Advanced Visualization of DEVS and Cell-DEVS Models in CD++/Maya

Ayesha Khan
Gabriel A. Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, K1S 5B6, Canada
amkhan2@connect.carleton.ca, gwainer@sce.carleton.ca

ABSTRACT: *CD++ is a modeling and simulation tool that was created to study complex systems by using a discrete-event cell-based approach. It was successfully employed to define a variety of models for complex applications using a cell-based approach. In order to improve model validation and analysis, we introduced a 3D visualization engine, which is based on the Maya 3D visualization tool and its scripting language. The application allows virtual worlds to be developed using the Maya visualization environment, and permits interaction with DEVS models built in CD++. The result is an enhanced simulation environment, which permits improved experimentation. We discuss how these two applications interact, and how models defined earlier in CD++ can interoperate with advanced visualizations built based on Maya 3D models.*

1. Introduction

At present, a large number of modeling and simulation techniques and tools have been developed to deal with complex systems. A technique that is gaining popularity in recent years is called **Discrete Event Systems Specification (DEVS)** [1], a framework for the construction of discrete-event hierarchical modular models, allowing for model reusing. In DEVS, basic models (**atomic**) are specified as black boxes, and they can be integrated together forming a hierarchical structural model (**coupled**). Cell-DEVS [2] extended the DEVS formalism allowing the simulation of discrete-event cellular models. The approach extends traditional Cellular Automata (CA) [3] defining each cell in a cell space as a DEVS atomic model and the space as a DEVS coupled model, including a flexible way of defining the timing of each cell.

We developed an environment, called CD++ [4], which implements DEVS and Cell-DEVS theories. CD++ enabled us to solve successfully a variety of complex problems [5, 6, 7]. CD++ also provides remote access to a high performance DEVS simulation server. The end user tools were organized as a simulation client applied to the CD++ simulator. Using these facilities, the users can now develop and test their models in local workstations, and submit them to be simulated in a remote CD++ server executing in a high performance platform. Then, they can receive, visualize and analyze the results on a local computer, improving model definition and execution.

Visualization tools are crucial in helping to understand better the behavior of these systems. CD++ was recently provided with facilities for 2D and 3D visualization using VRML and Java [8]. This 3D GUI enables sophisticated visualization of Cell-DEVS models only, and DEVS models can only be visualized in 3D; thus we have focused on new extensions that can be applied to both DEVS and Cell-DEVS. The interface here presented is based on the Maya modeling environment [9]. We will show how advanced DEVS models can be visualized using Maya facilities, giving a few examples of application, which permit discussing interoperability of a M&S tool based on DEVS and an advanced generic visualization environment like Maya.

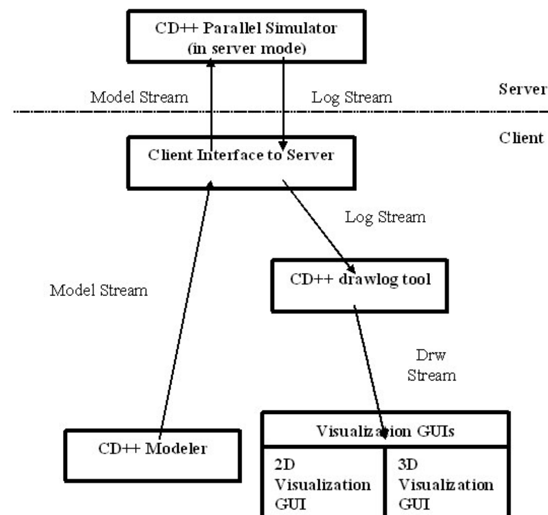


Figure 1: CD++ server architecture

2. DEVS, Cell-DEVS and CD++

A real system modeled with DEVS is described as a composite of sub-models, each of them being behavioral

atomic) or structural (coupled). A DEVS atomic model can be informally described as in Figure 2.

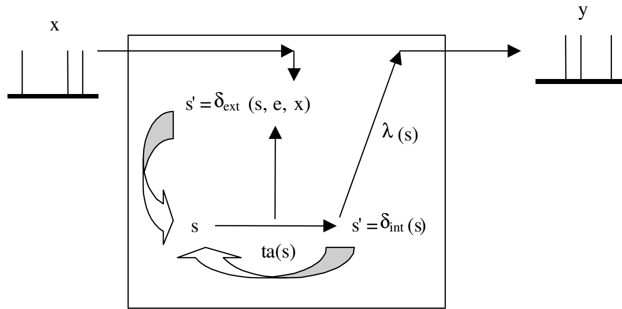


Figure 2: Informal Description of an Atomic Model

Each atomic model can be seen as having an interface consisting of *input* (x) and *output* (y) *ports* to communicate with other models. Every *state* (**s**) in 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* ($\boldsymbol{\delta}_{\mathbf{int}}$) is fired, producing a local state change. Input external events (those events received from other models) are collected in the input ports. An external transition function ($\boldsymbol{\delta}_{\mathbf{ext}}$) specifies how to react to those inputs, using the current state (s), the elapsed time since the last event (e) and the input value (x).

A DEVS coupled model is composed of several atomic or coupled sub-models, as in Figure 3.

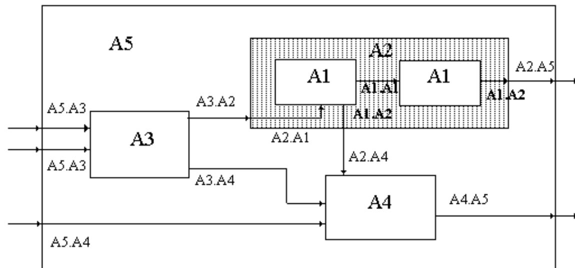


Figure 3: Informal Description of a Coupled Model

Coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the models' interfaces. The models' coupling defines how to convert the outputs of a model into inputs for the others, and how to handle inputs/outputs from/to external models.

Cell-DEVS has extended the DEVS formalism, allowing the implementation of cellular models with timing delays. A cellular model is a lattice of cells holding state variables and a computing apparatus, which is in charge of updating the cell state according to a local rule. This is done using the present cell state and those of a finite set of nearby cells (called its neighborhood). Each cell is defined as a DEVS atomic model, and it can be later integrated to a coupled model representing the cell space. Each cell uses N inputs 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.

