Modeling Space-Shaped Defense Applications with Cell-DEVS

Rami Madhoun *Gabriel Wainer*

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON. K1S-5B6 Canada
{rmadhoun, gwainer}@sce.carleton.ca

ABSTRACT: *the DEVS formalism has been used as modeling and simulation technique for different natural and artificial systems. Cell-DEVS is an extension of DEVS that allows for executing cellular automata models with the advantage of evaluating the cells asynchronously with different timing delays. Both techniques have shown success in simulating space-shaped models. In this paper, we describe how they can be used to model and simulate different defense-related systems. We emphasize on using Cell-DEVS to model a land battlefield between two armies. Each army is composed of different soldiers and one flag and the goal of each army is to acquire the other's flag. Different aspects of the soldier behavior are considered such as soldier state (alive, injured, or dead), soldier movements towards the enemy's flag, and soldiers fight with each other. In addition, we introduce two different simulation techniques using CD++ (a toolkit developed to execute DEVS and Cell-DEVS models) showing the noticeable performance gain when using some new features introduced in the CD++ toolkit.*

1. Introduction

The need for modeling and simulation techniques has become increasingly important to study the very complex artificial systems of these days, in which actual experimentation on the actual system is not feasible or is too dangerous. One of such techniques that gained a lot of attention in recent years is called DEVS [1] (Discrete Event Systems Specifications), a systems-theoretic formalism. DEVS relies on dividing the system under study into atomic models; each of which can exist in specific state at any point of time and has input/output ports to interact with other models and with the external world. This allows for building very complex models by connecting different atomic models in a hierarchical manner. Different extensions were introduced to extend DEVS capabilities. One of them, called Cell-DEVS, allows executing cellular models with different time delays associated with each cell [2]. We will show how to use DEVS and Cell-DEVS to model and simulate different defense-based applications (such a simple model of an UAV, and land battlefield models, which includes two armies fighting against each other and trying to capture/acquire the other's flag. Two implementation techniques using the CD++ toolkit [3] are presented. We discuss the advantages of newly introduced features in CD++ [4], which accomplished an improvement in terms of the model execution time and resource utilization.

2. Background

DEVS [1] establishes a framework for modeling and simulation of discrete event systems, which depends on representing the systems by a hierarchy of atomic components. DEVS provides and abstract approach of modeling by separating the modeling from the simulation aspects and hence facilitating the model usability and interoperability.

The basic building block of any DEVS model is the *atomic* model, which can be connected to other atomic models to form what is called a *coupled* model. A DEVS atomic model can be informally described as in Figure 1.

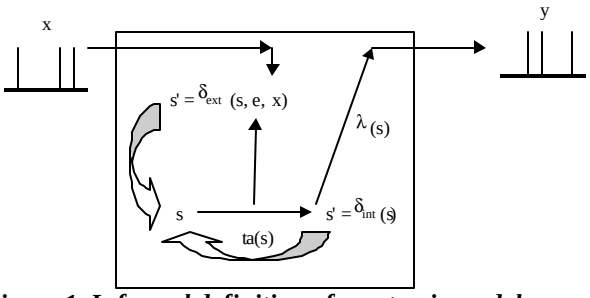


Figure 1. Informal definition of an atomic model

Each atomic model has an interface consisting of *input* (x) and *output* (y) *ports* to communicate with other models. In addition, the *state* (**s**) of the model is associated with a *time advance* (**ta**) function, which determines the duration of the state. Once the time assigned to the state is consumed, an internal transition is triggered. At that moment, the model execution results are spread through the model's output ports by activating an *output function* (l). Then, an *internal transition function* (d_{int}) is fired, producing a local state change. External input events (events received from other models) are collected through the input ports. An external transition function (d_{ext}) specifies how to react to those inputs.

A DEVS coupled model is composed of several atomic or coupled sub-models, as shown in Figure 2.

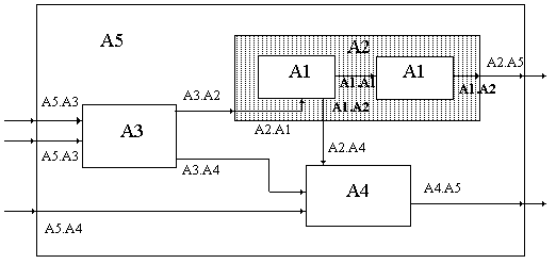


Figure 2. Informal description of a coupled model

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the model interfaces. The model's coupling scheme defines the interconnectivity between models and the interface with the external world.

Cell-DEVS [2] has extended DEVS, allowing the implementation of cellular models with timing delays. Each cell is defined as a DEVS atomic model, and it can be later integrated to a coupled model representing the cell space. Cell-DEVS atomic models can be described as in Figure 3.

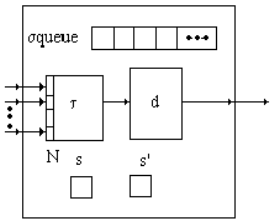


Figure 3. Cell-DEVS atomic model

Each cell uses N inputs (from its neighborhood) to compute its next state. These inputs, which are received through the model's interface, activate a local computing function (t). A delay (**d**) can be associated with each cell. The state (**s**) changes can be transmitted to other models, but only after the consumption of this delay. Two kinds of delays can be defined: *transport* delays model a variable commuting time, and *inertial* delays, which have preemptive semantics (scheduled events can be discarded if the computed value is different than the future state).

Once the cell behavior is defined, a coupled Cell-DEVS can be created by putting together a number of cells interconnected with its neighbors. A sample Cell-DEVS coupled model is presented in Figure 4. A coupled Cell-DEVS is composed of an array of atomic cells, with given size and dimensions. Each cell is connected to its neighborhood through standard DEVS input/output ports. Border cells have diffe rent behavior due to their particular locations, which may result in a non-uniform neighborhood.

