

Creating Spatially-Shaped Defense Models Using DEVS and Cell-DEVS

Gabriel Wainer

Rami Madhoun

Department of Systems
and Computer Engineering
Carleton University
Ottawa, ON K1S-5B6 Canada
gwainer, rmadhoun @sce.carleton.ca

In recent years, new techniques for military modeling and simulation provided the practitioner with advanced mechanisms to describe complex applications. Some of the recent efforts in the field tried to address important issues in open research areas, ranging from agent-based modeling, multiresolution/hierarchical models, hybrid models, and composability. We show how to address some of these issues through the application of a formal modeling and simulation technique and its application to the domain of defense applications. Our efforts consider the construction of multimodels, including components that can be defined as spatially-shaped models, using the Cell-DEVS and DEVS formalisms. DEVS is a mathematically sound framework in which a system is modeled by dividing it into a number of components (each of them having a discrete state and interacting with the environment via input/output ports). Cell-DEVS is an extension to DEVS that formulates the execution of cellular models with explicit timing delays. We show how these concepts can be applied to different defense-related spatial models, including a radar transmitter/receiver, a target-seeking device, and land battlefield models.

Keywords: DEVS, Cell-DEVS, cellular models, spatial models

1. Introduction

In recent years, a wide range of novel techniques became popular for developing defense modeling and simulation (M&S) applications. As discussed by Palmore [1], there are obvious reasons for using simulation in this area: although warfare is common, we cannot just generate conflicts to study the results of different combat strategies, weapons, new equipment, or advanced technologies. In addition, during warfare it is complex to obtain real data and make accurate observations. Likewise, making deliberate changes in the face of combat is extremely difficult [1]. Traditional analytical models cannot cope with the level of complexity of the systems of interest in this field, making M&S a useful tool, as it provides means for better understanding and analyzing the underlying phenomena, permitting evaluation of combat

situations, equipment, training, and logistics. Using simulation, users can make decisions by observing simulated results using realistic scenarios, examining the implications of change [2].

Current approaches do not suffice to achieve success in the complex models needed for defense applications. The feasibility of applying existing techniques, methodologies, and tools for modeling of information operations and C4ISR can be very mixed, mostly due to the lack of common, clear definitions and language to communicate ideas and concepts [1]. One of the main problems is derived from the advent of distributed simulation techniques and middleware (HLA, DIS, CORBA, SOA, etc.), which made possible reusing and integrating simulation artifacts in geographically remote areas. Consequently, in recent years we have witnessed a tremendous amount of research focused on advancing the theoretical foundations of simulation science to achieve these goals [3].

As discussed by Davis and Zeigler [4], we still lack the ability to apply the necessary theory, tools, and primers for building defense applications, although there are

insights in the literature that provide a foundation. Some of the recent efforts in the field tried to address important issues in the following open research areas for defense applications: agent-based modeling, advanced visualization methods, multiresolution modeling, model abstraction, hierarchical modeling, advanced simulation paradigms, automated model verification, dynamic structure M&S, multimodeling/hybrid modeling, and composability. In this article, we show how to address some of these issues through the application of a formal modeling mechanism and its application into the domain of defense M&S. The proposal is based on sound theoretical techniques, which involve mathematics and software engineering. We show a development of these ideas and we use them in implementing different military models.

Our efforts consider the construction of multimodels, including components that can be defined as cell spaces. The Cellular Automata (CA) formalism [5] is one of the techniques that have been widely used to describe complex systems with these characteristics. CA evolve by updating the state of every cell in the space synchronously and in parallel, by using a function that executes locally in each cell. The discrete-time nature of this formalism constrains the precision and efficiency of the simulated models. Likewise, CA cannot be easily composed or hierarchically integrated into a multimodel, or dynamically change their structure or behavior. Instead, Cell-DEVS [6] allows addressing of these issues. Cell-DEVS models are defined as a space composed of individual cells that can be coupled to form a complete cell space. Each cell is specified using DEVS [2], and it is defined by very simple rules and explicit timing delays. Atomic cellular models can be coupled with others defined in different specification languages/formalisms, forming a multicomponent model. DEVS is a hierarchical and modular M&S framework, based on systems theory.

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.

These techniques were successfully applied to address the open questions discussed early in other fields of application. Using DEVS and Cell-DEVS, we can construct agent-based models of a spatial nature (which are easily integrated in advanced visualization environments). We use a hierarchical modeling mechanism, which showed that it could be applied for multiresolution modeling, and define multiple submodels at different levels of abstraction. Integration of multiple views for each submodel is

possible, allowing the combination of different models in an efficient fashion. As the models are built as mathematical entities, we can prove basic properties regarding the structure and behavior of the models, while having a sound basis to build simulation tools according to the formal specifications. Models can be integrated into multiparadigm simulations (as different authors have mapped different discrete-event formalisms into DEVS, including Petri nets, state machines, state charts, queuing models, etc.). Thanks to recent advances in the field, we can also build hybrid multicomponent models, including continuous subcomponents defined by varied techniques that showed that they could be mapped into DEVS (ODEs, PDEs, bond graphs, Modelica, etc.). Because of the modularity of the approach, a wide variety of on-line control elements—including not only classic control mechanisms, but also neural networks, fuzzy logic, or expert systems—can be utilized. Cell-DEVS simulations are also more efficient than CA in terms of the memory use and computational power, as each cell in Cell-DEVS is only activated when it receives an input from its neighbor that is supposed to change its current state.

We carried out different experiments using the CD++ toolkit [7], a simulation engine that can be used to execute DEVS/Cell-DEVS models, which is a reliable engine that supports different platforms such as stand-alone, real-time, and parallel environments. In the following sections, we will discuss how this tool has been used to build different defense applications based on earlier successful efforts, and we will discuss how these methods and tools can be applied to address the different aspects introduced in this section.

2. Background

As mentioned in section 1, current military applications need further advances and research in numerous important fields.

- *Agent-based modeling*: This approach is based on the M&S of very complex systems as integrated by multiple agents that interact with each other (and with the surrounding environment) using very simple local rules. In the context of military applications, agents can include explicit decision mechanisms to make adaptive choices, instead of just following predetermined courses of action. Agents provide alternative approaches, ranging from extreme decentralization (small units with clear mission objectives) to more traditional centralized control [8].
- *Advanced visualization methods*: The goal is to provide a deeper real-world understanding and to help in exploring the large set of numerical

data produced in the simulation execution, which is a concern for model validation. It also allows the creation of mechanisms to exploit human capabilities.

- *Multiresolution modeling*: This technique, which permits combining models at different levels of resolution, proved to be very useful as it allows us to model the fact that we interact with the world at many different levels. As computing power has increased, multiresolution modeling allows analysis with models at one level of resolution, but occasionally calls to higher-resolution modules [3]. In general, low-resolution models are useful to understand the system as a whole, permitting the consideration of general problems and analysis of different choices available in a more abstract way. We also need these models when there is not enough detailed information, or when computational costs are extremely expensive for a higher resolution model. High-resolution models, instead, are useful for understanding the underlying phenomena in detail and reasoning about them with detailed knowledge. This allows the representation of varied fidelity, ranging from detailed (engineering level) to more aggregated models (theater/campaign level) [8]. This also allows the use of high-resolution information, and analysis of detailed behavior with the right level of accuracy. It has been suggested that it is crucial to design military models to produce integrated families crossing levels of resolution [4].
- *Model abstraction*: The concept of model abstraction is closely related with the idea of multiresolution modeling. We need to provide mechanisms for describing the basic behavior of a model without all the details. A model must capture the essence of the behavior of the real system of interest at the right level of detail (abstraction) [3].
- *Hierarchical modeling*: These techniques, in which you can replace a model by an equivalent one constructed as a multicomponent, have shown to be able to link the concepts of multiresolution modeling and model abstraction [2]. In many cases, we need a detailed model of the system of interest, which results in a simulation with thousands of entities. In those cases, model execution can be unfeasible; therefore, it is useful to aggregate the submodels, integrating detailed subcomponents into a higher level entity with fewer details. This hierarchical composition should allow keeping consistency when compared with the multimodel component. On the other hand, sometimes it is necessary to

replace a coarsely modeled entity with a more detailed version. This requires mapping all the input/output associated with each version so that their interconnections to the rest of the system can be still resolved [8].

- *Advanced simulation paradigms*: Traditional modeling paradigms (discrete-event, continuous system, Monte Carlo, etc.) are being slowly replaced by new modeling techniques. Gore [8] presented a non-comprehensive list of advanced techniques that includes object-oriented simulation, qualitative/fuzzy simulation, generative analysis, multimodeling/multifaceted modeling, Petri nets, neural nets, parallel/distributed simulation, concurrent simulation, web-based simulation, system dynamics modeling, adaptive/heuristic simulation, and man/hardware-in-the-loop simulation. Models and approaches such as CA, fuzzy logic, and neural networks are seen as useful paradigms for this field of application, as they can generate complex behavior from sets of relatively simple underlying rules. Using these techniques, we can find emergent behavior in a complex adaptive system without the need to include central control mechanisms/equations. Instead, basic bottom-up rules will define the higher level interactions of the components [8].
- *Automatic model verification*: The use of formal modeling techniques permits automating model verification. Kim et al. [9] presented an approach for discrete-event modeling based on an operational specification for the behavior of a model and an assertional specification for its temporal properties. A model's verification is based on a language acceptance checking mechanism. In Wainer et al. [10], we presented an attempt at adding automated verification capabilities to the CD++ toolkit. Specifically, automated rule verification, based on meeting basic logical properties in cellular models and coupled model definitions, were included. We also created a mechanism for automating the verification of multicomponent model coupling. Finally, we automated the creation of test case data to generate test inputs and collect the outputs. Experiments did show that such infrastructure could indeed help the designer to find defects in the models.
- *Experimental Framework*: The idea of the experimental framework (EF) [2] can be regarded as a vital component of the simulation setup phase. An EF permits documenting of objectives and issues to be addressed by the modeler

into conditions to run the experiments [11]. This allows the user to set up completely an experiment involving multiple executions of a single simulation with parameters changed for each run, or execution of multiple simulations with varying parameters [3]. EFs permit automating and documenting any choices made by the simulationist (e.g., the level of resolution and accuracy used). In Wainer et al. [10] and Labiche and Wainer [12], we discussed how to relate the concept of EF to software engineering techniques. We proposed a mechanism for integrating software testing techniques with EFs for the verification and validation of DEVS models, discussing open research paths for this field.

