A framework for remote execution and visualization of Cell-DEVS models

Gabriel Wainer Wenhong Chen

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

gwainer@sce.carleton.ca

Abstract

Simulation is becoming increasingly important in the analysis and design of complex systems with natural and artificial components. CD++ is a modeling and simulation tool that was created to study this kind of 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. Here, we present different extensions done to the tool using a client/server architecture. Users can create models in local workstations, execute them in a remote high performance simulation engine, then receive, visualize and analyze the results locally with easy-to-use 2D and 3D interfaces. The 3D interface was built using VRML in order to facilitate web-based visualization. The tool now enables running several models simultaneously, and it supports multi-view outputs.

Keywords: DEVS, Cell-DEVS, distributed simulation, simulation visualization, virtual reality.

1. INTRODUCTION

Scientists and engineers have long relied in using models to better comprehend the systems they study. Models have been used for analysis, design, prediction and understanding of different complex phenomena. In most cases, these models were defined using mathematical representations, enabling mathematical analysis techniques. Unfortunately, these methods showed to be infeasible for studying several complex artificial systems developed in the second half of the 20th century (traffic controllers, digital systems, automated factories, robots, etc.). Likewise, the complexity of the natural systems under analysis grew, making impossible to use analytical methods.

Digital computers provided alternative methods of analysis. Since the early days of computing, the users started translating their analytical models into computer simulations, which enabled them to experiment with virtual environments. Computer simulation has enabled the analysis of natural and artificial systems with a level of detail unknown in earlier stages of scientific development. Simulated models are also well suited for training purposes, as they provide cost-effective risk-free environments.

At present, there are a large number of modeling and simulation techniques and tools developed to deal with complex

systems. A formalism that is gaining popularity in recent years is called **DEVS (Discrete Event Systems Specification)** [01,2]. DEVS provides a framework for the construction of discrete-event hierarchical models in a modular manner, allowing for model reusing to reduce development time. In DEVS, basic models (called **atomic**) are specified as black boxes, and several DEVS models can be integrated together forming a hierarchical structural model (called **coupled**).

Cell-DEVS [3] extended the DEVS formalism allowing simulating discrete-event cell spaces. The approach is based on Cellular Automata (CA), which are defined as a lattice of cells updated synchronously and simultaneously [4]. Each cell in a CA holds a state variable and a computing apparatus that defines how to obtain a new value based on the current state and the values of neighboring cells. Cell-DEVS extends these concepts by defining a cell as a DEVS atomic model and a cell space as a DEVS coupled model. It also introduces a new way of defining the timing of each cell, which is more flexible than previously existing approaches.

The CD++ tool [5] enables simulating DEVS and Cell-DEVS models, and it has been used to create a variety of models in different areas: biology (watersheds, fire spread, ant colonies), physics (crystal growth, lattice gases, heat diffusion), chemistry (solution diffusion in moving fluids), and several artificial systems (autonomous robots, heat seekers, urban traffic, etc.) [6, 7, 8, 9].

In several of these models, we found out that the computing power provided by standard workstations is not enough, moreover when running large cell spaces. Despite this fact, most end users nowadays have restricted access to high performance computing resources. Likewise, many of them prefer to use personal computers with standard software packages for development and analysis of the simulation results. A solution to these problems is to let the user to execute the simulations in high-performance remote computers, while using standard workstations for development and analysis. In these cases, *client/server* architectures provide an adequate framework to organize the model's distributed execution. The simulation software can be designed as a server able to execute many simulation models simultaneously, whereas the users can communicate with this server through a network to request simulation services. The design of CD++ was modified following these ideas, and it was transformed into a simulation server.

Another problem of using CD++ for complex systems analysis was the lack of adequate visualization mechanisms. Graphical tools are crucial to understand better the behavior of complex systems, and they facilitate thinking, problem solving, and decision-making. Scientific visualization tools create visual displays, in which numeric values in data sets are represented visually as colors, shapes, or symbols [1011]. The goal of visualization is to provide a deeper understanding of the physical systems being investigated, and to help in exploring the large set of numerical data produced in the simulation execution, which is a concern for model validation. In order to achieve these goals, CD++ was extended to enable 2D and 3D visualization. A 3D GUI (Graphical User Interface) was developed, providing useful functions for the users (select geometries for the nodes, assign different colors, edit individual nodes and navigate in the field of visualization). The GUI was designed with the intention of being used by various users from remote locations. The main interest of the users is to rapidly obtain results, and data visualizations would help them to assist in analysis for scientific and technical purposes. Their expertise will be diverse, and although they might be familiar with the technical domain of the models under development, we want to reduce the learning curve. Consequently, the visualization environment relies on standard interface conventions.