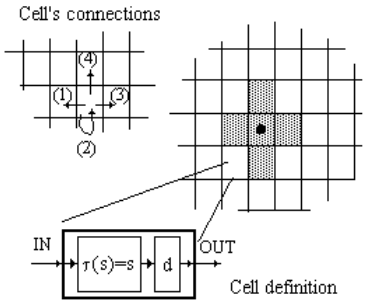


Figure 4. Cell-DEVS coupled model

CD++ [3] is a modeling and simulation environment developed in C++ following the formal specifications of DEVS and Cell-DEVS. It is used to build and execute DEVS and Cell-DEVS models. DEVS Atomic models are programmed in C++ and incorporated into CD++ class hierarchy. Once an atomic model is defined, it can be combined with others into a multi-component model using a specification language specially defined for this purpose. In addition, different versions have been developed for different platforms: a stand-alone version, a real-time simulator [5], and a parallel simulator [6].

In the case of Cell-DEVS models, the model specification includes the size, dimension of the cell space, the shape of the neighborhood and the borders. The cell's local computing function is defined using a set of rules with the following format:

POSTCONDITION DELAY { PRECONDITION }

This indicates that when the *PRECONDITION* is satisfied, the state of the cell will change to the designated *POSTCONDITION*, which computed value will be transmitted to the other cells after some *DELAY* has elapsed. If the precondition is *false*, the next rule in the list is evaluated until a rule is satisfied or there are no more rules. If no rules are evaluated for a certain cell or more than one has a condition evaluated to true, CD++ will generate an error in order for the modeler to crosscheck the rule definition.

```
[life]
width : 20              height : 20
delay : transport       border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1)
neighbors : (0,-1)  (0,0)  (0,1)
neighbors : (1,-1)  (1,0)  (1,1)
localtransition : new-life-rule

[new-life-rule]
Rule: 1 10 { (0,0) = 1 and ( truecount = 3
                    or truecount = 4 ) }
Rule: 1 10 { (0,0) = 0 and truecount = 3 }
Rule: 0 10 { t }
```

Figure 5. Definition of the Life game

Figure 5 shows the definition of a simple example. The rules in this example implies that a cell remains active when the number of active neighbors is 3 or 4 (*truecount*) using a transport delay of 10 ms. If the cell is inactive (`(0,0) = 0`) and the neighborhood has 3 active cells, the cell is activated (value = 1). In every other case, the cell remains inactive (t indicates a condition that is always true).

3. Defining Defense Applications

DEVS and Cell-DEVS have been used to model and simulate different military-related systems. In [7], we presented a model of the synchronization effect between radar transmitters and receivers, and a Cell-DEVS model of a simple vehicle seeking a target. We discussed how to integrate these models to achieve interoperation between two models defined with different techniques. Here, we will present new models that can be easily integrated into these applications, and we present a brief discussion of the models highlighting the main components in each model.

The first example we will discuss considers a model of an is Unmanned Ariel Vehicle (UAV) system that was built using Cell-DEVS. The UAV traverses an specific area searching for a target, and avoiding static and moving obstacles in its way. The model deals with multiple UAVs moving and avoiding multiple obstacles. In order to model the behavior of UAVs and obstacles, each entity is assigned a state value as follows:

	Empty Cell	UAV	Moving Obstacle	Static Obstacle
Color				
Movement	None			None
State	0	1	5	9

Figure 6. UAV state values

The model is specified using CD++ specification language, which defines the cell space shape, size and the rules that govern the model execution. The first portion of the coupled model defines the cell-space geometry and initial values as shown in Figure 7. The neighborhood shape covers the direction in which the UAV is moving (North-South).

```
[top]
components : uav

[uav]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 100
border : nowrapped
neighbors : uav(-2,-2) uav(-1,-2) uav(0,-2) uav(1,-2)
uav(2,-2)
neighbors : uav(-2,-1) uav(-1,-1) uav(0,-1) uav(1,-1)
uav(2,-1)
neighbors : uav(-2,0) uav(-1,0) uav(0,0) uav(1,0) uav(2,0)
neighbors : uav(-2,1) uav(-1,1) uav(0,1) uav(1,1) uav(2,1)
neighbors : uav(-2,2) uav(-1,2) uav(0,2) uav(1,2) uav(2,2)
neighbors : uav(3,-2) uav(3,-1) uav(3,0) uav(3,1) uav(3,2)
initialvalue : 0
initialrowvalue : 5    00000000099000000000
initialrowvalue : 0     10010100001000010000
initialrowvalue : 15    00000000900000000000
```

Figure 7. UAV coupled model specification

As shown in Figure 7, the cell space is composed of 20x20 cells with a Transport delay of 100 time units and initial values as defined by the *InitialRowValue* statement.

```
[noFlyZone9-rule]
rule : 9 100 { (0,0) = 9 }
%rule : 9 100 { (0,0) = 9 }
.
.
.
[uav-rule]
%000
%???
rule : 1 100 { (0,0)=0 and (0,-1)=0 and (0,1)=0 and (1,-
1)!=5 and (1,0)!=5 and (1,1)!=5 and (-1,0)=1}
rule : 0 100 { (1,0)=1 and (0,0)=1 }
.
.

%moving target rule
rule : 5 100 { (1,0) = 5 }
rule : 0 100 { (-1,0) = 5 }
```

Figure 8. UAV: rule definitions

Figure 8 shows part of the rule definition of the static obstacles, UAVs, and moving obstacles. The *noFlySone9-rule* implements the static obstacle rule (state value=9), which is constant all the time due to the static nature of the obstacles. The *uav-rule* implements the UAV movement avoiding the static and moving obstacles. Finally, the 'move target rule' implements a moving obstacle from south to north.