- *Dynamic structure M&S*: In many cases, system structure changes in the course of time. For example, a battle communication system might experience failures, upgrades on the equipment used for each of the nodes, variations in the bandwidth, etc. Such dynamic behavior is crucial for a military organization. Although significant research has been done on such techniques [13], most existing simulation languages do not support them [4].
- *Multimodeling/Hybrid modeling*: simulations should be able to include both continuous and discrete-event model components (hybrid models). For instance, the behavior governing the physics of a missile is described with differential equations (continuous modeling technique), while the missile digital control system might be better modeled using a discrete-event formalism. In the last few years, different approaches developed tried to simulate continuous systems under the discrete-event paradigm. This presents some advantages over discrete-time simulation, including reduction of the number of calculations for a given accuracy [14] and seamless integration of complex systems composed by both continuous time and discrete-event paradigms. The idea of this method, called quantized systems theory, is based on the DEVS formalism combined with quantization of the state variables obtaining a discrete-event approximation of the continuous system [15]. Spatial notions can provide extra facilities for understanding and visualizing the resulting simulation. For example, it would be possible to incorporate terrain models using GIS information.
- *Composability*: This is related to the fact that multiple models, simulations, and equipment might need to be put together in (both locally and

distributed) integrating a variety of components. Davis and Anderson [16] discussed exhaustively why we still could not answer the basic question of what factors determine what can be composed when, with how much expense and risk. They claim that composing very large models often requires lengthy and expensive efforts, most of which go into understanding and modifying components and interfaces to ensure validity. They show that the ability to develop composable systems for modern military operations will depend on advances on many fronts (including those presented in the previous bullets in this article): formal languages for describing models, representations suitable to effective communication and transfer while permitting the composition of models developed in different formalisms or representations, means for model abstraction and multiresolution modeling, the ability to create hybrid simulation models, explanation mechanisms (including agent-based models), and advanced means for human-computer interaction, including human behavior in virtual reality [16].

Our current research efforts are focused on how to achieve these goals, which would improve the construction of military M&S applications. We want to provide means for formal model construction, including multimodel applications in which spatial models can be defined and integrated with non-spatial components. Previous efforts have focused on solving these problems by using the Cellular Automata (CA) formalism, a widely used technique to describe complex cell spaces. CA evolve by executing a local transition function that updates the state locally in each cell, based on the current state values of the present cell and its neighbors. Conceptually, these local functions are computed synchronously and in parallel.

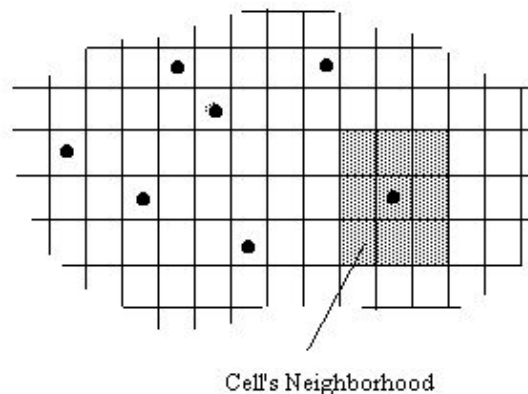


Figure 1. Sketch of a cellular automaton

CA have shown some success in modeling defense applications. An early effort in this field [17] showed how to build an agent-based combat simulation using CA to show tactics as an emergent behavior. There have been other similar efforts, including the U.S. Marine Corps' Project Albert [18], which explored how to build highly realistic models only considering the simplest dynamical variables. Champagne [19] presented a CA simulation of the U-boat war in the Bay of Biscay (between German U-boats and Allied aircrafts). He presented the results from two 6-month intervals of the operations and compared them to historical outcomes. The results indicate that the model was capable of reproducing historical outcomes for the two scenarios.

Different authors have used CA to model land battlefields. The idea is to model the interaction between two armies while each one is trying to achieve its goal (i.e., attacking the enemy's base or defending its own). The model presented by Woodcock et al. [17] introduced the idea of using CA to understand the complexity of a battlefield model. Ilachinski [18] presented the use of ISAAC (irreducible semi-autonomous adaptive combat agent) to create a model of software agents, each of which tried to mimic the behavior of a primitive combat element (soldier, tank, transport vehicle, etc.). ISAAC is a modular agent-based modeling system for experimentation using CA. Each agent includes specific characteristics: i) *doctrine* (a default local-rule set specifying how to act in a generic environment), ii) *mission* (goals directing behavior), iii) *situational awareness* (sensors generating an internal map of environment), and iv) *adaptability* (an internal mechanism to alter behavior and/or rules).

Das [20] presented a CA dealing with two opposing enemies: A (the friendly side) and K (the adversary). The CA models the evolution of an adopted scenario where A's strategy is to neutralize K's offensive. An effects-based strategy is formulated: instead of seeking the traditional purely militaristic solution, A responds to K's actions on all fronts using military, diplomatic, and socio-economic means [20]. A imparts a large number of small disturbances to K's military, political, and socio-economic establishments (randomly), producing minor effects. However, higher-order effects of these small disturbances accumulate to produce large-scale, cascading avalanches in an unpredictable fashion. Ilachinski [21] presented new techniques for building a behavioral military simulation, showing that equation-based models are not well suited to capturing the individual evolution of entities in these complex scenarios. He used entity-based models based on CA achieving much more realistic results, because CA makes computational complexity treatable, as the model rules are relatively simple. His environment is sufficient to capture much of the complexity of warfare.

Analysis of warfare data done by Lauren [22] provides evidence that intensity of conflicts obeys a fractal dependence on frequency. He showed how a CA used to describe modern maneuver warfare produces casualty distributions with fractal properties. He quantified the difference between CA and more traditional combat models (based on the physics of military equipment) [22].

Although these results are promising, CA have several problems. As shown in Wainer and Giambiasi [23], the discrete-time nature of the formalism constrains the precision and efficiency of the simulated models. Furthermore, it is usual that several cells do not need to be updated in every step, wasting computation time. These problems can be solved using a continuous time base, providing instantaneous events that can occur asynchronously at unpredictable times. In addition, CA cannot be easily composed with models defined in other formalisms, and they cannot be dynamically changed. Instead, Cell-DEVS [6] can address these issues. CA models are defined as a space composed of individual cells that can be laterally coupled to form a complete cell space. Each cell is a continuous time model, defined by very simple rules and a few parameters. Complex timing definition is overruled due to the use of different delay functions. The atomic cell models can be easily coupled with others, forming a multicomponent hierarchical model. Cell-DEVS atomic models can be described as in Figure 2.

Each cell uses N inputs (from its neighborhood or from other DEVS models) to compute its next state. These inputs, which are received through the model's interface, activate a local computing function (τ). 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.

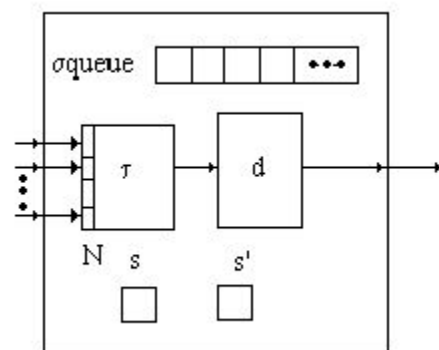


Figure 2. Cell-DEVS atomic model

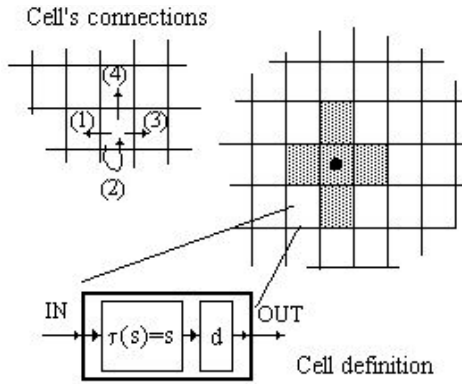


Figure 3. Cell-DEVS coupled model

Once the cell's behavior is defined, a coupled Cell-DEVS can be created by putting together a number of cells interconnected. A sample Cell-DEVS coupled model is presented in Figure 3. 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.

Cell-DEVS models are based on the DEVS formalism [2], a framework for M&S of discrete-event systems. DEVS provides an 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 4.

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* (λ). Then, an *internal transition function* (δ_{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 (δ_{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 5.

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

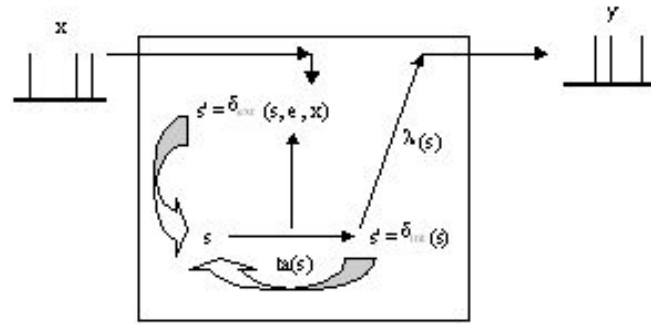


Figure 4. Informal definition of an atomic model

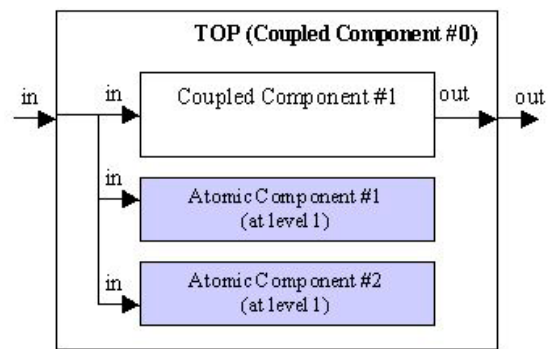


Figure 5. Informal description of a coupled model

between models and the interface with the external world.