In summary, CD++ was transformed into a client/server engine that is able to provide visual simulation results and remote access to a high performance DEVS simulation server. The end user tools were organized as a simulation client applied to the CD++ simulation engine. The server is able to receive model specifications from the clients, and run them in parallel, sending back results to the local computers. In addition, many users can run simulations simultaneously. Using these facilities, the users can now develop and test their models in local workstations, and send them to be simulated in a remote CD++ server executing in a high performance platform. Then, they can receive, visualize and analyze the result on the local computer, improving model definition and execution.

The following sections will present the results of this effort. We first introduce basic aspects related to the modeling techniques we used. Then, we present a design for the modified tools and the visualization environment. Finally, we show several examples presenting the visualization facilities of the toolkit.

2. THE DEVS FORMALISM

The DEVS formalism [1] was originally defined in the '70s as a discrete-event modeling technique. A real system modeled with DEVS is defined of a composite of sub-models, each of them being a behavioral model (called **atomic**) or a structural model (called **coupled**). Each model is defined by a time base, state variables, inputs, outputs, and functions to determine the next states and outputs. Tested models can be integrated into a model hierarchy, improving model reuse, reducing testing time and enhancing productivity.

A DEVS atomic model is described as:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

$\boldsymbol{X}$ is the input events set;

S is the state set;

$\boldsymbol{Y}$ is the output events set;

$\boldsymbol{\delta_{int}}$: $S \rightarrow S$, is the internal transition function;

$\boldsymbol{\delta_{ext}}$: $Q \times X \rightarrow S$, is the external transition function; where $Q = \{ (s, e) / s \in S, \text{ and } e \in [0, D(s)]\}$;

$\boldsymbol{\lambda}$: $S \rightarrow Y$, is the output function; and

D: $S \rightarrow \boldsymbol{R}_0^+ \cup \infty$, is the duration function.

The model is seen as having an interface consisting of input (X) and output (Y) ports to interact with other models. Each state in a model has an associated lifetime, defined by the duration function. Once the lifetime of a given state is consumed, the internal transition function is activated to produce an internal state change. Before this state change, the model can generate outputs using the current state values through the output ports. To do so, the output function executes before activating the internal transition. At any moment, a model can receive input external events from other models through its input ports. When an external event arrives, the external transition function is activated. This function computes a new state for the model using the present state, the input values, and the time elapsed since the last event. Every time a transition function is activated, a new lifetime must be associated with the new state.

A DEVS coupled model is composed of several atomic or coupled sub-models, and it can be defined as:

$$CM = < X, Y, D, \{Mi\}, \{Ii\}, \{Zij\}, select >$$

X is the set of input events;

Y is the set of output events;

D $\in$ N, D $<\infty$ is an index for the components of the coupled model, and

$\forall$ i $\in$ D, $\mathbf{M_i}$ is a basic DEVS model, where

$$M_i = < X_i, S_i, Y_i, \delta_{int_i}, \delta_{ext_i}, D_i >$$

$\mathbf{I_i}$ is the set of influencees of model i, and $\forall$ j $\in$ I_i, and

$\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$ is the i to j translation function.

Finally, **select** is the tie-breaking selector.

Each coupled model consists of a set of basic models (atomic or coupled) connected through the input/output ports of the interfaces. Each component is identified by an index number. Each model is associated with a set of influencees, defined as those models to which output values must be sent. The translation function uses an index of influencees, created for each model (I_i). This function defines which outputs of model M_i will be converted into inputs for model M_j. When two submodels have simultaneous events, the *select* function defines which of them should be activated first.

The Cell-DEVS formalism extended this basic behavior to allow the implementation of cellular models with timing delays [3]. Each cell in these spaces includes a state variable, which is updated according to a local rule that considers the present cell state and those of a finite set of nearby cells (called the cell's **neighborhood**). The cells are defined as atomic models, and they can be specified as:

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

where for #T $< \infty \wedge$ T $\in$ $\{\boldsymbol{N}, \boldsymbol{Z}, \boldsymbol{R}, \boldsymbol{\{0,1\}}\} \cup \{\phi\}$;

$\boldsymbol{X} \subseteq$ T is the set of external input events;

$\boldsymbol{Y} \subseteq$ T is the set of external output events;

S $\subseteq$ T is the set of sequential states for the cell;

$\boldsymbol{\theta}$ is the definition of the cell's state, defined as

θ = { (s, phase, σqueue, σ) / s $\in$ S is the status value for the cell, phase $\in$ {active, passive},

σqueue = { $((v_1,\sigma_1),...,(v_m,\sigma_m))$ / m $\in \boldsymbol{N} \wedge$ m $<\infty) \wedge \forall$ (i $\in \boldsymbol{N}$, i $\in$ [1,m]), $v_i \in$ S $\wedge \sigma_i \in \boldsymbol{R_0}^+ \cup \infty$};