Once the cell behavior is defined, a coupled Cell-DEVS can be created by putting together a number of cells interconnected with their neighbors. A Cell-DEVS model is informally presented in Figure 4.

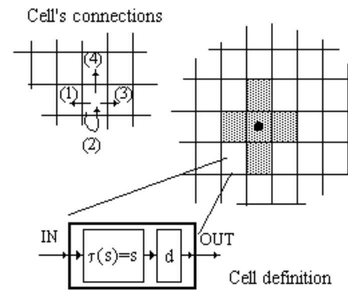


Figure 4: Description of a Cell-DEVS coupled model.

CD++ [4] is an M&S toolkit that implements DEVS and Cell-DEVS theory. Atomic models can be defined using a state-based approach (coded in C++ or an interpreted graphical notation), while coupled and Cell-DEVS models are defined using a built-in specification language. We will show the basic features of the tool through an example of application: a model of a car factory, which tries to coordinate different warehouses and assembly lines to make sure their productivity levels are suitable. The factory only manufactures one type of car and each sub-factory manufactures only one type of auto part. Each sub-factory sends its completed component to the Final Assembly Sub-factory where the automobile is assembled [10].

The model is defined as a DEVS coupled model, using all of the different components in the factory: four sub-factories devoted to manufacture different parts of a car (Chassis, Body, Transmission Case and Engine), and a warehouse devoted to the final Assembly. To make an automobile, only one of each component is needed (i.e. 1 Chassis + 1 Body + 1 Transmission Case + 1 Engine). To make an Engine we need four Pistons and one Engine Body (i.e. 4 Piston + 1 Engine Body = 1 Engine) [10]. The structure of this model is depicted in Figure 5. In order to build this application, we need to define and develop each of the atomic models depicted in Figure 5.

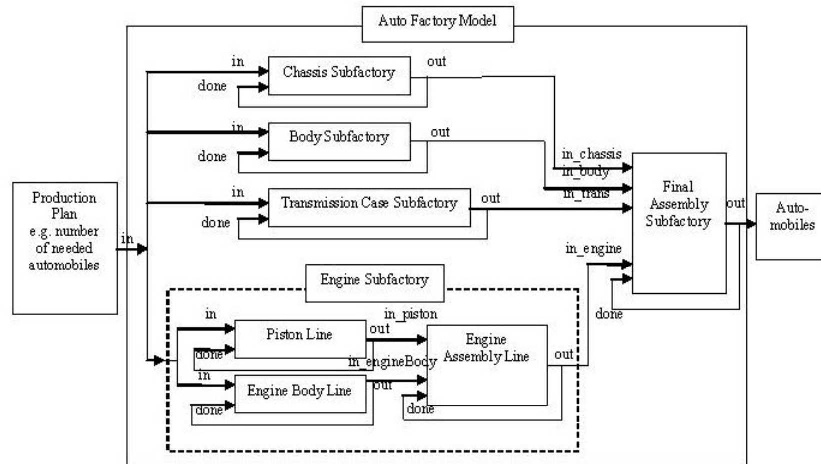


Figure 5: The auto factory layout [10].

```
Model EngineAssem::EngineAssem(const string
&name):Atomic(name), in_piston(addInputPort(
"in_piston") ), in_engineBody(addInputPort(
"in_engineBody") ), done(addInputPort("done") ),
out( addOutputPort("out")), manufacturingTime(
0, 0, 10, 0 ) { } // Model constructor

Model &EngineAssem::externalFunction( const
ExternalMessage &msg ) {
        if( msg.port() == in_piston ) {
        // parts received one by one
        elements_piston.push_back( 1 ) ;
        if( elements_piston.size() == 1 &&
           elements_engineBody.size()>=1)
              holdIn(active, manufacturingTime );
        //pushback if more than 1 received
        for(int i=2;i<=msg.value;i++)
             elements_piston.push_back( 1 ) ;
        }

        if( msg.port() == in_engineBody )     {
        elements_engineBody.push_back( 1 ) ;
        if( elements_engineBody.size() == 1 &&
            elements_piston.size()>=1)
        holdIn(active, manufacturingTime );
        //pushback if more than 1 received
        for(int i=2;i<=msg.value;i++)
             elements_engineBody.push_back( 1 ) ;
        }

        if( msg.port() == done ) {
          elements_piston.pop_front() ;
        elements_engineBody.pop_front() ;
        if(!elements_piston.empty() &&
              !elements_engineBody.empty())
        holdIn(active, manufacturingTime );
        }
}

Model &EngineAssem::internalFunction( const
InternalMessage & ) {  passivate(); }

Model &EngineAssem::outputFunction( const
InternalMessage &msg ) {
 sendOutput( msg.time(), out, elements.front());
}
```

Figure 6: Engine Assembly Line in CD++

As showed in Figure 6, we have defined each component as a DEVS atomic model, and implemented it using CD++. The model in Figure 6 represents the behavior of the Engine Assembly warehouse model. We start by defining *EngineAssem* as a subclass of the Atomic model class, and we also include the definition of the I/O ports needed by the model (*in_piston* and *in_engineBody*, which are used to receive the required parts). The *out* port is an output port used for assembled engines, while *done* is a feedback port we use when an engine is ready (in that way, we can check if there are enough stock of components, and we can start building a new engine as soon as one of them leaves the warehouse).

Most of the logic of the model is located in the external transition (δ_{ext}). This function determines what to do with the incoming parts. If a piston is received, is stocked until the number of pistons needed is available. The next internal event (δ_{int}) is scheduled by the *holdIn* method, which implements the time advance function (*ta*). When the time indicated by the variable *manufacturingTime* expires, the output function (λ) generates a ready part. The internal transition function simply passivates the model (i.e., sets the next internal transition time to infinity), waiting for the next part to come from other parts of the factory.