CD++ [7] is an M&S 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 multicomponent 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 [24], and a parallel simulator [25].

Defining models in C++ provides the users with flexibility to define the model's behavior. Nevertheless, a non-experienced user can have difficulties in defining models using this approach. Graphical model specification also improves the interaction with stakeholders and users, while allowing the modeler to think about the problem in a more abstract way. Therefore, we have used an extended graphical notation to allow definition of an atomic model's behavior. Each

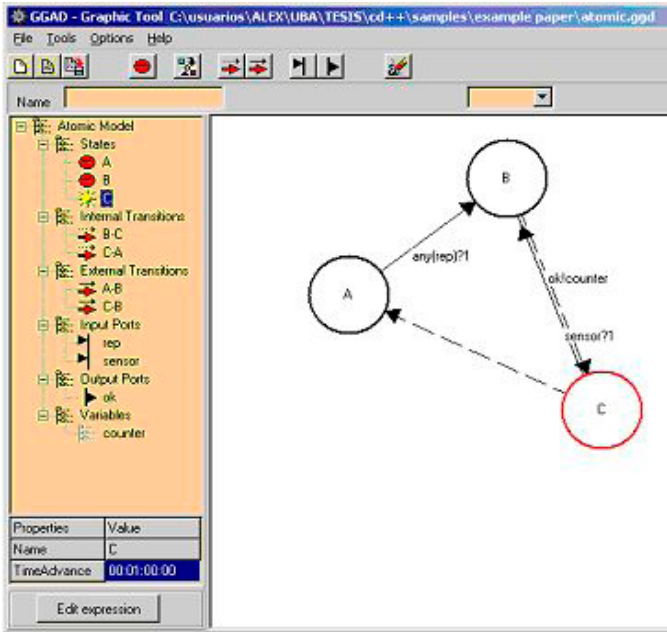


Figure 6. An atomic model defined as a DEVS graph

model is defined by a unique identifier, and states are represented by vertices (bubbles) in a directed graph. Each bubble includes an identifier and a state lifetime.

Figure 6 shows a simple atomic model including three states: A, B, and C. Dotted lines represent internal transitions, while full lines define external transitions. In this case, if the model is in state A and it receives an external event through the *rep* input port (shown in the left panel) the *any* function is evaluated. If the result of this evaluation is 1, the model changes to the state B. While in B, the model waits its lifetime to be consumed. It then executes the output function, which will send the value of the intermediate state variable *counter* through the output port *ok*. After that, the internal transition function executes, and the model changes to the state C.

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 the *DELAY*. 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.

3. Defining Defense Applications

In this section, we discuss how to create a few simple models that address the creation of DEVS and Cell-DEVS models in CD++. The first model represents an unmanned aerial vehicle (UAV) built using Cell-DEVS. The UAV traverses a 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.

As we can see, we have four different valid states for a cell: empty, a UAV is occupying the cell, or the cell contains a static/moving obstacle (each represented with a different discrete value). Each agent has different movement rules (the UAVs move in north/south/east/west directions, while the moving obstacles only move to the north). In order to specify the model in CD++, we need to define the cell space shape, size, and the rules governing the model execution. The first portion of the coupled model defines the cell-space geometry and initial values as shown in Figure 8.

As shown in Figure 8, the cell space is composed of 20×20 cells with a transport delay of 100 time units and initial values as defined by the *InitialRowValue* statement. These initial values show the states on each of the cells, according to Figure 7. The neighborhood shape covers the direction in which the UAV is moving.

Figure 9 shows part of the rule definition of the static obstacles, UAVs, and moving obstacles. The *noFlyZone9-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 *MovingTargetRule* implements a moving obstacle from south to north.

| | Empty Cell | UAV | Moving Obstacle | Static Obstacle |
|----------|------------|----------|-----------------|-----------------|
| Color | | | | |
| Movement | None | ⬆️⬇️⬇️⬆️ | ⬆️ | None |
| State | 0 | 1 | 5 | 9 |

Figure 7. UAV state values

```
[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 00000000099900000000
initialrowvalue : 0 10010100001000010000
initialrowvalue : 15 00000000900000000000
```