$\sigma \in \boldsymbol{R_0}^+ \cup \infty$ } for transport delays, or

θ = { (s, phase, f, σ) / s $\in$ S, phase $\in$ {active, passive}, f $\in$ T, and $\sigma \in \boldsymbol{R_0}^+ \cup \infty$ } for inertial delays;

I $\in S^{\eta+\mu}$ is the set of states for the input events;

d $\in \boldsymbol{R_0}^+$, d $< \infty$ is the transport delay for the cell;

$\boldsymbol{\delta_{int}}$: $\theta \rightarrow \theta$ is the internal transition function;

$\boldsymbol{\delta_{ext}}$: Qx$X \rightarrow \theta$ is the external transition function, where Q is the state values defined as:

$$Q = \{ (s, e) / s \in \theta \text{ x } I \text{ x } d; e \in [0, D(s)]\};$$

$\boldsymbol{\tau}$: $I \rightarrow S$ is the local computation function;

$\boldsymbol{\lambda}$: $S \rightarrow Y$ is the output function; and

D: θ x I x d $\rightarrow \boldsymbol{R}_0^+ \cup \infty$, is the state's duration function.

A cell uses the input values I to compute its next state, which is obtained by applying the local computation function τ. A delay function associated with each cell enables deferring the moment to transmit the computed result. There are two types of delays: *inertial* and *transport*. For the transport delay, the next value will be added to a queue sorted by output time, and the results will be stored during the delay. When this time is consumed, the value will be sent out. Inertial delays use a preemptive policy, that is, if the cell state changes before the delay, the previously computed result is not transmitted. This basic behavior is provided by the δ_{int}, δ_{ext}, λ, and D functions.

After the basic behavior of a cell is defined, a whole cell space is built by creating a coupled Cell-DEVS model that includes copies of each of the atomic cells. The Cell-DEVS coupled model can be defined as:

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

where for $\#T < \infty \ \wedge \ T \in \{\boldsymbol{N}, \boldsymbol{Z}, \boldsymbol{R}, \boldsymbol{\{0,1\}}\} \cup \{\phi\}$;

$\boldsymbol{X} \subseteq T$ is the set of external input events;

$\boldsymbol{Y} \subseteq T$ is the set of external output events;

Ylist = $\{ (k,l) / k \in [0,m], l \in [0,n]\}$ is the list of output coupling;

Xlist = $\{ (k,l) / k \in [0,m], l \in [0,n]\}$ is the list of input coupling; and

I = $< P^X, P^Y >$ represents the definition of the modular model interface. Here,

for $i = X \mid Y$, P^i is a port definition (input or output respectively), where

$$P^i = \{ (N(f,g)^i, T(f,g)^i) / \forall (f,g) \in Xlist, N(f,g)^i = i(f,g)_k \text{ (port name), and } T(f,g)^i \in T \text{ (port type)}\};$$

$\boldsymbol{\eta} \in \boldsymbol{N}$ is the neighborhood size and **N** is the neighborhood set, defined as

$$N = \{ (i_p,j_p) / \forall p \in \boldsymbol{N}, p \in [1,\eta] \Rightarrow i_p, j_p \in \boldsymbol{Z} \wedge i_p, j_p \in [-1, 1] \};$$

{m, n} $\in \boldsymbol{N}$ is the size of the cell space;

C defines the cell space, where $C = \{C_{ij} / i \in [1,m], j \in [1,n]\}$, with

$$C_{ij} = < X_{ij}, Y_{ij}, S_{ij}, N_{ij}, d_{ij}, \delta_{intij}, \delta_{extij}, \tau_{ij}, \lambda_{ij}, D_{ij} >$$

is a Cell-DEVS atomic model;

B is the set of border cells, where

- $B = \{\varnothing\}$ if the cell space is wrapped; or
- $B = \{C_{ij} / \forall (i = 1 \vee i = m \vee j = 1 \vee j = n) \wedge C_{ij} \in C\}$, where

$$C_{ij} = < X_{ij}, Y_{ij}, S_{ij}, I_{ij}, d_{ij}, \delta_{intij}, \delta_{extij}, \tau_{ij}, \lambda_{ij}, D_{ij} >$$

is a Cell-DEVS atomic model, if the border cells have different behavior than the rest of the cell space.