Once every atomic model in the hierarchy is defined (as in Figure 6), we can build a coupled model following the model architecture presented in Figure 5. Figure 7 presents the definition of such a model in CD++. The top model here is composed of one coupled models (*engineSubFact*) and four atomic components (*chassis*, an instance of the Chassis atomic model; *body* an instance of the Body atomic model; *trans,* an instance of the Trans[mission] model, and *finalAssem[bly]*). The *engineSubFact* contains three components, as showed in Figure 5. The input and output ports define the model's

interface, and the links between components define the model's coupling, following the structural description in Figure 5.

```
[top]
components : chassis@Chassis body@Body
trans@Trans finalAssem@FinalAssem engineSubFact
out : out
in : in
Link : in in@chassis
Link : in in@body
Link : in in@trans
Link : in in@engineSubFact
Link : out@finalAssem out
Link : out@finalAssem done@finalAssem
Link : out@chassis in_chassis@finalAssem
Link : out@chassis done@chassis
Link : out@body in_body@finalAssem
Link : out@body done@body
Link : out@trans in_trans@finalAssem
Link : out@trans done@trans
Link : out@engineSubFact in_engine@finalAssem

[engineSubFact]
components : piston@Piston engineBody@EngineBody
engineAssem@EngineAssem
out : out
in : in
Link : in in@piston
Link : in in@engineBody
Link : out@piston in_piston@engineAssem
Link : out@piston done@piston
Link : out@engineBody in_engineBody@engineAssem
Link : out@engineBody done@engineBody
Link : out@engineAssem out
Link : out@engineAssem done@engineAssem
```

Figure 7: Specification of a Coupled Model in CD++

Once this model is completely defined, we can execute it within a given experimental framework, and analyze the simulation results, which are provided in a log file with the format shown in Figure 8.

```
X/00:000/top/in/2 to chassis
X/00:000/top/in/2 to body
X/00:000/top/in/2 to trans
X/00:000/top/in/2 to enginesubfact
D/00:000/chassis/02:000 to top
D/00:000/body/02:000 to top
D/00:000/trans/02:000 to top
X/00:000/enginesubfact/ in/2 to piston
X/00:000/enginesubfact/ in/2 to enginebody  ...
Y/02:000/chassis/out/1 to top
D/02:000/chassis/... to top
X/02:000/top/done/1 to chassis
X/02:000/top/in_chassis/1 to finalass ...
*/02:000/top to enginesubfact
*/02:000/enginesubfact to enginebody
Y/02:000/enginebody/out/1 to enginesubfact
D/02:000/enginebody/... to enginesubfact
X/02:000/enginesubfact/done/1 to enginebody
X/02:000/in_enginebody/1 to engineassem
D/02:000/enginebody/02:000 to enginesubfact
D/02:000/engineassem/02:000 to enginesubfact ...
```

Figure 8: Excerpt from the auto factory log file.

In this figure, we can see that at 01:000, a purchase order arrives, and it is transmitted to the factory (X-messages). According to the manufacturing time, we receive outputs (Y-messages) from each of the components. By analyzing the log file, we can see the activation of the different parts in the plant, and each of the elements involved in the manufacturing simulation.

CD++ also includes an interpreter for Cell-DEVS models. The model specification includes the definition of the size and dimension of the cell space, the shape of the neighborhood and borders, as presented in figure 3. The cell's local computing function is defined using a set of rules with the form: POSTCONDITION DELAY {PRECONDITION}. These indicate that when the *PRECONDITION* is satisfied, the state of the cell will change to the designated *POSTCONDITION*, whose computed value will be transmitted to other components after consuming 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. Figure 9 shows the definition of a very simple example.

```
[life]
size: (20,20) delay : transport border : wrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0)
(0,1) (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 9: Definition of the Life game.

The rules in this example say that a cell remains active when the number of active neighbors is 3 or 4 (*truecount* indicates the number of active neighbors) using a transport delay of 10 ms . If the cell is inactive (`(0,0) = 0`) and the neighborhood has 3 active cells, the cell activated (represented by a value of 1 in the cell). In every other case, the cell remains inactive (*t* indicates that whenever the rule is evaluated, a *True* value is returned).

In [11], we presented a definition of maze-solving algorithms using Cell-DEVS and their implementation using CD++. When these rules are processed, the simulation results show the algorithm effectively blocking off every dead-end path in the maze. Every free cell that is accessible from only one direction must be a dead end, therefore cannot be part of the solution, and is therefore turned into a wall cell. The simulation repeats this procedure until the system is stable and all the cells are wall cells except for the cells that form the solution to the maze. In the case where there is no solution to the maze, all the cells become wall cells [11].

Figure 10 shows an excerpt from a draw file generated from CD++ log files, showing a simple maze in a 20x20 cellular array solving itself. As we can see, studying the simulation results based on these notations can be error-prone and cumbersome, more so for a specialist without much experience in computer programming. Instead, the provision of a graphical environment can improve the results obtained, as discussed in the following sections.

```
Line : 822 - Time: 00:00:00:000
    01234567890123456789
   +--------------------+
  0|11111111111111111111|
  1|     1 11 11      11|
  2|1111 1 11 1111111 11|
  3|1  1 1         11 11||
  4|1 11 1 11 1111 11 11|
  5|1 11 1 11 1111 11 11|
  6|1  1 1  1 1       11|
  7|1 11 1111 1 1111 111|
  8|1 11    1 1111 1 111|
  9|1 11 11 1 1 11 1  11|
 10|1 11 11 1 1 11 11 11|
 11|1    11   1 11 11 11|
 12|11 1 111111 11 11 11|
 13|11 1        11 11111|
 14|11 11111111111 1   1|
 15|11 11      111 1 1 1|
 16|11 11 1 11     1 1 1|
 17|11 11 1111111111 1 1|
 18|1   1            1 1|
 19|1111111111 111111111|
   +--------------------+
```

```
Line : 1001 - Time: 00:00:00:100
    01234567890123456789
   +--------------------+
  0|11111111111111111111|
  1|     11111111     11|
  2|1111 1 11 1111111 11|
  3|1 11 1         11 11|
  4|1 11 1 11 1111 11 11|
  5|1 11 1 11 1111 11 11|
  6|1 11 1 11 1       11|
  7|1 11 1111 111111 111|
  8|1 11    1 111111 111|
  9|1 11 11 1 1111 1  11|
 10|1 11 11 1 1 11 11 11|
 11|1    11   1 11 11 11|
 12|11 1 111111 11 11111|
 13|11 1        11 11111|
 14|11 11111111111 1   1|
 15|11 11      111 1 1 1|
 16|11 11 1111     1 1 1|
 17|11 11 1111111111 1 1|
 18|11 11            111|
 19|1111111111 111111111|
   +--------------------+
```