Figure 8. UAV coupled model specification

```
[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 9. UAV rule definition

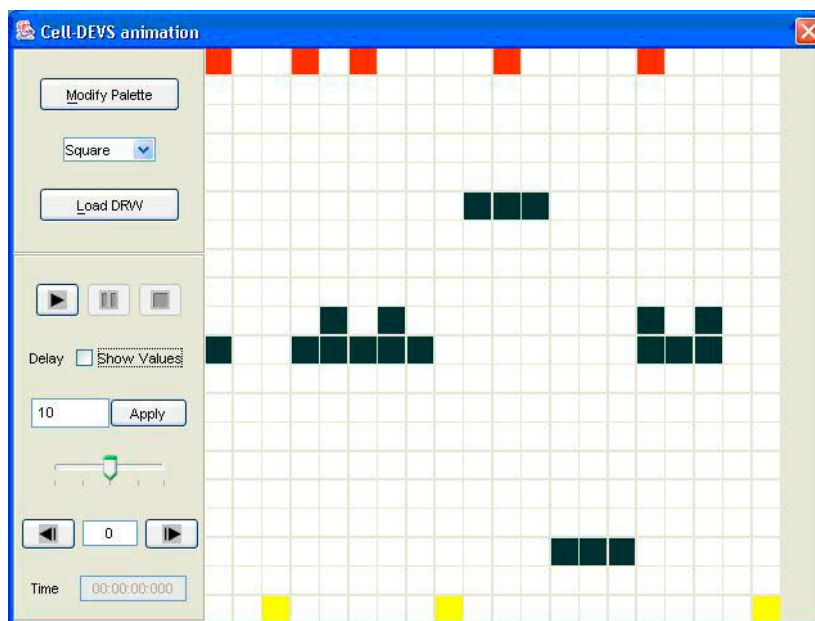


Figure 10. Initial allocations of UAVs and obstacles

Figure 10 shows a snapshot of the execution of this model with initial allocations of UAVs 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).

This example shows some of the basic aspects that we can cover with respect to the goals defined in section 2. We can build agent-based models with ease using a spatial approach (which runs with high performance using a discrete-event simulation engine). The spatial nature of the model permits easy integration with visualization engines. The application is built using an advanced simulation paradigm that can be combined with basic automated facilities for model verification. For instance, we can guarantee that the simulator execution is correct, as it has been built using DEVS and Cell-DEVS simulation algorithms, which were proven to be correct. Likewise, we can use basic logic results applied to the model's rules in order to be able to verify correctness (in terms of completeness of the rules defined, existence of ambiguous rules, or undefined status, as shown in [10]).

A second example we will introduce presents a transmitter/receiver model for a radar system [26]. The model was developed in [7] to model the synchronization effect between radar transmitter and receiver. When using scanning radar receiver, the interception of radar signals can be severely limited if the scan rate of the receiver becomes synchronized with a radar transmitter. The goal is to generate a receiver scan pattern that limits this effect, as it seriously degrades the probability of interception (POI) for the receiver. Synchronization occurs when a particular transmitter sends out radar pulses periodically, with the receiver scheduled to scan periodically in such a manner that the receiver is never "listening" when the transmitter is transmitting. Radar transmitters transmit on a particular frequency (for specified duration), with a particular pulse rate, azimuth, and beam width. *Scanning radar receivers* receive on a tuned frequency (for a specified duration), with a particular azimuth and beam width, and have a "tuning time" associated with the change from one listening frequency to another. The sequential operation of the receiver that defines the tuned frequency, listening time, azimuth, and beam width is specified by a "scan pattern."

Receivers can communicate with each other, with each receiver notifying the other receivers about radar transmitters that have been detected. Each receiver is connected to a simple communications bus, and it maintains a tracking table containing all the information about the currently known transmitters.

This model allows us to show how to address some of the remaining aspects discussed in section 2. The

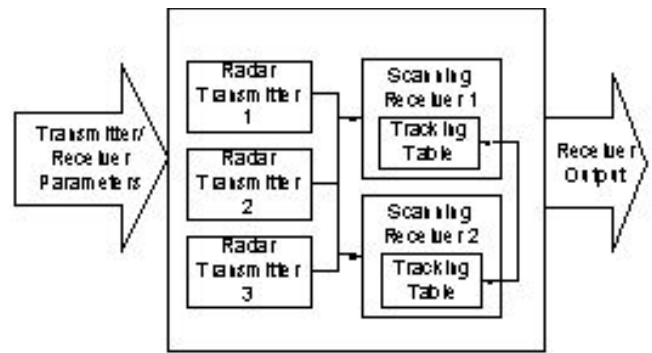


Figure 11. Structure of the radar transmitter/receiver model [26]

model is defined using a hierarchical specification, in which we can construct submodels at different levels of abstraction; i.e., we can be interested in studying the behavior of Radar Transmitter 3 in detail, or we can ignore such level of detail and deal with the external inputs/outputs only. If needed, the radar transmitters could be specified at different levels of resolution. (If the input/output interfaces remain unchanged, the resolution of the data transmitted and the amount of computation of each transmitted can be easily modified.) By connecting an EF to this model (for instance, creating transmission/reception parameters and studying the receiver's outputs for each of the inputs), we can automate the experimentation phase (creating multiple executions of a single simulation with parameters changed for each run, or execution of multiple simulations with varying parameters). Such an EF can be also used to document objectives and issues to be addressed by the modeler. If the dynamic structure DEVS modeling formalism [13] is used, we can change the structure and behavior of the submodels in runtime.

In order to create and execute this model, the first step was to identify and define each one of the model components. Once identified, a DEVS atomic model was built for each subcomponent. As an example, the tracking model is presented, which is responsible for maintaining the list of transmitters that are known to the local receiver.

$$\text{Scanning Receiver} = \langle S, X, Y, \delta_{\text{int}}, \delta_{\text{ext}}, ta, \lambda \rangle$$

$$S = \{ \text{Scan, Signal_Detected, Process_Signal, Notify} \}$$

$$X = \{ \text{ext_signal} \}$$

$$Y = \{ \text{notify, detected_signal_properties} \}$$

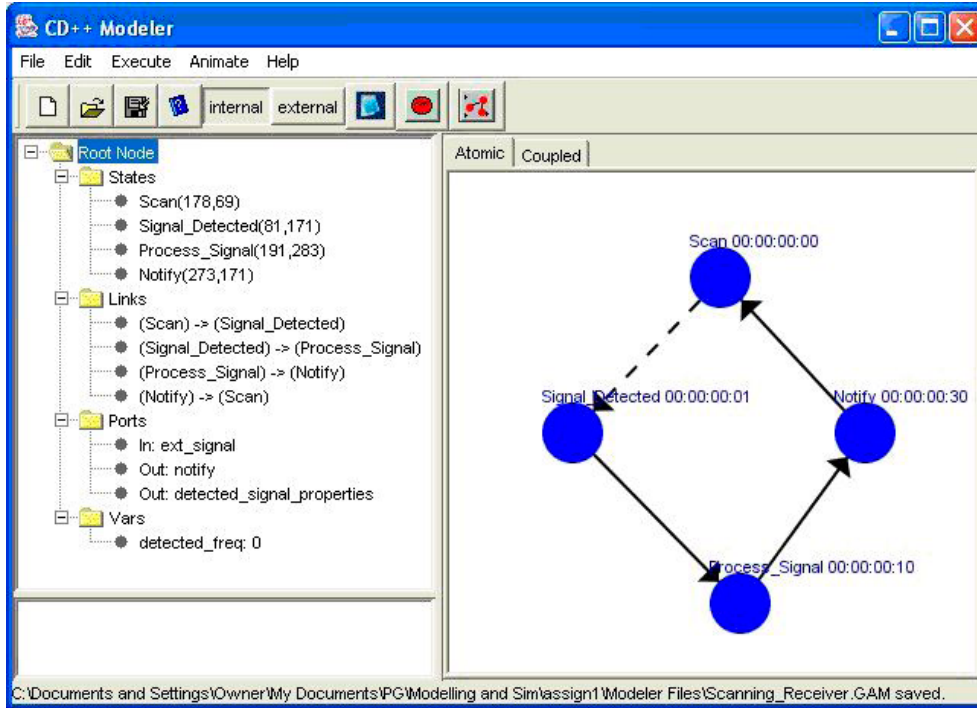


Figure 12. Specification of the tracking table model

$$\delta_{\text{int}} = \left\{ \begin{array}{l} \delta_{\text{int}}(\text{Signal_Detected}) = \text{Process_Signal}, \\ \delta_{\text{int}}(\text{Process_Signal}) = \text{Notify}, \\ \delta_{\text{int}}(\text{Notify}) = \text{Scan} \end{array} \right\}$$

$$\delta_{\text{ext}} = \left\{ \delta_{\text{ext}}(\text{Scan}, \text{ext_signal}) = \text{Signal_Detected} \right\}$$

$$ta = \left\{ \begin{array}{l} ta(\text{Scan}) = \infty, \\ ta(\text{Signal_Detected}) = \text{DETECTION_TIME}, \\ ta(\text{Process_Signal}) = \text{PROCESS_TIME}, \\ ta(\text{Notify}) = \text{NOTIFY_TIME} \end{array} \right\}$$

$$\lambda = \left\{ \begin{array}{l} \lambda(\text{Signal_Detected}) = \text{notify}, \\ \lambda(\text{Process_Signal}) = \text{detected_signal_properties} \end{array} \right\}$$

This model evolves through different states (S): scan for signals, a signal has been detected, a signal is being processed, notify about the signal reception. The model changes from one state to the other by executing the transition functions. As seen in the external transition (δ_{ext}), when the scanning receiver detects a signal ($\text{ext_signal} \in X$) it changes its state from *Scan* to *Signal_Detected*. As we can see in the definition of the ta function, after the *DETECTION_TIME* is consumed, the model executes the output function (λ) and, in this case, a *notify* output is issued. Then, the internal transition (δ_{int}) is activated, and the

model changes to the *Process_Signal* state (which is held during *PROCESS_TIME* time units, as defined in ta). At this point, the *detected_signal_properties* are outputted, and the internal transition function makes the model change to the *Notify* state. After *NOTIFY_TIME* time units, the internal transition executes, and the model returns to the *Scan* state (a passive state, as its ta function is infinity).

The model was subsequently built in CD++ using the state machine specification presented in Figure 12. The four states of the model are immediately apparent. External transitions are displayed as dashed lines, with internal transitions as solid lines. The input and output ports are visible in the tree diagram.

A different model, built using Cell-DEVS, describes the behavior of a simple vehicle, which seeks a target [27]. As shown in Figure 13, the seeker steers a vehicle toward a specified position in global space. This behavior adjusts the vehicle so that its velocity is radially aligned toward the target. This model permits us to show how to build a model with continuous elements (speed, acceleration, and the equations relating them) combined with a spatial-based approach. It also permits showing the definition of a model at different levels of abstraction.

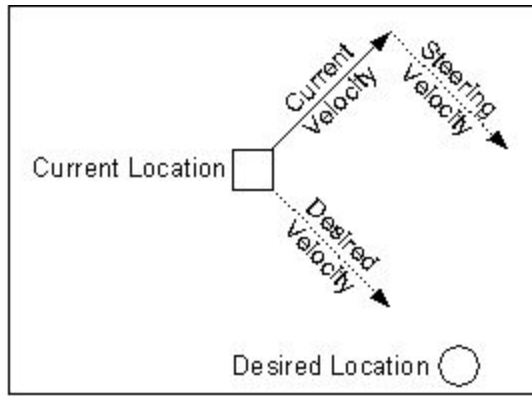


Figure 13. Informal behavior of the Seek model

Using the hierarchy of motion behaviors defined by Reynolds [27], the “action selection” of the seeker is specified by dictating the destination location. The simple vehicle model has the following attributes:

$\{mass \text{ (scalar)}, position \text{ (vector)}, velocity \text{ (vector)}, max_force \text{ (scalar)}, max_speed \text{ (scalar)}, orientation \text{ (} N \text{ basis vectors)}\}$, where $N = 2$.

The motion of the model is defined by

$steering_force = truncate(steering_direction, max_force)$,
 $acceleration = steering_force / mass$,
 $velocity = truncate(velocity + acceleration, max_speed)$,
 $position = position + velocity$;

and the new basis vectors by

$new_forward = normalize(velocity)$,
 $approximate_up = normalize(approximate_up) // \text{if needed}$
 $new_side = cross(new_forward, approximate_up)$,
 $new_up = cross(new_forward, new_side)$.

The seek behavior motion is defined by

$desired_velocity = normalize(position - target) * max_speed$,
 $steering = desired_velocity - velocity$.

To model the seek behavior using Cell-DEVS, it was necessary to create discrete states to represent the ‘current’ state of the simple vehicle. The following state variable was used (Table 1):

Table 1. Vehicle state assignment

| State | Description |
|------------------|---|
| Current Velocity | A state indicating a vehicle with no velocity, or motion in one of 8 directions: moving diagonally up and left (value = 1), up (2), diagonally up and right (3), left (4), stationary (5), right (6), diagonally down and left (7), down (8), diagonally down and right (9) |

The model uses the following neighborhood definition:

$$N = \{ (-2,-2), (-2,-1), (-2, 0), (-2,1), (-2,2), (-1,-2), (-1,-1), (-1,0), (-1,1), (-1,2), (0,-2), (0,-1), (0, 0), (0,1), (0,2), (1,-2), (1,-1), (1, 0), (1,1), (1,2), (2,-2), (2,-1), (2, 0), (2,1), (2,2) \}$$