Since the model was built and tested using the CD++ toolkit following the formal Cell-DEVS definition, it can be incorporated with other DEVS and Cell-DEVS models such as the radar model and the battlefield model (discussed in the next section).

Figure 9 shows a snapshot of CD++ Modeler, (part of the CD++ toolkit) with an initial allocations of UAV and obstacles. The UAVs (shown in red/dark gray) try to move from north to south facing static obstacles (shown

in black) as well as moving obstacles (shown in yellow/light gray).

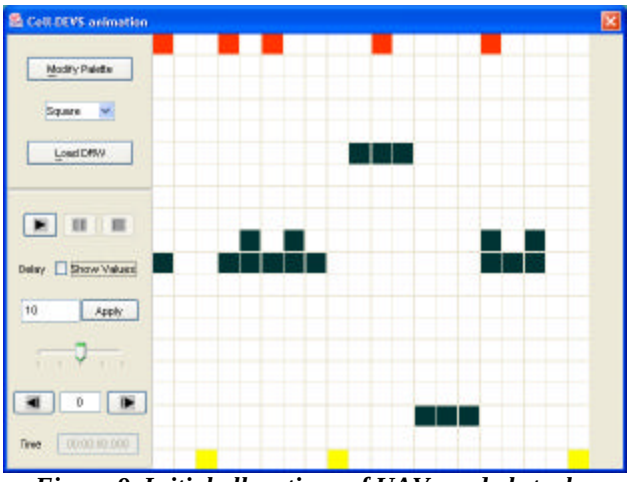


Figure 9. Initial allocations of UAVs and obstacles

Our second example shows the use of Cell-DEVS to model and simulate a land battlefield. Different approaches followed in previous research of this kind of systems include Cellular Automata as in [8] and software agents [0]. In our case, we converted these models into Cell-DEVS, and implemented them using the CD++ toolkit. In this scenario, two armies engage in a fight, each one is composed of different soldiers and a flag. The goal of each army is to capture the enemy's flag or to defend its own.

The characteristics of the systems can be summarized as follows:

- A Two dimensional battlefield is considered without any airplanes or missiles.
- Each soldier can exist in one of three states: *alive*, *injured*, *dead*.
- The situation awareness of the soldier is limited to his neighborhood (no telecommunication equipment are used).
- If a soldier is in state *Alive*, and attacked by an enemy soldier, his state changes to *injured*.
- If a soldier is in state *injured* and is attacked by an enemy soldier, he becomes *dead*.
- The soldier's ability to fight is dependent on a randomly assigned factor (Fighting Ability FA). In addition, the *injured* soldier will have a less fighting ability than the *alive* one.
- *Injured* soldiers recover to *alive* state if not surrounded by enemy soldiers.
- If a soldier is not surrounded by enemy soldiers, he tends to move towards the enemy's flag.
- If a soldier is surrounded by an enemy soldier/s, he engages in a fight. The outcome of this fight depends on the fighting ability (FA) of the soldiers engaged in the fight.
- The flag is acquired once an enemy soldier moves to its neighborhood.

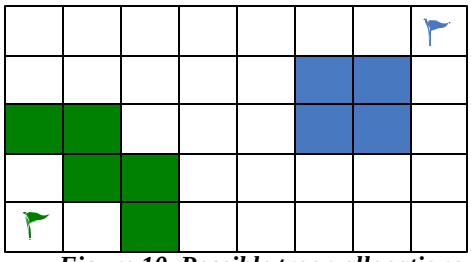


Figure 10. Possible troop allocations

The status of the soldier is represented by a signed integer to distinguish between the two armies. One of the armies has positive values (army A) and the other has negative values (army B).

The following table describes this representation:

Status	Description
2	Fighter of army A alive
1	Fighter of army A injured
0	Fighter is dead and cell is empty
-1	Fighter of army B injured
-2	Fighter of army B alive
5	Flag of army A
-5	Flag of army B

The fighting ability of each soldier is represented by a randomly assigned real number ranging from 0 to 1. Zero represents no fighting ability at all (in the case of flag and dead soldiers), while 1 represents a very high fighting ability. In addition, the soldier will have an effect on the enemy soldier only if his fighting ability is greater than 0.5. The assignment is done using random function with a uniform distribution and is executed at two points:

- At the beginning of the battle
- After engaging in a fight with an enemy soldier

The following table describes the fighting ability factor:

Table 1. Fighting Ability states

Status	Fighting Ability
2	Uniformly distributed number in the range [0.45, 1]
1	Uniformly distributed number in the range [0,0.55]
0	Fighter is dead and cell is empty 0.0
-1	Uniformly distributed number in the range [0,0.55]
-2	Uniformly distributed number between in the range [0.45,1]
5	Does not engage in fights 0.0
-5	Does not engage in fights 0.0

When two or more soldiers engage in a fight, the outcome depends on the difference between their fighting abilities (FAs), as seen in Figure 11.

Since each soldier aims to acquire the enemy's flag, he needs to know about the flag position. This information is represented as a real number having the integer part representing the flag row number (y-coordinate) and the fractional part representing the flag column number (x-coordinate), i.e. **Row + Column/100 (ex. row=2, column=4 → 2.04)**.

