cellular automata make use of a Von-Neumann neighborhood that includes each cell as well as its neighbors in the four cardinal directions North, East, South and West. However, because the Cell-DEVS implementation includes a third dimension, the neighborhood must be expanded to allow each plane access to values in the plane immediately below it. The expanded neighborhood includes each cell, its four cardinal neighbors, and the five corresponding cells in the plane below. Figure 5 below illustrates the neighborhood for any given cell (the shaded cell in the figure).

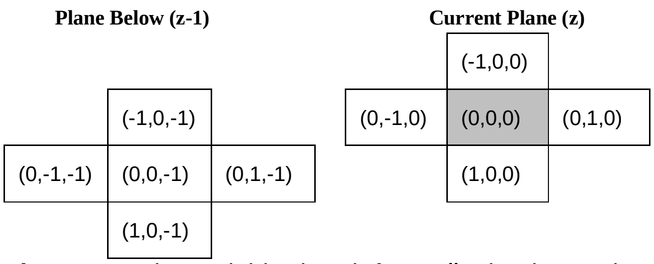

Figure 5: The neighborhood for cells in the path-planning model

The following is the formal specification for the Cell-DEVS path-finding model:

$CD = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$

X = Ø
Y = Ø
S = ∈ R (real numbers)
N = neighborhood = { (-1, 0, 0), (0, -1, 0), (0, 1, 0), (1, 0, 0), (0, 0, 0), (-1, 0, -1), (0, -1, -1), (0, 1, -1), (1, 0, -1), (0, 0, -1) }
d = 10 ms
τ: N→S is defined by the rules below. Each z-plane has its own set of rules.

This Cell-DEVS model has dimension 10x10x4, representing a 10x10 maze and the four data planes used to model it. The initial cell values are loaded from an external file. The model defines four sets of rules (nothing-rule, bound-rule, plane2-rule, and plane3-rule) which are used by each of the cell planes (0, 1, 2, 3 respectively). The three-dimensional cell model is effectively divided into four linked two-dimensional models by using separate zones consisting of plane regions (i.e. each zone Z consists of cells {(0,0,z)..(9,9,z)}.

The rule sets are as follows:

nothing-rule
This rule essentially does nothing. Cell values are not changed. This rule is used by the original data plane to keep the values from being changed.

bound-rule
This rule performs the coding of edge directions as described in [3]. Patterns of cell values in each cell and its neighborhood are classified as one of 12 "edge codes". The rules in this section perform the classification, causing cells in this plane to take on integer values between 1-12 if the cells in the data plane correspond to one of the 12 templates.

plane2-rule
The paper indicates that cells with edge codes from 1-4 are discarded. Cells with edge codes 5-12 are copied into a new CA grid and given a flag value for propagation in the third stage. The rules in this section carry over the values from the second plane which satisfy the criteria (4 < edge_code < 13). In this Cell-DEVS implementation, cells are flagged by adding a fractional value 0.1 to their value. They can then be tested for the presence of this "flag" by checking for a fractional part, and the flag can be removed by using the `trunc()` function [3]. The cell values propagate across to neighboring cells if the flag is set.

The authors of the algorithm suggests that the flag value of a cell is not copied when a neighboring cell receives its new value. This appears to be an error, as this algorithm was observed to work well only when flags **are** copied when cells change value.

plane3-rule
The final plane represents the Voronoi diagram produced by the CA algorithm. In the previous plane, cells receive data values from their immediate neighbors and in doing so they effectively propagate the data out from any given starting point. The theory behind this path finding algorithm is that points where these data wavefronts collide are points farthest away and equidistant from the starting points (obstacles). As such, these are the points of interest when plotting a path for a robot.

The rules in this plane examine the cell values (and their flags) in the neighborhood of the plane below. According to the algorithm, if a cell's neighborhood consists of more than one cell whose flag is set, and those cells with flags do not contain the same values, then the cell belongs on the Voronoi diagram. The cell on the Voronoi diagram is given a time stamp – the CA iteration number at which the cell was added to the diagram.

The Cell-DEVS model was simulated using the CD++ tool. Figure 7 shows a sample maze and the state of each of the four dimensions of the cellular model when the algorithm completes.

To interpret the Voronoi diagram, "for a diamond shape of diagonal size d, the path planning process selects those Voronoi edges that consist of points with labels of value $\ell \geq d + ½$" [3]. In this case since the first values that appear on the diagram are 2's, one should add that offset to find the desired values. In this case, for a robot of diagonal size 2, the points on the graph of value 4 or 5 represent viable travel paths.