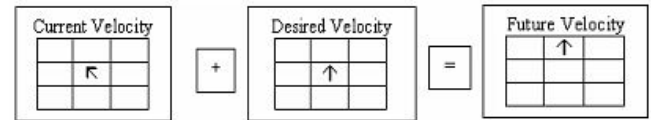


Figure 14. Definition of update rules

An input was provided to each cell to specify the desired velocity of the vehicle. The model rules detail the discrete motion that was implemented to simulate the effect of a desired velocity on a vehicle. Multiple combinations of actual and desired velocity could result in the same destination cell for a vehicle.

With the many combinations of velocities, the possibility for collisions is great. The neighborhood for each cell is dependent on its velocity. A simple priority is used to resolve any conflict when multiple vehicles want to move into the same cell. Stationary vehicles have the highest priority, “up and left” have the lowest, and “down and right” have the second highest.

The model was completely implemented in CD++ following the Cell-DEVS rule specifications, and it was tested initially using a single vehicle, with different initial velocities and different desired velocities. After all the rules were implemented, all possible velocities were tested in all possible desired velocities. Following that, different vehicles were used to simulate the collision avoidance scheme.

The following figures display the two state variables employed in the definition of the Cell-DEVS model (displayed side-by-side). The left-hand plane (mostly white) displays the current location and velocity of the three vehicles. The right-hand plane describes the “desired velocity vector field” of the vehicles. The “desired location” for all three vehicles is the center of the plane, and the “desired velocity vectors” steer them to that point. We can see how to include different levels of abstraction in the model. The right-hand

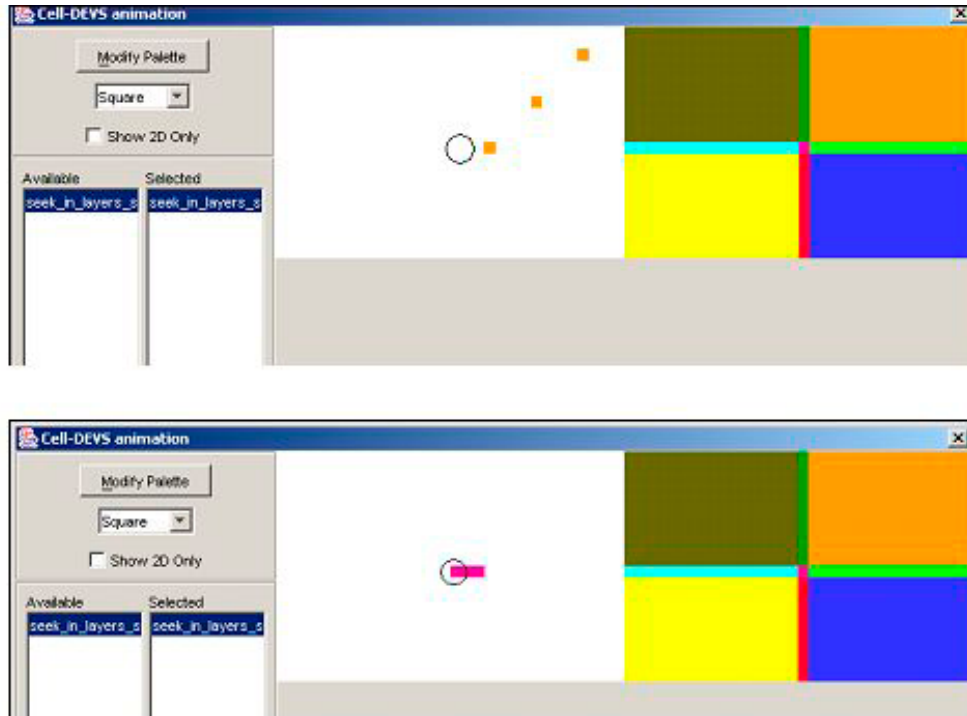


Figure 15. Three vehicles seeking the desired location [26]

plane contains detailed information about the desired velocity field and computes the related equations with a high level of precision, whereas the left-hand plane shows the current location of the vehicle using a discrete notation, including less information.

In Figure 15, three vehicles enter from the top-right corner of the plane, and they stop when they cannot move any closer to the “desired location.” The vehicles enter (at time 0, 500, and 900 ms) with a velocity different from the desired velocity, and each acts in accordance with the state transitions to “turn” to the desired velocity. At 1.2 seconds, the first vehicle enters a region with a different desired velocity. Note that the vehicle (and each subsequent vehicle) “turns” to the desired velocity.

4. Modeling a Land Battlefield

In this section, we will present an advanced model of land battlefield between two armies. The idea is to model the interaction between two armies in a battlefield while each one is trying to achieve its goal; this can be attacking the enemy’s base or defending its own base. This example will allow us to show how to represent advanced defense models with multiple resolutions, and how to use the available tools to achieve varied levels of abstraction.

We created a Cell-DEVS model representing a

combat battlefield, which is based on the ISAAC model presented in section 2 [28]. The model is represented by a 2-D CA. Each cell can be occupied by one of two kinds of troops: *red* or *blue*. Red and blue “flags” are also typically (but not always) positioned in diagonally opposite corners: a red flag in the red corner and a blue flag in the blue corner. A typical goal is to reach successfully the flag positioned in the diagonally opposite corner. Each soldier can be in one of three states: *alive*, *injured*, or *killed*. Injured troops can (but are not required to) have different personalities from when they were alive. By default, an injured soldier’s ability to shoot an enemy is equal to half of its ability when alive. In addition, if the soldier chooses its moves from among lattice sites within a distance of two or more from its current position, an injured soldier’s moving range is reduced to the minimum possible range of one unit. Up to 15 distinct groups of personalities, of varying sizes, can be defined. Each soldier has associated with it a set of ranges (sensor range, fire range, communications range, etc.), within which it senses and assimilates simple forms of local information, and a *personality*, which determines the general manner in which it responds to its environment.

We followed these ideas in order to model and simulate a land battlefield using Cell-DEVS. The model accounts for the major aspects in a modern battlefield (with some assumptions) in terms of the soldier state,

personality factor, situation awareness range, etc.

The model we built consists of a land battlefield between two armies; 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 system can be summarized as follows:

- A 2-D 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 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 toward 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.

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). Table 2 describes this representation.

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 the flag and dead soldiers), while 1 represents

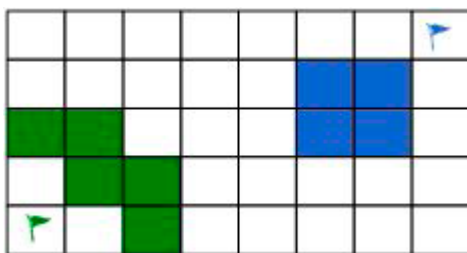


Figure 16. Possible troop allocations

Table 2. Fighter state assignment

| 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 |

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

Table 3 describes the fighting ability factor.

Table 3. Fighting ability states

| Status | Fighting Ability (FA) |
|--------|---|
| 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 in the range [0.45, 1] |
| 5 | Does not engage in fights 0.0 |
| -5 | Does not engage in fights 0.0 |

As we can see in Figure 17, when two or more soldiers engage in a fight, the outcome depends on the difference between their fighting abilities.

If a soldier is not surrounded by the enemy, he tends to move toward 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, 6); he will have

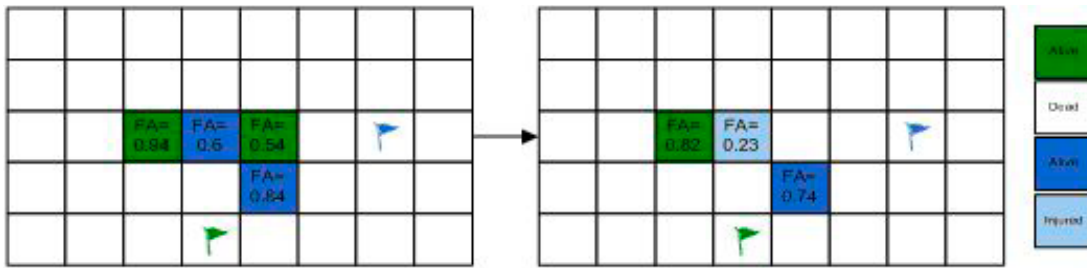


Figure 17. The effect of different FAs in a fight

two options, either to move to the east or to the south, as shown in Figure 18.

After deciding on the direction of the next step, the directions are assigned integer values according to Table 4.

The free-cell move-in factor is an integer number that is calculated for every free cell to resolve any

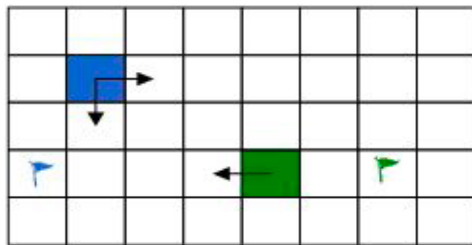


Figure 18. Movement directions

Table 4. Direction values

| Direction | Value |
|-----------|-------|
| North | 10 |
| East | 20 |
| South | 30 |
| West | 40 |

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. Figure 19 illustrates this point.

A different implementation computes the free-cell move-in factor by checking the 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. Figure 20 illustrates this point.

The model was implemented using CD++ (a detailed definition of the specification can be found in Madhoun and Wainer [28]). Each piece of information was implemented using a different layer, which resulted in a 3-D cell space. Each of the layers can address the submodel at different resolution and abstraction levels. The layers used to implement the model are as follows:

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

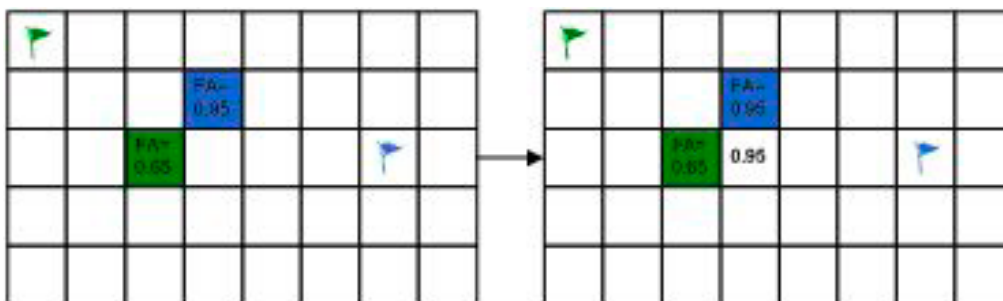


Figure 19. Free-cell move-in factor evaluation

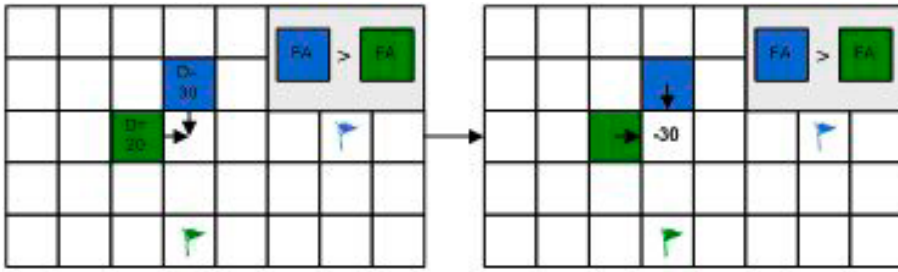


Figure 20. Free-cell move-in factor with intention

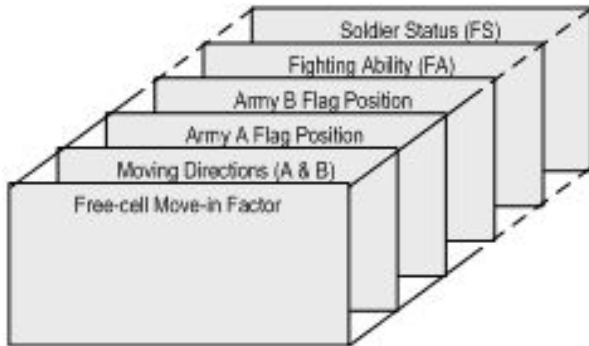


Figure 21. Cell space definition

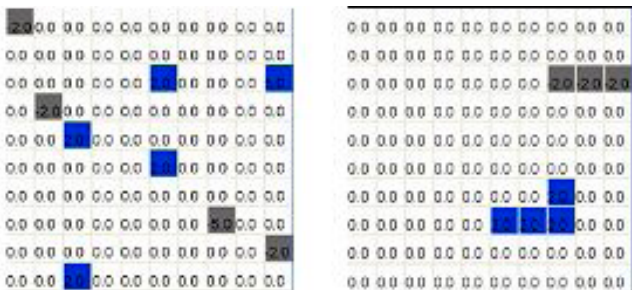


Figure 22. Testing movement rules

- Layer 2: flag position of army B. This information is needed for all the soldiers of army A to 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.

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 toward the enemy's

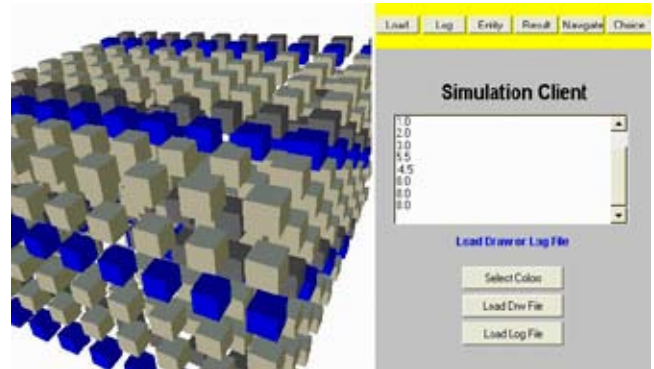


Figure 23. Multilayer display: execution results

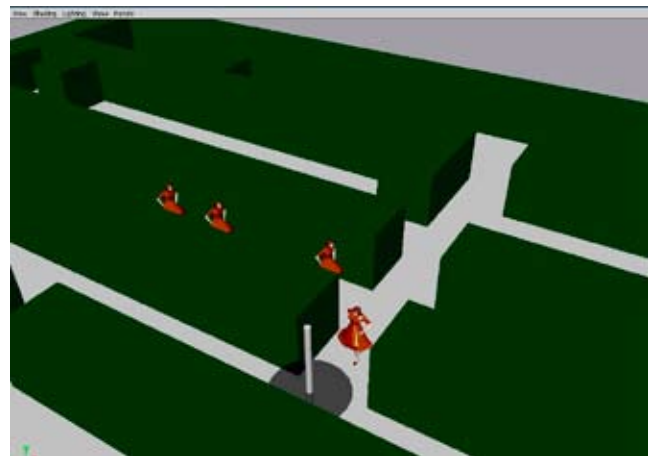


Figure 24. Displaying 3-D models in CD++/Maya

flag. Figure 22 shows the initial and final configuration of the army (one fighter of each army was killed in the battle; both armies eventually reached the flags).

Different tests were carried out, including several overall executions of the model. Figure 23 shows a 3-D visual result of the execution of the model, in which each of the layers previously discussed, is depicted.

Using advanced VR environments, as the one

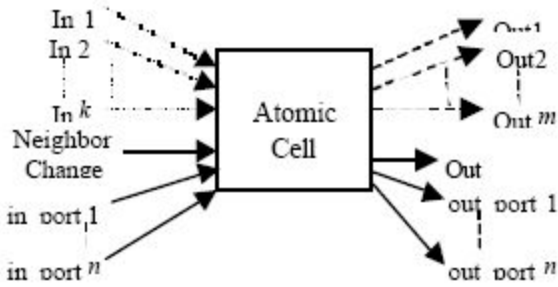


Figure 25. Multi-port cell

depicted in the Figure 24, we can build advanced realistic representations of the models of interest, allowing both better analysis capabilities and training facilities

The battlefield model was extended using new advanced facilities available in a recently developed version of CD++ [29]. 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 25.

The input/output ports connect each cell to all of its neighboring cells, so it is useful to represent information that needs 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 layers of cells to represent new pieces of information.

The original battlefield model was implemented using these new services, as a 2-D cell space with the following input/output ports:

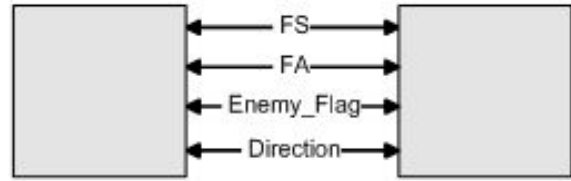


Figure 26. Multi-port connectivity between two cells

- **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;
- **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 the following:

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