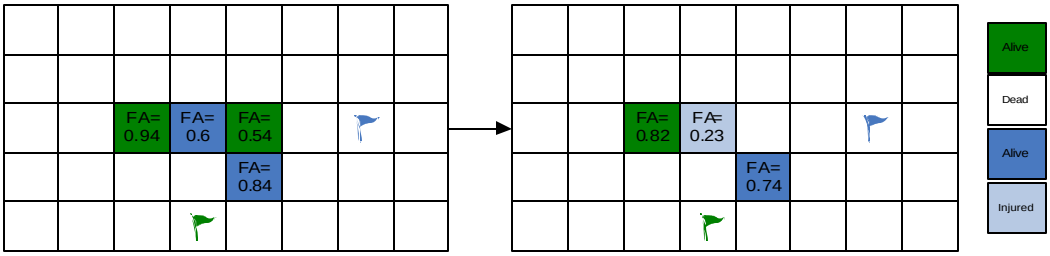


Figure 11. The effect of different FAs in a fight

If a soldier is not surrounded by the enemy, he tends to move towards enemy's flag. To do so, the soldier needs to calculate his direction in the next step to come closer to his target. This is done by comparing the current cell position of the soldier with the enemy's flag position. For example, if the soldier is standing at cell (1, 1) and the enemy's flag position is at cell (3, 4); he will have two options, either to move to the east or to the south as shown in Figure 12.

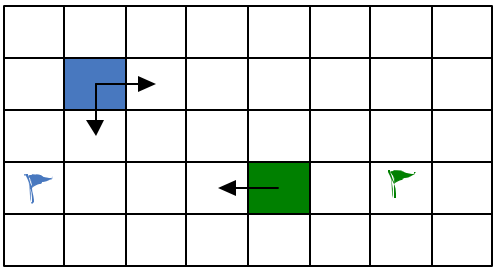


Figure 12. Movement directions

After deciding on the direction of the next step, the directions are assigned integer values according to the following table:

Table 2. Direction values

Direction	Value
North	10
East	20
South	30
West	40

The Free Cell move-in factor is an integer number that is calculated for every free cell to resolve any conflict if two or more soldiers want to move to the same free cell.

In one of our implementations, this factor is evaluated as the maximum fighting ability of the soldiers surrounding the free cell. The following figure illustrates this point.

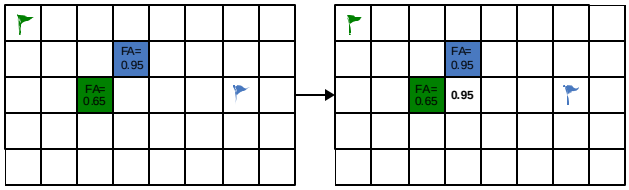


Figure 13. Free-cell move-in factor evaluation

A different implementation computes the free-cell move-in factor as the maximum fighting ability of the soldiers in the neighborhood who **intend** to move to the cell. Only the one with the maximum FA will be allowed to move to the free cell. In this scenario, the free-cell move-in factor will be the direction of that soldier (the one with maximum FA) with an opposite sign to indicate that the cell will be occupied by the soldier coming from that direction. The following figure illustrates this point.

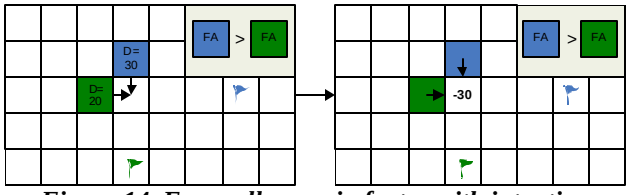


Figure 14. Free-cell move-in factor with intention

The model was implemented using CD++ (a detailed definition of the specification can be found in [10]. Each piece of information was implemented using a different layer, which resulted in a 3-dimensional cell space. The layers used to implement the model are as follows:

- Layer 0: soldier's status and allocation in the battlefield.
- Layer 1: fighting ability factor (FA), used for movement and fighting rules evaluation

- Layer 2: flag position of army B. This information is needed for all the soldiers of army to A calculate the next movement direction.
- Layer 3: flag position of army A. This information is needed for all the soldiers of army B to calculate the next movement direction.
- Layer 4: movement directions of each soldier.
- Layer 5: move-in factor associated with each free cell.

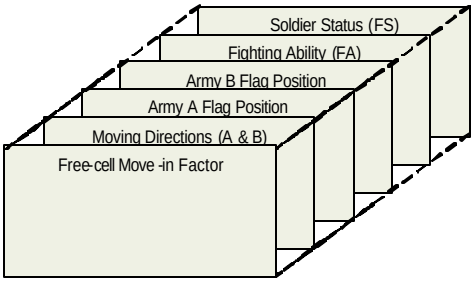


Figure 15. Cell space definition.

The model was executed with different test scenarios. The first one we present here is devoted to analyze only the movement rules of the fighters towards the enemy's flag. Figure 16 shows the initial and final configuration of the army (one fighter of each army was killed in the battle; both armies eventually reach the flags).

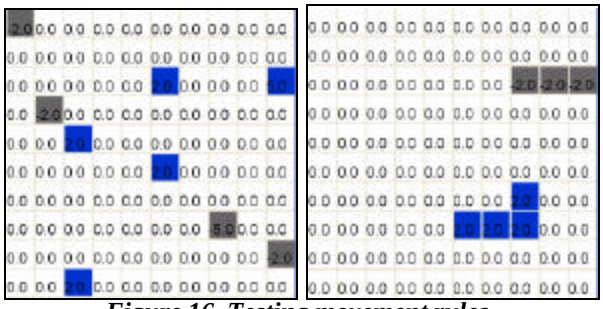


Figure 16. Testing movement rules.

Different tests were carried out, including several overall execution of the models. The following figure shows a 3D visual result of the execution of the model, in which each of the layers previously discussed is depicted.

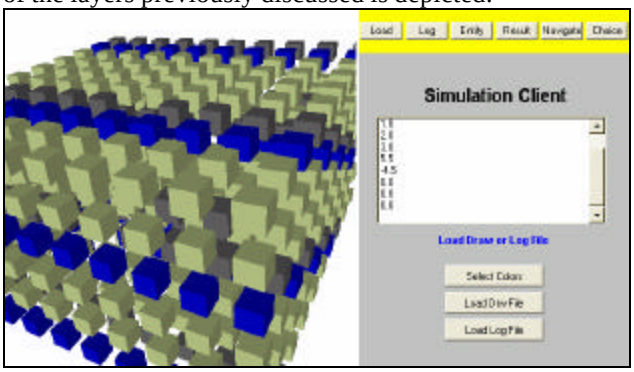


Figure 17. Multilayer display: execution results.