```
Line : 1146 - Time: 00:00:00:200
    01234567890123456789
   +--------------------+
  0|11111111111111111111|
  1|     111111111    11|
  2|1111 111111111111 11|
  3|1111 1         11 11|
  4|1 11 1 11 1111 11 11|
  5|1 11 1 11 1111 11 11|
  6|1 11 1111 11      11|
  7|1 11 1111 111111 111|
  8|1 11    1 111111 111|
  9|1 11 11 1 111111  11|
 10|1 11 11 1 1111 11 11|
 11|1    11   1 11 11111|
 12|11 1 111111 11 11111|
 13|11 1        11 11111|
 14|11 11111111111 1   1|
 15|11 11      111 1 1 1|
 16|11 11 1111     1 1 1|
 17|11 11 1111111111 111|
 18|11111            111|
 19|1111111111 111111111|
   +--------------------+
```

Figure 10: Three instances from the maze model draw file.

3. Visualization of 3D Models in MAYA

Maya [9] is a powerful application for three dimensional modeling and animation, using special effects and rendering. It allows one to create digital imagery, three dimensional animation and visual effects. The Maya software interface is fully customizable and it allows users to extend their functionality within Maya by providing access to the Maya Embedded Language (MEL). Using MEL, programmers can tailor the user interface to their needs and to add in-house tools. Since MEL is recognized by embedded web browsers, MEL commands can also be issued form a webpage. Maya's modeling and animation tools were used to create three-dimensional environments for Cell-DEVS and DEVS models. To do that, the user must use Maya facilities to create visual scene files, while an application written in MEL permits to create a user interface that allows CD++ log files to interact with Maya, and to visualize the corresponding model in a 3D visual environment. This instantiates a MEL script specific to a particular model, and animates the three-dimensional world (scene file) in accordance with the CD++ log file. Figure 11 shows the relationships between these procedures.

The *logFileAnimator* method acts as an interface requesting the user to select a particular model, as showed in Figure 12. The user has two choices after providing the required information, the "Print File Contents" button will instantiate *readFile* and the "Animate" button will instantiate *animator*. The *readFile* method locates and opens the file corresponding to the file name provided for the express purpose of reading it and printing the contents to the Script Editor Window in Maya, as showed in Figure 13. In this way, the user can analyze the detailed results found in the log files.