```
#BeginMacro(move_from_west_factor)

rule : { ~direction := if { (0,-1)~direction = 20, -20, (0,0)~direction } ; } 100
{ (0,0)~fs = 0 and (0,0)~direction = 0 and
  (0,-1)~fa > if((-1,-1)~direction = 25, (-1,-1)~fa,0)
  and (0,-1)~fa > if((-1,0)~direction = 30, (-1,0)~fa,0)
  and (0,-1)~fa > if((-1,1)~direction = 35, (-1,1)~fa,0)
  and (0,-1)~fa > if((0,1)~direction = 40, (0,1)~fa,0)
  and (0,-1)~fa > if((1,-1)~direction = 15, (1,-1)~fa,0)
  and (0,-1)~fa > if((1,0)~direction = 10, (1,0)~fa,0)
  and (0,-1)~fa > if((1,1)~direction = 45, (1,1)~fa, 0) }

#EndMacro
```

Figure 27. Sample free-cell move-in factor rule

- Movement rules: they define the behavior of the soldiers when moving in the battlefield.

As an example of these rules, we present the new implementation of the free-cell move-in factor discussed earlier. Figure 27 shows the CD++ rule definition of one of the rules used to calculate the free-cell move-in factor. In this rule, the soldier from the west is examined to check if he intends to move to the cell and if his fighting ability is higher than all the soldiers in the cell's neighborhood who intend to move to that cell. In this case, the move-in factor is equal to (-20) to indicate that the cell will be occupied by a soldier coming from the west. After evaluating the move-in factor for the free cell, the next step would be the actual move of the concerned soldier from his original cell to the empty one. This is accomplished by a set of rules similar to the one shown in Figure 28. The *move_from_west* rules have two parts: the first part copies the soldier state (fighter status, fighting ability, etc.) from his original cell to the free cell and the second part clears the soldier state in the original cell.

Another set of rules used in the definition of the battlefield model, is the fighting rule shown in Figure 29.

The macro *fight_rule_1* in Figure 29 checks if the soldier (from army A) is in the neighborhood of an enemy soldier (from army B). Then, it checks if the soldier (from army B) has a higher fighting ability, and

in that case adds (-1) to the overall value of the macro for each such soldier.

The number generated by *fight_rule_1* is used in the main body of the rule (presented in Figure 30) to evaluate the following conditions:

- If a soldier in army A is injured (FS = 1) and is surrounded by enemy soldiers whose fighting abilities 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 abilities are higher than his, he will be dead, and his fighting ability will be assigned the value 0.
- If a soldier in army A is alive (FS = 2) and is surrounded by enemy soldiers whose fighting abilities are less than his, he will remain alive and will be assigned a 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 a 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 0.

```
#BeginMacro(move_from_west)

rule :-fs := (0,-1)-fs; -fa := (0,-1)-fa ; -direction :=0; -target_flag := (0,-1)-target_flag;
0
{ (0,0)-fs = 0 and ( (0,-1)-fs = 2 or (0,-1)-fs = -2) and (0,-1)-direction = 20
  and (0,0)-direction = -20 }

rule : ~fs := 0 ; ~fa := 0 ; ~direction := 0 ; ~target_flag := -1; } {Scf := 0; } 0
{ ((0,0)-fs = 2 or (0,0)-fs = -2) and (0,1)-fs = 0 and (0,0)-direction = 20
  and (0,1)-direction = -20}