4. Advanced Battlefield model definition

The Battlefield model was extended using new advanced facilities available in a recently developed version of CD++ [4]. This new CD++ extensions include the ability to define multiple input/output ports for each cell in the cell space, and the ability to define multiple state variables per cell, as shown in Figure 18.

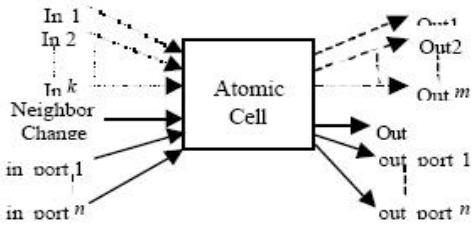


Figure 18. Multi-port cell

The input/output ports connect each cell to all of its neighboring cells, so it is useful to represent information that need to be transferable between different cells. However, the state variables are local to the cell, and are used to represent any variable that does not need to be referenced from outside the cell. Both features are used to re-implement the original battlefield model dispensing with the need to define extra layer of cells to represent new piece of information.

The original battlefield model was implemented using these new services, as a two-dimensional cell space with the following input/output ports:

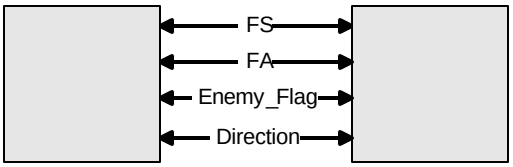


Figure 19. Multi-port connectivity between two cells

- **FS**: is used to represent the soldier status (i.e. *alive*, *injured*, *dead*)
- **FA**: is used to represent the fighting ability of the soldier
- **Enemy_Flag**: is the location of the enemy flag represented in the same format explained earlier.
- **Direction**: is used to represent the direction of the next move of the soldier.

In order to implement the model using the new version of CD++, different rules were defined to mimic the behavior of soldiers in a battlefield. These rules include:

- Initialization rules: initialize the cell ports to their initial values.
- Fighting rules, define the behavior of soldiers when engaged in a fight.
- Flags-under-attack rules, defines the behavior of the flag when attacked by an enemy soldier.
- Flags-not-attacked rules, defines the behavior of the flag when not attacked.
- Movement-direction rules, defines the direction of the next step for each soldier to come closer to the enemy flag.
- Movement rules, define the behavior of the soldiers when moving in the battlefield.

As an example of these rules, we present the implementation of the fighting rules in CD++.

```
#BeginMacro(fight_rule_1)
(
if( ((-1,0)~fs = -1 or (-1,0)~fs = -2) and (-1,0)~fa > 0.5 and
  ((-1,0)~fa > (0,0)~fa) , -1, 0) +
if( ((0,-1)~fs = -1 or (0,-1)~fs = -2) and (0,-1)~fa > 0.5 and
  ((0,-1)~fa > (0,0)~fa), -1, 0)  +
if( ((0,1)~fs = -1 or (0,1)~fs = -2) and (0,1)~fa > 0.5 and
  ((0,1)~fa > (0,0)~fa), -1, 0) +
if( ((1,0)~fs = -1 or (1,0)~fs = -2) and (1,0)~fa > 0.5 and
  ((1,0)~fa > (0,0)~fa) , -1, 0)
)
#EndMacro
```

Figure 20. Fighting Rules Macros

The macro *fight_rule_1* in Figure 20 checks if the soldier (from army A) is in the neighborhood of an enemy soldier (from army B). Then, checks if the soldier has a higher fighting ability, and in that case adds (-1) to the overall value of the macro for each such soldier.

```
rule : { ~fs:= 1 ; ~fa:= uniform(0,0.55) ; ~direction := 0 ;  } 100
{  (0,0)~fs = 1 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and (#macro(fight_rule_1)) = 0  }

rule : { ~fs:= 0 ;~fa:= 0 ; ~direction := 0 ; ~enemy_flag := -1 ; } 100
{  (0,0)~fs = 1 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and  (#macro(fight_rule_1)) < 0  }

rule : { ~fs:= 2 ; ~fa:= uniform(0.45,0.99) ; ~direction := 0; } 100
{  (0,0)~fs = 2 and  ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and (#macro(fight_rule_1)) = 0  }

rule : { ~fs:= 1 ; ~fa:= uniform(0,0.55) ; ~direction := 0 ;  }  100
{ (0,0)~fs = 2 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and  (#macro(fight_rule_1)) = -1 }

rule : { ~fs:= 0 ; ~fa:= 0 ; ~direction := 0 ; ~enemy_flag := -1; } 100
{  (0,0)~fs = 2 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and  (#macro(fight_rule_1)) < -1 }
```

Figure 21. Fighting Rules

The number generated by *fight_rule_1* is used in the main body of the rule (presented in Figure 21) to evaluate the following conditions:

- If a soldier in Army A is injured (FS=1) and is surrounded by enemy soldiers whose fighting ability are less than his, he will remain injured but will be assigned a new fighting ability factor.
- If a soldier in Army A is injured (FS=1) and is surrounded by enemy soldiers whose fighting ability are higher than his, he will be dead and his fighting ability is assigned the value 0.
- If a soldier in Army A is alive (FS=2) and is surrounded by enemy soldiers whose fighting ability are less than his, he will remain alive and assigned new fighting ability factor.
- If a soldier in Army A is alive (FS=2) and is surrounded by enemy soldiers and only one of them has a higher fighting ability, he will be injured and assigned new fighting ability factor.
- If a soldier in Army A is alive (FS=2) and is surrounded by enemy soldiers and more than one of them has a higher fighting ability, he will be dead and his fighting ability factor becomes zero.