5. CONCLUSION

We have presented the application of the Cell-DEVS formalism in maze solving problems. We used the CD++ toolkit to model and simulate the proposed maze applications, showing that this approach permits easy solving of these applications. We were able to describe Cell-DEVS models described that correctly simulate the behaviour of the path-finding algorithms presented in [2, 3].

The use of Cell-DEVS to solve the path-finding (or large scale maze solving) problem is very efficient, as it can operate extremely quickly (in just a few cycles of CA evolution) and every cell is being solved in parallel. This is a stark contrast to more traditional, mathematical approaches to path-finding which can require many mathematical calculations of distances and angles, and backtracking to recover from dead-ends.

The logic behind the algorithms is very simple and can be scaled to larger spaces without difficulty. It would be interesting, in further studies, to determine if this algorithm would scale to 3-dimensional spaces (or even n-dimensional ones) or whether the algorithm would still be valid for different neighbor sets (i.e. a hexagonal or triangular neighborhood).

```
   0 1 2 3 4 5 6 7 8 9       0 1 2 3 4 5 6 7 8 9
 +--------------------+    +--------------------+
0| 1 1 1 1 1 1 1 1 1 1|   0|                    |
1| 1 1 1              |   1|   5 7              |
2|                    |   2|       2 2 2 2 2 2  |
3|                    |   3|   2 2 2 2 2 2 2 2  |
4|                    |   4|   2 2 2       2 2  |
5|         1 1 1      |   5|   2 2  11 912   2  |
6|         1 1 1      |   6|   2 2   8 5 7   2  |
7|                    |   7|   2 2 2       2 2  |
8|                    |   8|                    |
9| 1 1 1 1 1 1 1 1 1 1|   9|                    |
 +--------------------+    +--------------------+

  0 1 2 3 4 5 6 7 8 9       0 1 2 3 4 5 6 7 8 9
 +--------------------+    +--------------------+
0|                    |   0|                    |
1|   5 7 7 7 7        |   1|   3 3     5 5      |
2|   5 7     912      |   2|   3 3   4 5 5      |
3|   5 7  11 91212    |   3|   4 4 4 4 4 4      |
4|   5    11 912  12  |   4|   5 4   3 3 3      |
5|  11111111 9121212  |   5|   5 4 3 2 2 2 3 4  |
6|   8 8 8 8 5 7 7 7  |   6|   5 4 3 2 2 2 3 4  |
7|     8   8 5 7   7  |   7|         3 3 3      |
8|       8 8 5 7 7    |   8|         4 4 4      |
9|                    |   9|                    |
 +--------------------+    +--------------------+
```

Figure 7: The four planes as modeled by Cell-DEVS: the original maze on the upper left, and the Voronoi diagram on the bottom right.

The downside to these algorithms is that they require a full knowledge of the obstacle space prior to solving it (since they cannot operate on cells of unknown value). In a real-world implementation, this could be provided from an overhead camera that generates a bitmapped image representing the space. In addition, neither model provides a complete solution in the case where there is not one distinct solution path. If there exist several paths, the algorithms provide a partial solution. The authors of [3] are evidently aware of this limitation, indicating that the Voronoi diagram produced "is suitable for post-processing in a variety of external tasks."

ACKNOWLEDGMENTS

This work was partially funded by the Natural Sciences and Engineering Research CouncIl of Canada (NSERC) and the Institute of Robotics and Intelligent Systems (IRIS, Canada).

REFERENCES

[1] TALIA, D. "Cellular processing tools for high-performance simulation". IEEE Computer. September 2000. Pp. 44 –52.

[2] NAYFEH, B. "Cellular Automata For Solving Mazes". *Doctor Dobb's Journal*, February 1993.

[3] TZIONAS, P., THANAILAKIS, A., TSALIDES, P. "Collision-Free Path Planning For a Diamond-Shaped Robot Using Two-Dimensional Cellular Automata". In *IEEE Transactions on Robotics and Automation*, Vol 13, No 2, April 1997.

[4] WAINER, G.; GIAMBIASI, N. "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation". G. Wainer, N. Giambiasi. *Simulation*, Vol. 71, No. 1. January 2001. pp. 22-39.

[5] WAINER, G. "CD++: a toolkit to define discrete-event models". 2002. In *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306.