#EndMacro
```

Figure 28. Move from west rule

```
#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 29. Fighting rules macros

```

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 30. Fighting rules

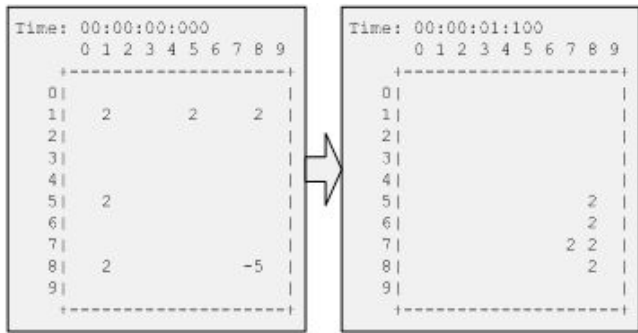


Figure 31. Testing movement rules

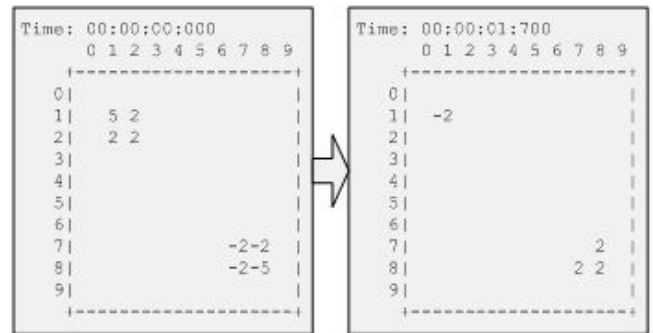


Figure 33. Overall test of the model

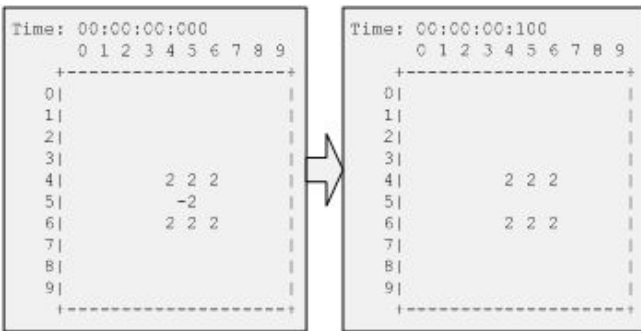


Figure 32. Testing fighting rules

The same rule is used for B soldiers when surrounded by A army soldiers by changing the corresponding soldier status values. The following figures show 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; see Figure 31.
- *Fighting rules*: In this scenario, the fighting rules are activated when the soldiers of both armies engage in a fight; see Figure 32.
- *Global test*: All of the rules are activated to test the overall behavior of the model; see Figure 33.

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 the following:

- Situation awareness of the soldier (neighborhood) was extended to include the eight surrounding cells. Hence, the soldier is able to attack and move diagonally as well as horizontally or vertically; see Figure 34.

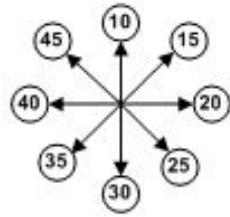
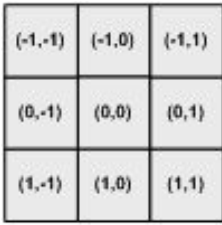


Figure 34. Extending the soldier's neighborhood to Moore's neighborhood

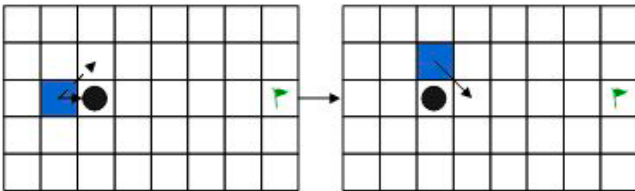


Figure 35. Obstacle avoidance example

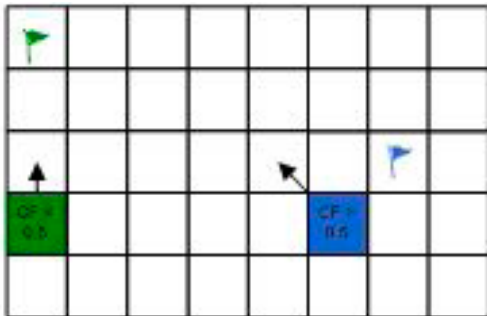


Figure 36. Effect of the courage factor (CF) on the soldier's behavior

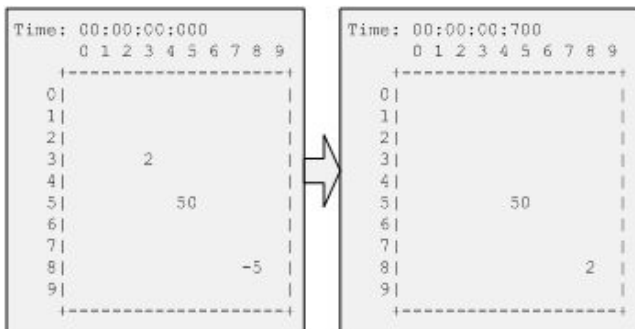


Figure 37. Testing the obstacle avoidance feature

- Obstacle avoidance: The soldiers are able to avoid obstacles (FS = 50) while moving toward the enemy's flag; see Figure 35.
- 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 toward his own base/flag; see Figure 36.

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; and
- The second one tests the overall behavior of the model after incorporating the courage factor (CF).

The results of these tests are shown in Figures 37 and 38.

The definition of these extended rules for the newly defined behavior took less than 3 person-hours, showing the adequacy of the tools to improve the models being created at a low cost in terms of modeling effort. We can see that changing the behavior of the agents involved can be done with little effort.

5. Multimodel Composition

In this section, we show how to compose some of the previously developed models (radar transmitter/receiver and seeker models) in order to show how to address some of the remaining issues discussed in section 2: how to build a multimodel integrating varied paradigms, and how to address basic composability issues. Integrating the models presented here with the battlefield models introduced in section 4 is straightforward (and is done in the same way as the rest of the examples in this section). This interaction is implemented at the model level, and no changes

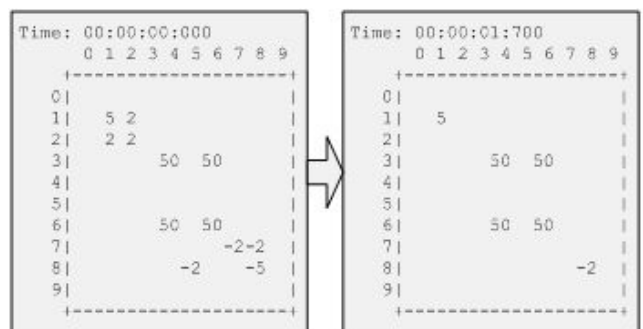


Figure 38. Testing the overall behavior of the model

were done to the simulation engine, as the models only communicate at the level of their interfaces. Let us consider, for instance, the existence of a new model, *Radar*. The radar model is prepared to scan a cell space according to a given frequency. Figure 39 shows how to integrate this new model with the two other models defined earlier in this section. These three models were built independently, but they can be easily integrated due to the modular nature of the DEVS interfaces.

The *Transmitter/Receiver* model is used to start radar scanning activities. Upon activation, the Radar will scan the field defined by the Seek Cell-DEVS model, and will generate two outputs: a reception signal for the Transmitter/Receiver, and a number of operator messages according to the values received in the field. The Seek model advances independently of the execution of the radar, because these models are built as discrete-event specifications, and each subcomponent progresses according to its own internal time base. In CD++, the coupled model defining the composition of the submodels can be defined as in Figure 40.

As seen in Figure 40, the *top* model is now integrated with the three original components. The coupling of the model was initially defined. Then, the definition of the *Seek* model is shown. The model produces outputs that can be used by the Radar model.

A *zone* in which the cells will generate outputs was defined (by using the *out-rule* definition). Finally, the Transmitter/Receiver model (*Tx-Rx*) included two new input/output ports in order to provide interaction with the Radar model. The Radar model is not defined in

the file, as it has been defined as a DEVS atomic model, and only the coupling with the other models needed to be defined.

6. Conclusion

We have shown how DEVS and Cell-DEVS techniques can be used to address fundamental problems existing when modeling and simulating space-shaped military applications. Both techniques are based on sound mathematical foundations that offer 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 if one follows DEVS or Cell-DEVS modular specifications. We have shown how different models can be easily integrated, while the separation of concerns between the model definition and simulation engine enables the modeler to concentrate on building the model without studying the internals of the simulator.

The methods presented allow automatic definition of cell spaces using the DEVS formalism. Integration of multiple views for each submodel is possible, allowing the combination of different models in an efficient fashion. The use of a formal approach allowed proving properties regarding the cellular models. It also provided a sound basis upon which to build simulation tools related with the formal specifications. Simultaneously, Cell-DEVS is more efficient than CA in terms of the memory use and computational

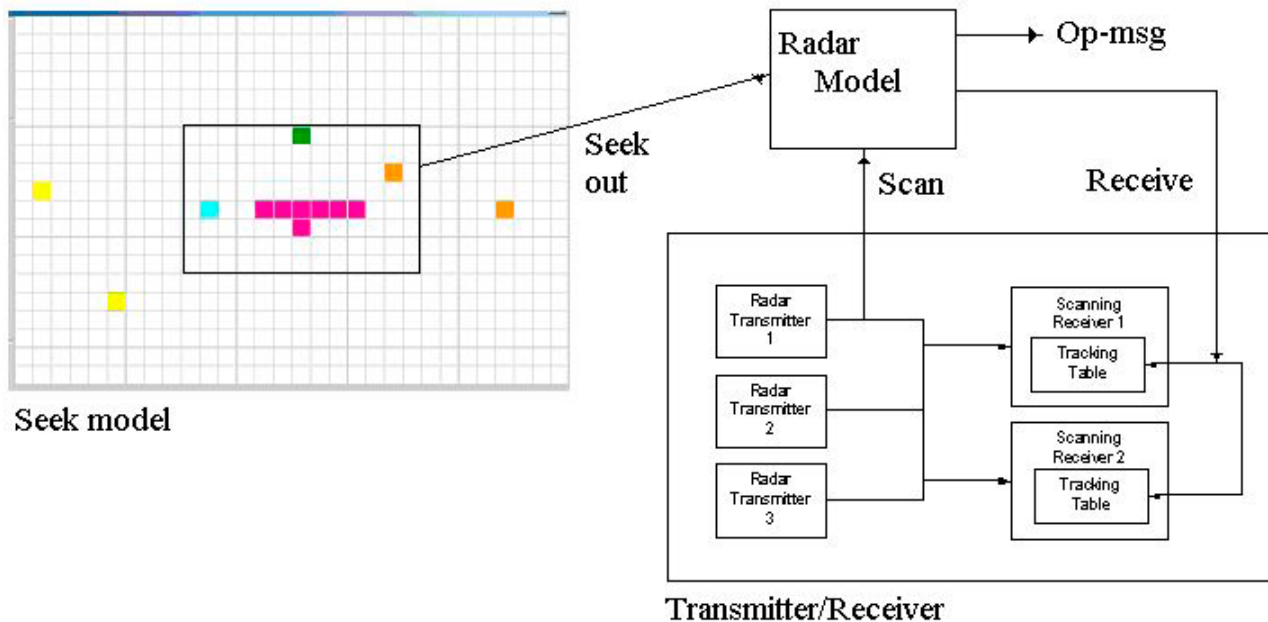


Figure 39. Multimodel composition

```
[top]
components : seek Tx-Rx radar@Radar
out : op-msg
link : seek-out@seek seek-out@radar
link : op-msg@radar op-msg
link : scan@tx-rx scan@radar
link : receive@radar receive@tx-rx