The same rule is used for B soldiers when surrounded by an A army soldiers by changing the corresponding soldier status values.

The following figure shows different scenarios for testing, each activating some specific rule/s and then testing the overall model with a scenario that activates all of the rules simultaneously. Three scenarios were used to test the model behavior:

- *Movement rules*: in this scenario, only the movement rules are activated as the soldiers of army A move towards and acquire the B flag.

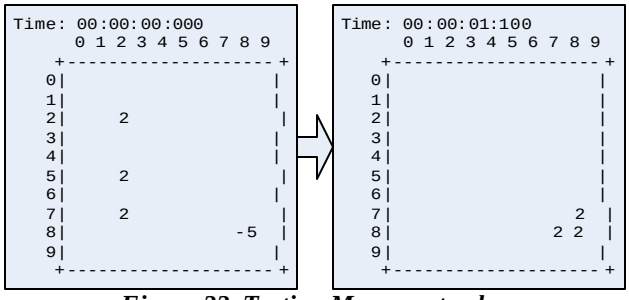


Figure 22. Testing Movement rules

- *Fighting rules*: in this scenario the fighting rules are activated when the soldiers of both armies engage in a fight.

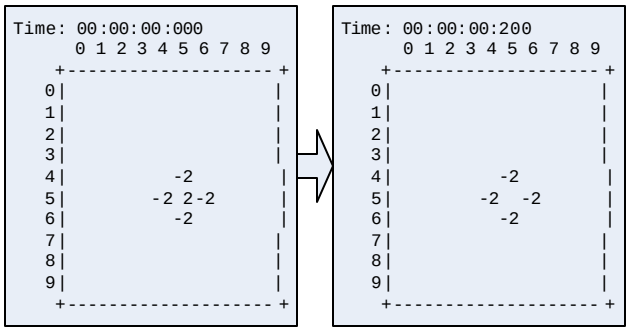


Figure 23. Testing Fighting rules

- *Global test*: all of the rules are activated to test the overall behavior of the model.

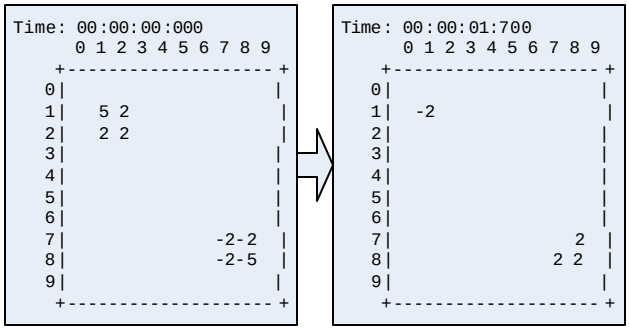


Figure 24. Overall test of the model

5. Performance Analysis

After implementing the same model using the old and the new versions of CD++, some performance metrics (execution time, CPU load…etc) were collected to compare between the two versions. The scenario used in these test is identical to the last scenario in the previous section. The tests were performed using PIV machine with 512 MB of RAM and running Redhat Linux.

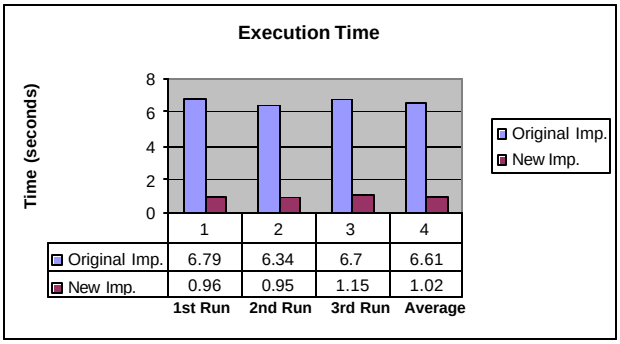


Figure 25. Comparison of the execution time between two different implementations

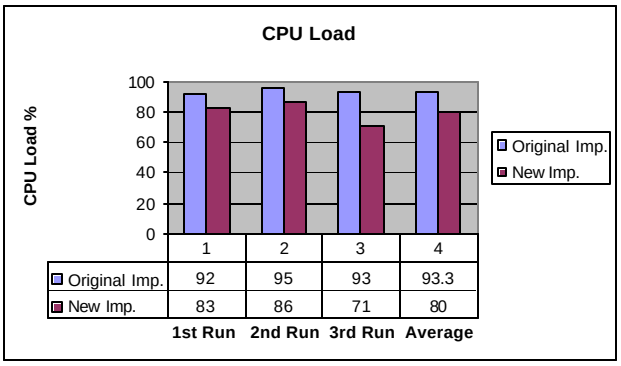


Figure 26. Comparing CPU load

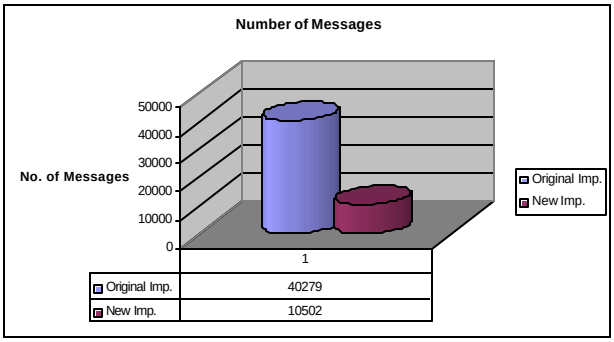


Figure 27. Comparing the number of messages

The memory used by the simulator was the same for both implementations (~3.6 MB). However, by comparing the execution time, CPU load, and the number of messages generated for each implementation, we will find very noticeable performance enhancement when using the new extensions offered by CD++. This enhancement is because the cell space was simplified (2-D instead of 3-D) when implementing the model using the new CD++ features.