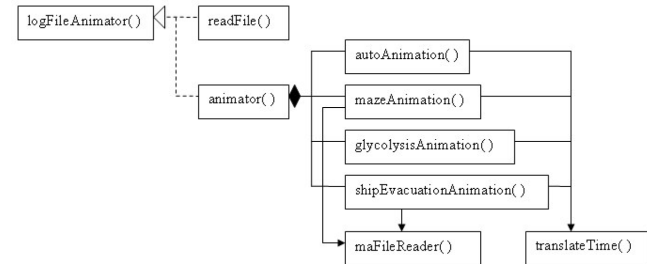


Figure 11: Architecture of the visualization environment

The *animator* method instantiates the animation procedure for that particular model, associating CD++ simulation results with graphic scenes defined in Maya. Each instance of the animation procedure opens the log File, reads it and stores pertinent information, which is then used to animate the objects in the three dimensional scene opened. All the information pertaining to a particular object from the log file is used to animate that same object in the scene file.

The *translateTime* method is in charge of accurately following the log File, and making the animation to match time with the time present in the simulation log.

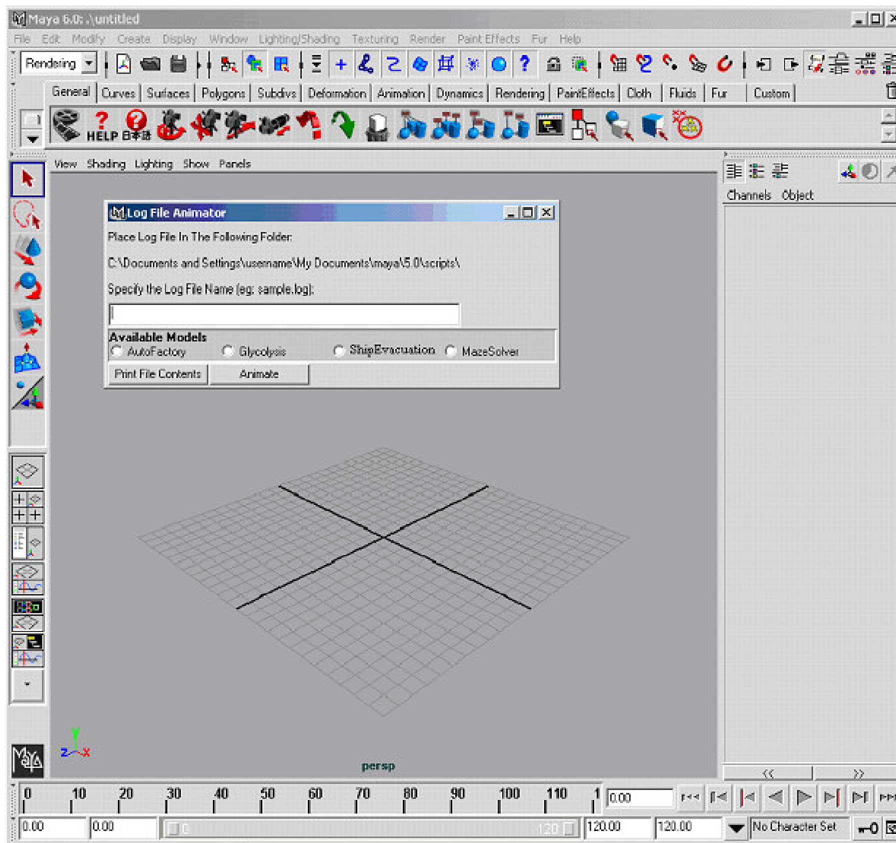


Figure 12: The *logFileAnimator* dialogue box.

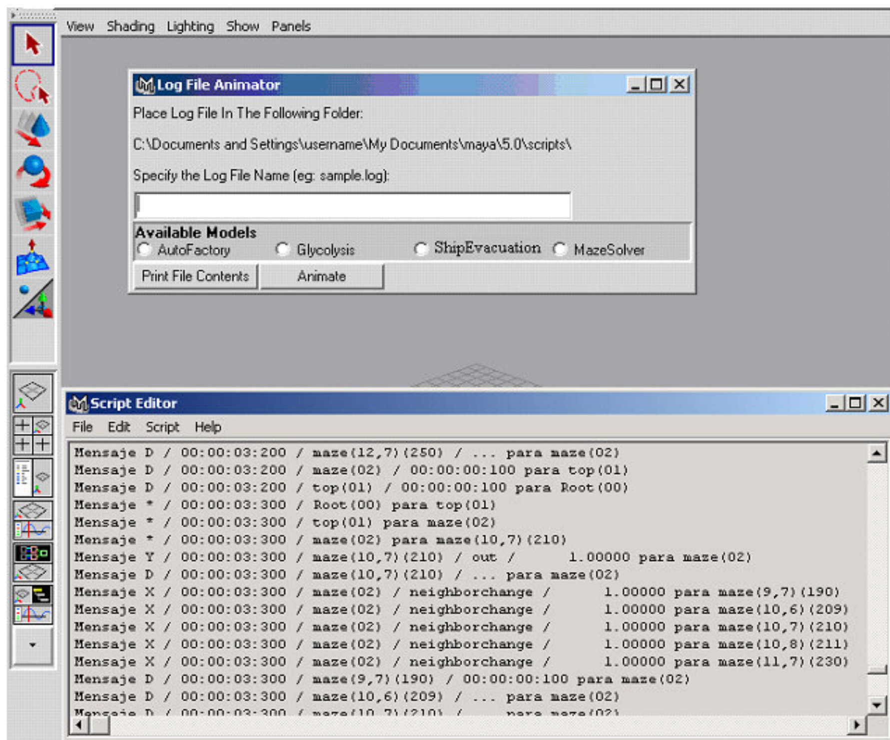


Figure 13: The Script Editor Window displays the contents of the log file.

The *maFileReader* method is called by MEL to obtain the initial state values of each cell for Cell-DEVS models. This procedure parses the coupled model files and stores the initial state values of each cell. Then it animates the scene file for time 00:00:00:000 accordingly.

We have applied the toolkit to different modeling examples, including the two presented in the previous section. Figure 14 shows the visual results of the Maze model, when we apply our new visualization environment. As we can see, the visual results impact the understanding of the maze-solving technique when compared with the previous results.

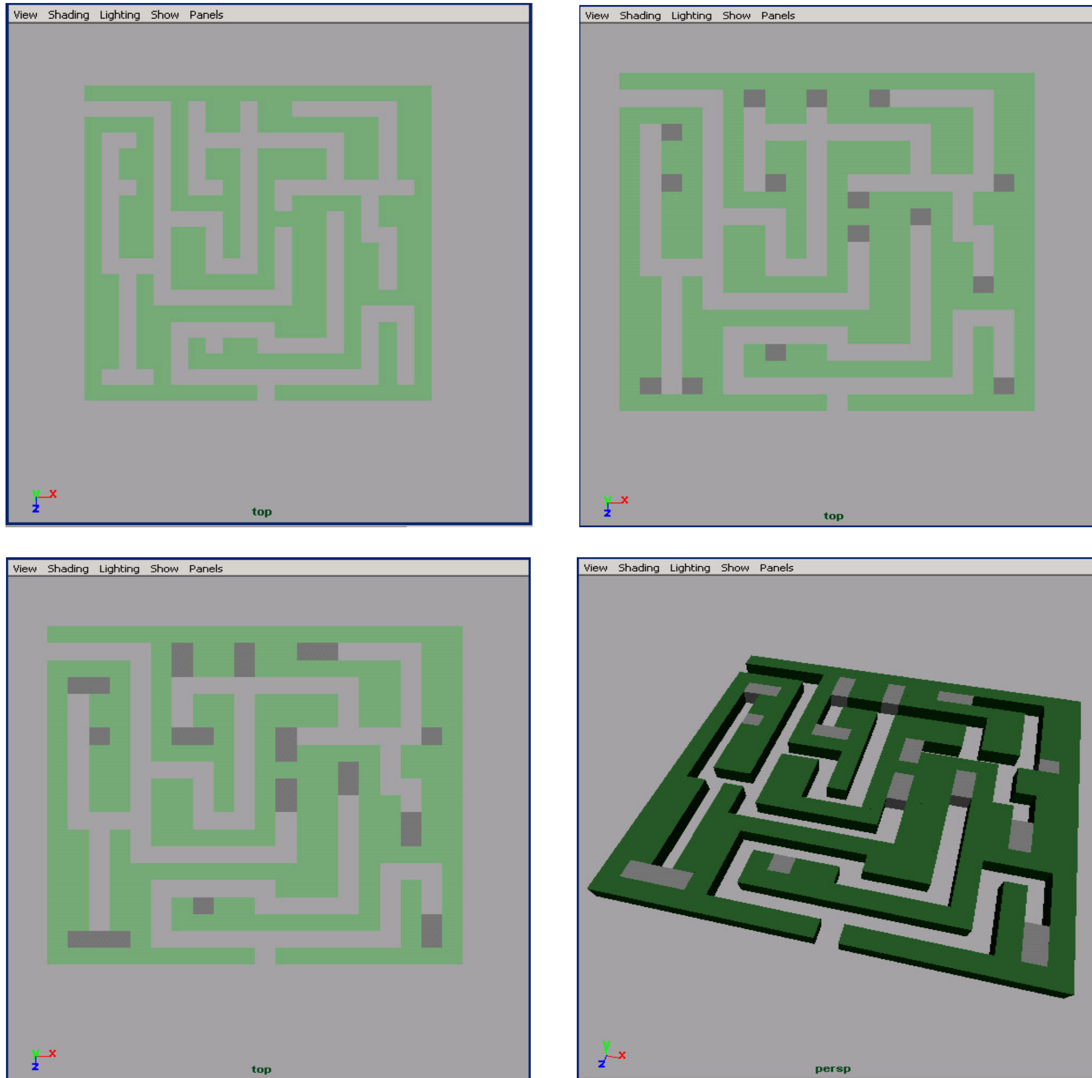
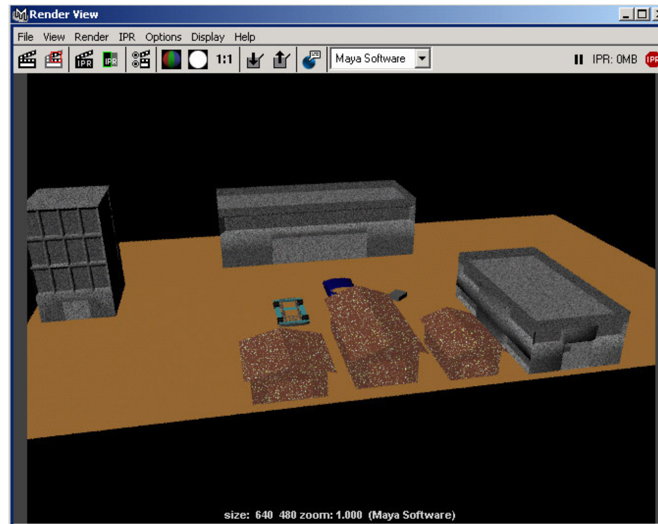
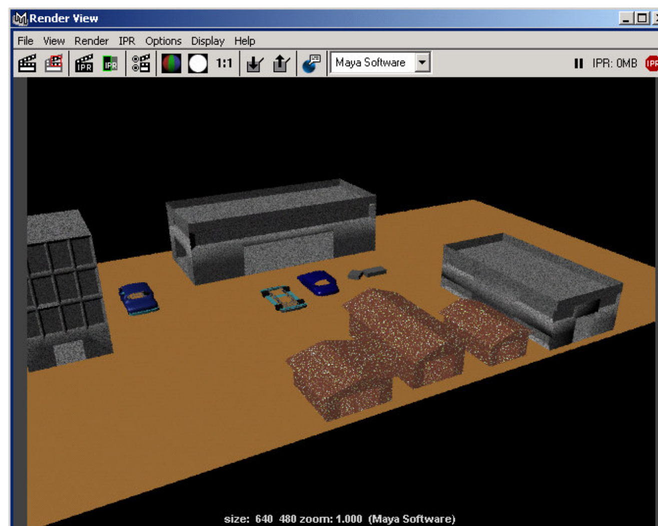


Figure 14: Perspective View of the maze model.

Likewise, Figure 15 shows an animation snapshot obtained when executing a 3D version of the Factory model Figure 15 a) shows the visual results of the model at time 02:000, in which the log file presented in Figure 8 shows that the three sub-factories have generated outputs (chassis, body and transmission). Reviewing Figure 8, we can see Y-messages (representing outputs) from the chassis and body models. These are represented as parts leaving the warehouses, and being directed to the assembly factory. At time 02:000 all three outputs start moving towards the Final Assembly sub-factory. Figure 15 b) shows the result at 04:000 (when three more parts are ready to be assembled, and the parts previously arrived at 02:000.



(a)

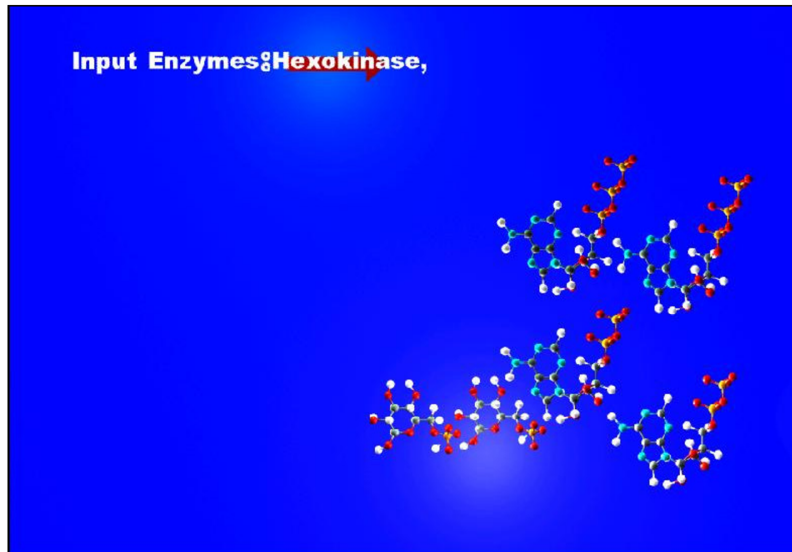


(b)

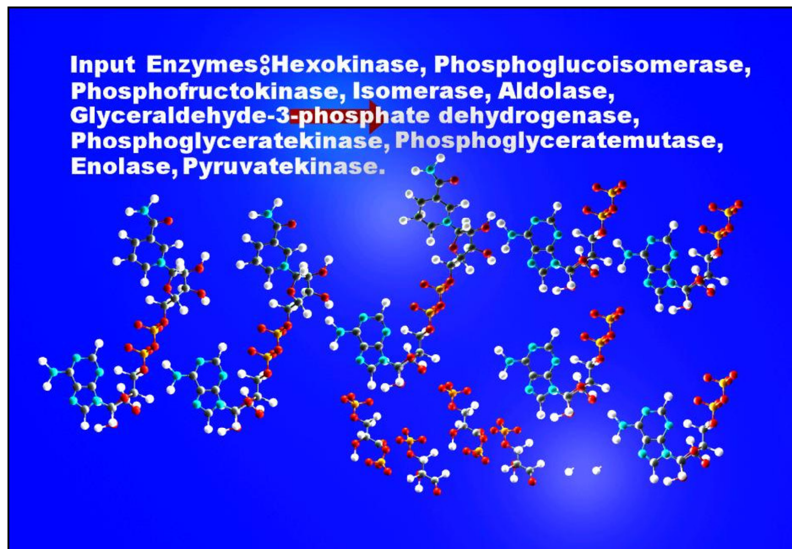
Figure 15: Rendered View of the auto factory animation (a) 02:000 (b) 04:000.

We also employed these facilities for the visualization of models of virtual cells. In [12], we developed a model of glycolysis, the sequence of reactions occurring in the cells that permit to break down one glucose molecule into two molecules of pyruvate. There are ten steps in glycolysis that result in the production of Nicotinamide adenine dinucleotide (NADH) and Adenosine TriPhosphate (ATP) [13]. Each step in the sequence is controlled by a specific enzyme. The glycolysis sequence can be divided into two phases, where in the first phase glucose is converted into two Glyceraldehyde-3-Phosphate molecules (GDP), and in the second phase two pyruvate molecules.

Each step in the glycolysis pathway was defined as a DEVS atomic model specification, which was used to analyze basic properties of the models. Afterwards, each model was implemented in CD++, and tested separately. Once every model was thoroughly tested, a coupled model was built, connecting all the sub-models previously defined, each representing a step. Different simulation experiments were conducted [12]. Figure 16 shows the visual results of the execution of two of the steps. Figure 16 a) shows the end of Glycolysis step 1, where two Alpha-Gluco-Phospate (G-6-P) and two Adenosine DiPhosphate (ADP) are formed. Figure 16 b) shows step 6, which begins at the appearance of 3 molecules of Nicotinamide Adenine Dinucleotide (NAD+).



(a)



(b)

Figure 16: Glycolisis model (a) step 1 (b) step 6.

A related model was focused on the Krebs Cycle, a sequence of enzyme-catalyzed reactions in the cell. Glycolysis and Krebs cycle are two major stages in the process of the metabolism of glucose. Glycolysis is the first stage and breaks down glucose to Pyruvate, where as the Krebs Cycle is the second stage. Each turn of this cycle produces two molecules of carbon dioxide and eight atoms of hydrogen [12].