[seek]
type : cell
dim : (20,20)
delay : transport
border : wrapped
out : seek-out
neighbors : (-2,-2) (-2,-1) (-2,0) (-2,1) (-2,2)
neighbors : (-1,-2) (-1,-1) (-1,0) (-1,1) (-1,2)
neighbors : (0,-2) (0,-1) (0,0) (0,1) (0,2)
neighbors : (1,-2) (1,-1) (1,0) (1,1) (1,2)
neighbors : (2,-2) (2,-1) (2,0) (2,1) (2,2)

% Cells producing outputs
zone : out-rule { (10,0)..(19,19) }
link : outTRF@seek(10,0)..(10,19) seek-out
localtransition : out-rule

[out-rule]
rule : {(send(L, y-t-room)) 0 { t }

localtransition : move-rule

[move-rule]
% Rules which do not allow a move to occur (collision avoidance)
rule : 5 100 {
  % Moving up and left
  (
    ( ((0,0)=1) and ((0,1)=1) ) or ( ((0,0)=5) and ((0,1)=1) ) or
    ( ((0,0)=2) and ((0,1)=1) ) or ( ((0,0)=2) and ((0,1)=4) ) or
    ( ((0,0)=2) and ((0,1)=7) ) or ( ((0,0)=4) and ((0,1)=1) ) or
    ( ((0,0)=4) and ((0,1)=2) ) or ( ((0,0)=4) and ((0,1)=3) )
  )
  ... % Remaining moving rules
}

[Tx-Rx]
components: tr1@Transmitter tr2@Transmitter tr3@Transmitter netrx1 netrx2
out : notify1 notify2 notify3 scan
in : receive
...
% Same as the previously defined model
```

Figure 40. Defining a multimodel in CD++

power needed, and we were able to reproduce well-known applications with ease, showing that we can build advanced models based on existing applications effortlessly. As DEVS is a modular and hierarchal approach to modeling, it provides the means to easily connect the battlefield model to components in a model library. In addition, as each cell is activated only when it receives an input from an active neighbor, the level of activity is reduced and the model executes faster.

The examples presented show different aspects to consider when building DEVS and Cell-DEVS models for defense simulations. These techniques address several of the aspects discussed: we can construct agent-based spatial models (which are easily integrated in advanced visualization environments). The models are built using a hierarchical modeling mechanism, which showed that it could be applied to multiresolution modeling and to define multiple submodels at different levels of abstraction. Integration

of multiple views for each submodel is possible, allowing the combination of different models in an efficient fashion. The use of a formal approach allowed proving properties regarding the cellular models. It also provided a sound basis to build simulation tools related with the formal specifications. Models can be integrated in multiparadigm simulations and hybrid multicomponent models. This provides the basis for future exercises in advanced modeling and simulation efforts using these techniques.

7. References

- [1] Palmore, J. (Chair) "Mini-Symposium/Workshop Report. Warfare Analysis and Complexity." Military Operations Research Society. September 15-17, 1997. JHU/APL. Laurel, MD. <<http://www.mors.org>>.
- [2] Zeigler, B., T. Kim, and H. Praehofer. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.

- [3] Sisti, A., and S. Farr. "Modeling and Simulation Enabling Technologies for Military Applications." In *Proceedings of the 1996 Winter Simulation Conference*. Coronado, CA, 1996.
- [4] Davis, P., and B. Zeigler. "Technology for the United States Navy and Marine Corps, 2000–2035. Becoming a 21st-Century Force" Vol. 9. "Modeling and Simulation." National Academy of Sciences. <http://www.nap.edu/html/tech_21st/msindex.htm> 1997.
- [5] Wolfram, S. *A New Kind of Science*. Wolfram Media, 2002.
- [6] Wainer, G., and N. Giambiasi. "N-Dimensional Cell-DEVS." *Discrete Events Systems: Theory and Applications* 12, no. 1 (January 2002): 135–157 (Kluwer).
- [7] Wainer, G. "CD++: A Toolkit to Define Discrete-Event Models." *Software, Practice and Experience* 32, no 3 (November 2002): 1261–1306.
- [8] Gore, J. "Chaos, Complexity and the Military." National Defense University, National War College, Military Strategy and Operation Seminar D. Technical Report 96-E-61.<<http://www.ndu.edu/library/n1/96-E-61.pdf>> 1996.
- [9] Kim, T.G., S. M. Cho, and W. B. Lee. "DEVS Framework for Systems Development." In *Discrete Event Modeling & Simulation: Enabling Future Technologies*. Springer-Verlag, 2001.
- [10] Wainer, G., L. Morihama, and V. Passuello. "Automatic Verification of DEVS Models." In *Proceedings of the SISO Spring Interoperability Workshop*, 2002.
- [11] Zeigler, B. P., D. Fulton, J. Nutaro, and P. Hammonds. "M&S Enabled Testing of Distributed Systems: Beyond Interoperability to Combat Effectiveness Assessment." 9th Annual M&S Workshop, ITEA White Sands Chapter, 2003.
- [12] Labiche, Y., and G. Wainer. "Towards the Verification and Validation of DEVS Models." In: *Proceedings of the 1st Open International Conference on Modeling & Simulation*, Clermont-Ferrand, France, 2005.
- [13] Barros, F. J. "Modeling Formalisms for Dynamic Structure Systems." *ACM Transactions on Modeling and Computer Simulation* 7, no. 4 (October 1997): 501–515.
- [14] Zeigler, B. P. "Continuity and Change (Activity) Are Fundamentally Related in DEVS Simulation of Continuous Systems." *LNCS 3397/2005*, 1–17. New York: Springer-Verlag, 2005.
- [15] Zeigler, B. "DEVS. Theory of Quantization." DARPA Contract N6133997K-007, ECE Dept., University of Arizona, Tucson, AZ, 1998.
- [16] Davis, P., and R. Anderson. "Improving the Composability of Department of Defense Models and Simulations." RAND National Defense Research Institute, 2003.
- [17] Woodcock, A. E. R., L. Cobb, and J. Dockery. "CA: A New Method for Battlefield Simulation." *Signal* 42 (January 1988): 41–50.
- [18] Ilachinski, A. "Irreducible Semi-Autonomous Adaptive Combat (ISAAC): An Artificial Life Approach to Land Combat." *Military Operations Research* 5, no. 3 (2000): 29–46.
- [19] Champagne, L. "Bay of Biscay: Extensions into Modern Military Issues." In *Proceedings of the 2003 Winter Simulation Conference*, New Orleans, LA, 2003.
- [20] Das, B. "Effects-Based Operations: Simulations with CA." Technical Report DSTO-RR-0275, Command and Control Division, Information Sciences Laboratory, Australian Government, Department of Defence, Defence Science and Technology Organisation, AR-012-980, June 2004.
- [21] Ilachinski A. *Artificial War: Multiagent-Based Simulation of Combat*. World Scientific Press, 2004.
- [22] Lauren, M. "Fractal Methods Applied to Describe Cellular Automaton Combat Models." *Fractals* 9, no 2 (2001): 177–185.
- [23] Wainer, G., and N. Giambiasi. "Application of the Cell-DEVS Paradigm for Cell Spaces Modeling and Simulation." *Simulation* 71, no. 1 (January 2001): 22–39.
- [24] Glinsky, E., and G. Wainer. "Performance Analysis of Real-Time DEVS Models." In *Proceedings of 2002 Winter Simulation Conference*, San Diego, CA, 2002.
- [25] Troccoli, A., and G. Wainer. "Implementing Parallel Cell-DEVS." In *Proceedings of the Annual Simulation Symposium*, Orlando, FL, 2003.
- [26] MacSween, P., and G. Wainer. "On the Construction of Complex Models Using Reusable Components." In *Proceedings of the 2004 Spring Simulation Interoperability Workshop*, Arlington, VA, 2004.
- [27] Reynolds, C. W. "Steering Behaviors for Autonomous Characters." <<http://www.red3d.com/cwr/papers/1999/gdc99steer.html>> (Checked February 2006).
- [28] Madhoun, R., and G. Wainer. "Modeling a Battlefield Using Cell-DEVS." In *Proceedings of SISO Spring SIW*, San Diego, CA, 2005.
- [29] López, A., and G. Wainer. "Improved Cell-DEVS Model Definition in CD++." *LNCS 3305* Edited by P. M. A. Sloot, B. Chopard, and A. G. Hoekstra. ACRI 2004, Springer-Verlag, 2004.

Acknowledgements

This work has been partially funded by NSERC, the Canadian Foundation for Innovation, the Ontario Innovation Fund, and Precarn. Rami Youssef and Peter MacSween collaborated in developing some of the models presented in the paper.

Author Biographies

Gabriel Wainer received M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) from the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada), where he is now an Associate Professor. Previously, he was Assistant Professor at the Computer Sciences Department 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, another on discrete-event simulation, and over 110 research articles. He was PI of several research projects (NSERC, Precarn IRIS, IBM Scholars, Usenix, CFI, CONICET, ANPCYT). Dr. Wainer is an Associate Editor of *Simulation: Transactions of the SCS*, and the *International Journal of Simulation and Process Modeling*. He is a member of the Board of Directors of the SCS, a chair of the DEVS standardization study group (SISO), Associate Director of the Ottawa Center of The McLeod Institute of Simulation Sciences, and chair of the Ottawa M&SNet. His current research interests are related to modeling methodologies and tools, parallel/distributed simulation, and real-time systems. Dr. Wainer's web address is <http://www.sce.carleton.ca/faculty/wainer>.

Rami Madhoun received a Bachelor's degree in Electrical and Computer Engineering from the University of Qatar, 2000. He worked at Qatar Telecom in positions related to software development and network/system administration. He also worked at Convergys (Canada) as a technical support engineer before joining the Department of Systems and Computer Engineering at Carleton University, Canada, as an M.A.Sc. student. Mr. Madhoun holds scholarships including the Ontario Graduate Scholarship (OGS), Ontario Graduate Scholarship for Science and Technology (OGSST), HPCVL-Sun, and Precarn. He is second year master's student working in the area of parallel/distributed simulation.