After implementing the original model using the new CD++ version, some extra features were added to the model to improve its behavior. These features are:

- Extending the situation awareness of the soldier (neighborhood) to include the eight surrounding cells. Hence, the soldier is able to attack and move diagonally as well as horizontally or vertically.

(-1,-1)	(-1,0)	(-1,1)
(0,-1)	(0,0)	(0,1)
(1,-1)	(1,0)	(1,1)

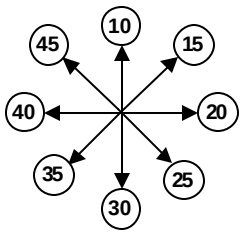


Figure 28. Extending the soldier's neighborhood to Moore's Neighborhood

- Obstacle avoidance, the soldiers are able to avoid obstacles (FS=50) while moving towards the enemy's flag.

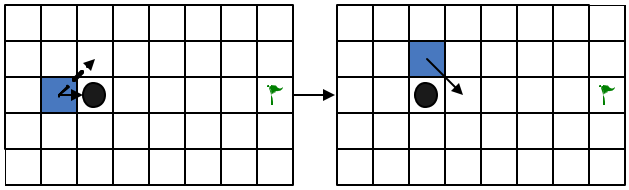


Figure 29. Obstacle avoidance example

- Courage Factor (CF), this factor is used to simulate that not all the soldiers in a battlefield will have the same courage to fight the enemy. Hence, this factor will determine if the soldier is going to attack the enemy or retreat towards his own base/flag.

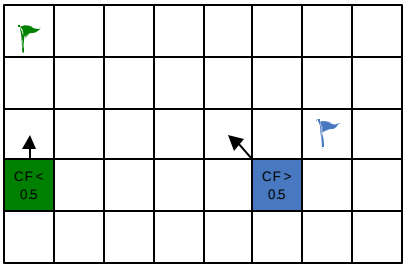


Figure 30. Effect of the Courage Factor FA on the soldier's behavior

In order to test the new features incorporated in the model, two scenarios are considered here:

- The first one tests the diagonal movement and obstacle avoidance of the soldiers.
- The second one, test the overall behavior of the model after incorporating the courage factor CF.

The results of these tests are shown in the following figures:

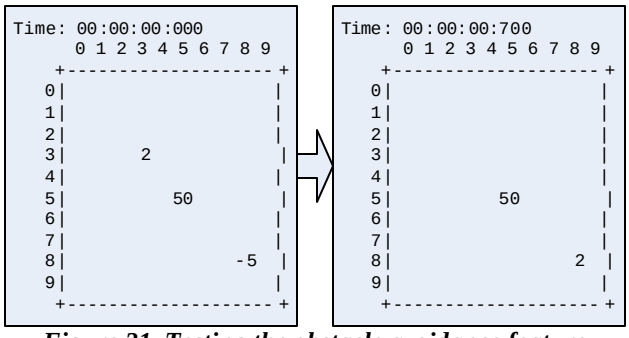


Figure 31. Testing the obstacle avoidance feature

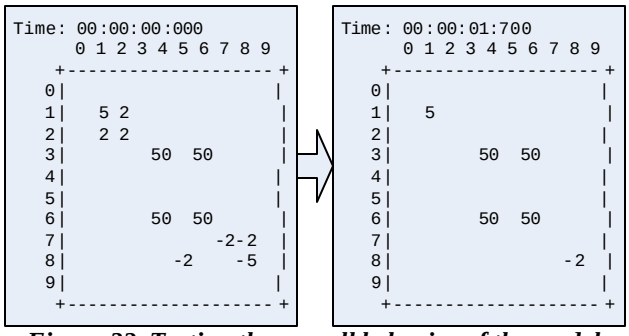
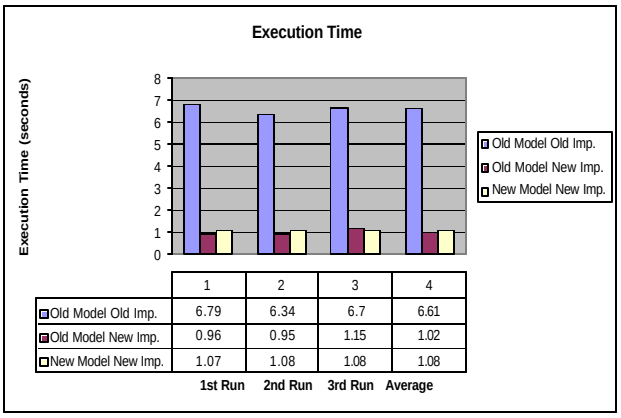


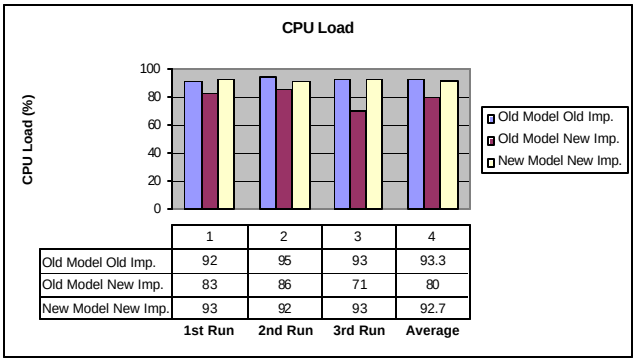
Figure 32. Testing the overall behavior of the model

The following figures show an overall performance analysis of the old model (old and new implementations) and the improved one (new implementation) is presented.