In this case, we also defined the behavior of each stage as a DEVS atomic model specification, reproducing the behavior of inputs and outputs observed for each step. Figure 17 shows a snapshot one of the reactions in the Krebs Cycle Animation done in CD++/Maya: the formation of Acetyl CoA, and the production of Carbon dioxide and NADH as byproducts.

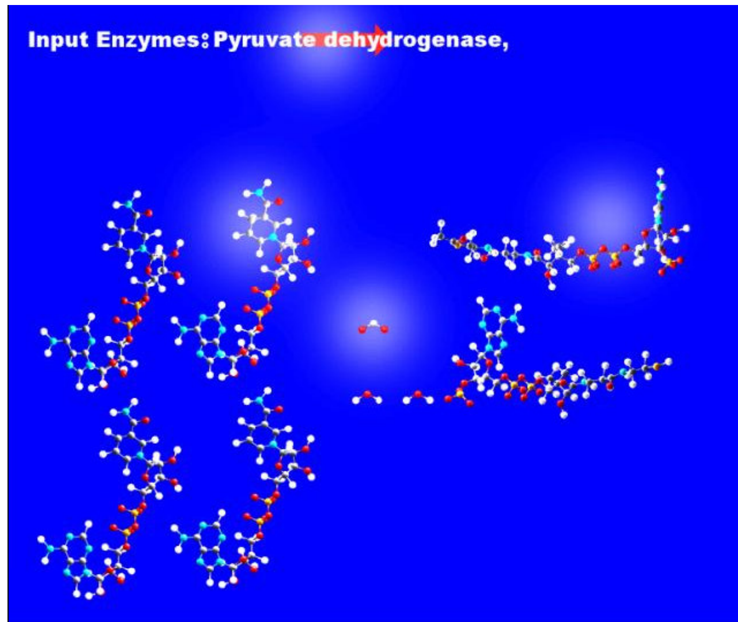


Figure 17: The Krebs Cycle: Acetyl CoA is formed.

We also defined an advanced visualization model of evacuation. These models that could predict and present the results of human beings evacuating structures, such as buildings, ships and houses etc, during an emergency [14]. This model is based on a cellular automata model for ship evacuation [15]. We will show how the results of our visualization environment facilitate and ease the interpretation of the simulation results.

In this model, the rules calculate the shortest distance of each cell to the nearest exit and assign people randomly to the cells. The basic idea was to simulate the behavior and movement of every single person involved in the evacuation process. We used two planes: one for the floor plan of the structure and the people moving, and the other for orientation to an exit. Each cell in the grid represents 0.4 m^2 (one person per cell). The orientation layer contains information that serves to guide persons towards emergency exits. We assigned a potential distance to an exit to every cell of this layer. The persons will move for the room trying to minimize the potential of the cell in which they are. The Cell-DEVS model characterizes a person's behavior: a normal person goes to the closest exit; a person in panic goes in opposite direction to the exit. People move at different speeds; if the way is blocked, people can decide to move away and look for another way.

In figure 18 the state value “1” represents walls or obstacles, and the state value “2” represents exits. The even state values are occupied cells and the odd ones are empty cells. Each state value also represents the shortest direction to the exit. Eventually all cells become empty as people leave the structure.