	1	2	3	4
Old Model Old Imp.	6.79	6.34	6.7	6.61
Old Model New Imp.	0.96	0.95	1.15	1.02
New Model New Imp.	1.07	1.08	1.08	1.08

Figure 33. Comparing execution time between



	1	2	3	4
Old Model Old Imp.	92	95	93	93.3
Old Model New Imp.	83	86	71	80
New Model New Imp.	93	92	93	92.7

Figure 34. Comparing CPU load

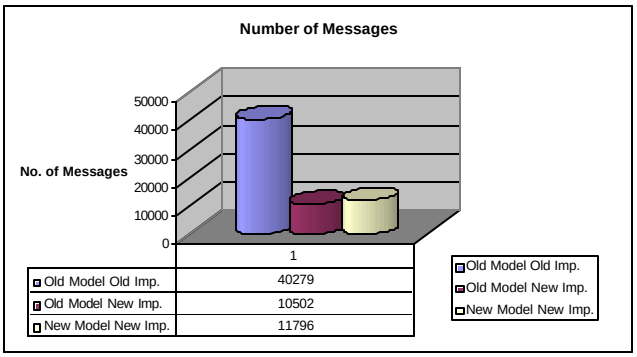


Figure 35. Comparing the number of messages between three different implementations

The previous figures show that the new features in the model have added some overhead in terms of execution time, CPU load, and the number of messages exchanged. However, this overhead is negligible when compared with the performance gains achieved when re-implementing the model using the new version of CD++.

6. Conclusion

We have presented how DEVS and Cell-DEVS can be very useful techniques for modeling and simulating space-shape models. As both of them are based on sound mathematical foundations, that offer a better interoperability capabilities between different models. One can use an existing model of any system and start building on top of it or connect different modules to it provided that he follows DEVS and Cell-DEVS formalisms. In addition, the separation between the model and simulator followed by DEVS and CD++, enables the modeler to concentrate on building the model without studying the internals of the simulator which results in a fast learning curve in terms of using the CD++ toolkit.

The examples presented in this paper show the different aspects to consider when building DEVS and Cell-DEVS models. One of these aspects is to use simple cell-space (if possible) as it executes faster that the complex one. However, some of the systems are highly complex by nature (such as the battlefield model) and this is where the new CD++ features become handy. With multi-port cells, the modeler has the ability to model complex systems by incorporating different kinds of information about the system in these ports. These new features come with a price: the overhead introduced when using this extra functionality. Thus, the performance gain achieved with the battlefield model, may not be achievable in the case of simple models. Previous work [4] has shown that re-implementing different models using the new CD++ facilities version has introduced overhead when executing the model [4]. Hence, the model nature and specification play an important rule in determining whether the old or new versions of CD++ should be considered. In addition, one need to be careful when using multiple input/output ports as they increase the number of messages exchanged within the model, which in turn affect the performance when executing the model in parallel.

References

[1] B. Zeigler; T. Kim; H. Praehofer: Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Academic Press, 2000.

[2] G. Wainer; N. Giambiasi: "Application of the Cell-DEVS Paradigm for Cell Spaces Modeling and Simulation", *Simulation*, Vol. 71, No. 1, pp. 22-39, January 2001.

[3] G. Wainer: "CD++: a Toolkit to Define Discrete-Event Models", *Software, Practice and Experience, Wiley*, Vol. 32, No 3. pp. 1261-1306. November 2002.

[4] A. López, G Wainer. Improved Cell-DEVS model definition in CD++. P.M.A. Sloot, B. Chopard, and A.G. Hoekstra (Eds.): ACRI 2004, LNCS 3305. Springer-Verlag. 2004.

[5] E. Glinsky; G. Wainer: "Performance Analysis of Real-Time DEVS models", In *Proceedings of 2002 Winter Simulation Conference*, San Diego, U.S.A.

[6] A. Troccoli; G. Wainer: "Implementing Parallel Cell-DEVS", In *Proceedings of Annual Simulation Symposium*. Orlando, FL. U.S.A. 2003.

[7] P. MacSween, G. Wainer: "On the Construction of Complex Models Using Reusable Components", In *2004 Spring Simulation Interoperability Workshop*, Arlington, VA, USA, 2004.

[8] A. E. R. Woodcock, L. Cobb, J.T. Dockery: "Cellular Automata: A New Method for Battlefield Simulation", *Signal*, pp. 41-50, January 1988.

[9] A. Ilachinski. "Irreducible Semi-Autonomous Adaptive Combat (ISAAC)-An Artificial Life Approach to Land Combat", *Military Operation Research*, Vol. 5, No 3, pp. 29-46, 2000.

[10] R. Madhoun. "Modeling a battlefield using Cell-DEVS". On-line report. Dept. of Systems and Computer Engineering, Carleton University. http://www.sce.carleton.ca/faculty/wainer/wbgraf [Accessed: January 22, 2005]

Acknowledgements

This work has been partially supported by NSERC (National Science and Engineering Research Council of Canada), the Canadian Foundation for Innovation, the Ontario Graduate Scholarship program, and the IBM Eclipse Innovation Grants program.

Biographies

RAMI MADHOUN has received Bachelor in Electrical and Computer Engineering from the University of Qatar, Qatar, 2000. Then he joined Qatar Telecom to work in activities that include Software Development and Network/System support. He also worked at Convergys as technical support agent before joining the Dept. of System and Compute Engineering at Carleton University, Canada as graduate student. He is first year master student working in the area of Discrete Event Simulation. His e-mail address is rmadhoun@sce.carleton.ca.

GABRIEL WAINER received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. He is Assistant Professor in the Dept. of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada). He was Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires, and a visiting research scholar at the University of Arizona and LSIS, CNRS, France. He is author of a book on real-time systems and another on Discrete-Event simulation and more than 100 research articles. He is Associate Editor of the Transactions of the SCS, and the International Journal of Simulation and Process Modeling (Inderscience). He is Associate Director of the Ottawa Center of The McLeod Institute of Simulation Sciences. He has been awarded Carleton University's Research Achievement Award (2005-2006). His e-mail and website address are gwainer@sce.carleton.ca, http://www.sce.carleton.ca/faculty/wainer.