```
Line : 4926 - Time: 00:00:00:000
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
   +-----------------------------------------------------------+
  0|  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1|
  1|  1  5  5  5  5  5  4  4  4  4  3  3  4 10 10  9 10 10  3  1|
  2|  1  5  5  7  1  1  5  5  5  6  5  6  3  1  1  1  1  1  3  1|
  3|  1  1  1  1  1  1  1  1  1  1  1  1  4  1  1  1  1  1  4  1|
  4|  1  4  1  3  4  3  4  3  4  3  3  1  3  4  3  3  4  3  4  1|
  5|  1  3  9  9 10 10  9  5  6  6  6  6  4  9  9 10  9 10  4  1|
  6|  1  4  1  1  1  1  1  1  1  1  1  1  3  1  1  1  1  1  3  1|
  7|  1  3 10  9 10  9 10  6  6  5  6  6  4  9 10  9 10  9  4  1|
  8|  1  4  1  1  1  1  1  1  1  1  1  1  4  1  1  1  1  1  4  1|
  9|  1  2 10  9  9  9 10  6  5  5  5  6  2 10  9 10 10 10  9  1|
 10|  1  8  1  1  1  1  1  1  8  1  1  1  1  1  1  1  1  1  8  1|
 11|  1  7 10  9 10  9  9  5  8 10 10 10  9  9  9  9  9  1  7  1|
 12|  1  7  1  7  1  1  1  1  7  8  8  8  7  8  7  8  4  1  7  1|
 13|  1  1  1  8  1  1  1  1  1  1  1  1  7  7  7  3  4  1  7  1|
 14|  1  6  6  7 10  9  1  1  1  1  1  1  7  8  6  6  5  5  8  1|
 15|  1  5  6  7  8  7 10 10  9 10 10  5  8  1  1  1  1  1  8  1|
 16|  1  5  6  8  7  8  7  8  8  8  8  5  8 10  9  9 10  1  7  1|
 17|  1  7  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  8  1|
 18|  1  7 10  9  9 10  9 10  9 10  9  6  6  5  5  6  5  6  7  1|
 19|  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1|
   +-----------------------------------------------------------+
```

Figure 18: An Excerpt from a Ship Evacuation draw file.

As seen in figure 18 this visualization of the simulation results is complex to interpret and understand. When these results are integrated into the new visualization engine, the results become easier to observe. Figure 19 illustrates the results obtained through Maya. Compare figure 18 with Figure 19 a), which shows the same state than the one presented in Figure 18, and the beginning of the animation at time 00:00:00:000. Figure 19 b) shows people moving towards the exits and evacuating the building at time 00:00:03:000.Then in figure 19 c) we observe the animation at time 00:00:03:200, and finally in figure 19 d) at time 00:00:06:000 the building is empty.

(a) (c)

(b) (d)

Figure 19: The Ship Evacuation Model at time (a) 00:00:00:000 (b) 00:00:03:000 (c)00:00:03:200 and (d) 00:00:06:00

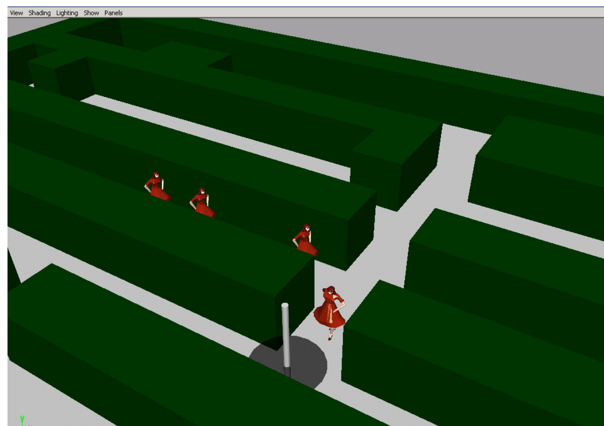


Figure 20: A close up at time 00:00:05:240

In Figure 20 we show a close up view of the ship evacuation animation at time 00:00:05:240 when only four people are left in the building.

4. Conclusion

Simulation is becoming increasingly important in the analysis and design of complex systems. CD++ is a tool for the simulation of complex physical systems that can be used to simulate a variety of models. To facilitate the users to use the CD++ simulator, we extended its design to provide a number of services. The 3D visualization GUI enables sophisticated visualization of DEVS and Cell-DEVS models. To better understand the results, the user can select shapes to represent a node in the 3D space, select different colors, shapes, edit scenes, etc. The current facilities have highly improved the use of the previously existing tools, thus enhancing the analysis experience of the modelers using the toolkit.

The approach relies on the use of DEVS methodology and it is supported by the use of CD++, a DEVS tool that has been built following the formal definitions of DEVS models. The use of DEVS enables proving the correctness of the simulation engines and permits to model the problem even by a non-computer science specialist. The high level language of CD++ reduces the algorithmic complexity for the modeler while allowing complex cellular timing behaviors. DEVS allows independence of the simulator, the models developed, the experiment conducted and the visual engine, while maintaining unity in the model specification and tool interoperation.

Acknowledgments

This work has been partially supported by NSERC (National Science and Engineering Research Council of Canada), and it was developed within Carleton University Immersive Media Lab (http://www.cims.carleton.ca), directed by Prof. Michael Jemtrud, who provided support for this work.

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: "Timed Cell-DEVS: modeling and simulation of cell spaces " In "Discrete Event Modeling & Simulation: Enabling Future Technologies" Springer-Verlag, 2001.

[3] T. Toffoli: "Occam, Turing, von Neumann, Jaynes: How much can you get for how little? (A conceptual introduction to cellular automata)" Proceedings of ACRI'94, 1994.

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

[5] J. Ameghino; A. Troccoli; G. Wainer: "Modeling and simulation of complex physical systems using Cell-DEVS" In Proceedings of 34th IEEE/SCS Annual Simulation Symposium,. Seattle, U.S.A, 2001.

[6] G. Wainer: "Modeling and simulation of complex systems with Cell-DEVS" Accepted for publication in Proceedings of the Winter Simulation Conference, Washington, DC. IEEE Press, 2004.

[7] J. Ameghino; G. Wainer: "Application of the Cell-DEVS formalism for modeling cell spaces" In Proceedings of AIS'2004, Jeju Island, Korea, Lecture Notes in Computer Science, 2004.

[8] G. Wainer and W. Chen. "A framework for remote execution and visualization of Cell-DEVS models". *Simulation*. Vol. 79, pp. 626-647. November 2003.

[9] ALIAS Corp. "Maya 6 Features in Detail," [Online document], 2004, [cited 2004 Oct. 25], Available: http://www.alias.com/eng/products-services/maya/file/maya6_features _in_detail.pdf

[10] W. Sun: "A model of a car manufacturing plant", Internal Report (available on-line: http://www.sce.carleton.ca/faculty/wainer/wbgraf), Department of Systems and Computer Engineering, Carleton University, 2001.

[11] K. Lam; G. Wainer: "Modeling of maze-solving problems using Cell-DEVS". K. Lam, G. Wainer, In Proceedings of the 2003 SCS Summer Computer Simulation Conference. Montreal, QC, Canada, 2003.

[12] R. Djafarzadeh; G. Wainer; T. Mussivand: "Modeling and simulation of cellular metabolism and energy production by mitochondria", Accepted for publication. DEVS workshop; SpringSim. San Diego, CA. 2004.

[13] B. Alberts; D. Bray; J. Lewis; M. Raff; K. Roberts; J. Watson: "Molecular Biology of the cell", Third Edition, Garland Publishing, 1994.

[14] J. Ameghino, G. Wainer. "Using Cell-DEVS for modeling complex cell spaces". Proceedings of 13th International Conference on AI, Simulation, and Planning in High Autonomy Systems, AIS 2004, Jeju Island, Korea. LNCS Vol. 3397. 2004.

[15] J. R. Weimar: "Cellular automata model for ship evacuation", Internal Report (available online: http://www.jweimar.de/jcasim/schiff1.html), Institut für Informationssysteme, Technische Universität Braunschweig.