Z is the translation function, defined by:

Z: $P_{kl}{}^{Y_q} \rightarrow P_{ij}{}^{X_q}$, where $P_{kl}{}^{Y_q} \in I_{kl}$, $P_{ij}{}^{X_q} \in I_{ij}$, $q \in [0,\eta]$ and $\forall(f,g) \in N$, $k = (i+f) \bmod m$; $l=(j+g) \bmod n$;

$P_{ij}{}^{Y_q} \rightarrow P_{kl}{}^{X_q}$, where $P_{ij}{}^{Y_q} \in I_{ij}$, $P_{kl}{}^{X_q} \in I_{kl}$, $q \in [0,\eta]$ and $\forall(f,g) \in N$, $k = (i-f) \bmod m$; $l = (j-g) \bmod n$;

select is the tie-breaking selector function, with select $\subseteq mxn \rightarrow mxn$.

Here, $\mathbf{X}_{list}$ and $\mathbf{Y}_{list}$ are input/output coupling lists, used to define the model interface ***I***. ***X*** and ***Y*** represent the input/output event sets. The space size is defined by {m, n}, and **N** defines the neighborhood shape. C, together with B, the set of border cells, and Z the translation function define the cell space. The B set defines the cell's space border. If this set is empty, the space is "wrapped", meaning that cells in one border are connected with those in the opposite. In this case, every cell in the space will be considered as having identical behavior. Otherwise, the border cells need to be provided with a behavior different from those of the rest of the model. Finally, the Z function allows defining the coupling of cells in the model. This function translates the outputs of m-eth output port in cell C_{ij} into values for the m-eth input port of cell C_{kl}. Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood [01]. The ports' names are generated using the following notation: $P_{ij}{}^{X_q}$ refers to the q-eth input port of cell C_{ij}, and $P_{ij}{}^{Y_q}$ to the q-eth output port. These ports correspond with the port names denoted as X_q or Y_q for each cell. This definition, valid for bidimensional models, has been extended for n-dimensional spaces in [12].

CD++ [5] is a tool built to implement the DEVS and Cell-DEVS theory. The toolkit has been built as a set of independent software pieces, each of them independent of the operating environment chosen. The tool allows defining models according to the specifications introduced in the previous section. The models are built as a class hierarchy, and new atomic models can be incorporated into this class hierarchy by writing DEVS models in C++, overloading the basic methods representing DEVS specifications: external transitions, internal transitions and output functions. Once an atomic model is tested, it can be stored in a model database and then combined into a multicomponent model. Coupled models are defined using a specification language specially defined with this purpose, following DEVS formal definitions.

CD++ can also be used to define Cell-DEVS models. The tool includes an interpreter for a specification language that allows describing the behavior of each cell, including the local computing function and a delay. In addition, it allows defining the size of the cell space and its connection with other DEVS models, the border and the initial state of each cell. This language was defined following the theoretical definitions for the Cell-DEVS formalism.

The behavior specification of a cell is defined using a set of rules, each indicating the future value for the cell's state if a precondition is satisfied. A delay is associated with each of the rules, and the state changes will be distributed to the neighbors only after this delay. The local computing function evaluates the first rule, and if the precondition does not hold, the following rules are evaluated until one of them is satisfied or there are no more rules. For instance, Figure 1 shows an example for the specification of a Cell-DEVS model developed using CD++. The specification follows Cell-DEVS coupled model's formal definitions. In this case, Xlist = Ylist = { ∅ }. The set {m, n} is defined by *width-height*, which specifies the size of the cell space (in this example, m=20, n=40). The N set is defined by the lines starting with the

neighbors keyword. The border (B) is wrapped. Using this information, the tool builds a cell space (specified by C in the formal specification), I/O ports, and the Z translation function following Cell-DEVS specifications. Each cell in the cell space is built following Cell-DEVS specifications for atomic models. The X, Y, S, N, θ, δ_{int}, δ_{ext}, λ, and D functions are built following Cell-DEVS definitions (see [03] for details). The user only needs to define the τ function (defined by *localtransition*), and the delay (defined by *delay*, the delay values in each rule, and *defaultDelayTime*).

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

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

Figure 1. A Cell-DEVS specification in CD++

The outputs of the cells are delayed by using a specified time. The main operators available to define rules and delays include: Boolean, comparison, arithmetic, neighborhood values, time, conditionals, angle conversion, pseudo-random numbers, error rounding and constants (i, e, gravitation, acceleration, light, Planck, etc.). n the example presented in Figure 1, the local computing function executes very simple rules. The first one indicates that, whenever a cell state is 1 and the sum of the state values in N is 8 or 10, the cell state remain in 1. This state change will be spread to the neighboring cells after 100 ms. The second rule states that, whenever a cell state is 0 and the sum of the inputs is larger or equal to 10, the cell value changes to 1. In any other case (*t* = true), the result remains unchanged, and it will be spread to the neighbors after 150 ms. As we can see, cells evolve using a discrete-event approach.

```
Message * / 00:00:08:686 / Root(00) to top(01)
Message * / 00:00:08:686 / top(01) to processor(03)
Message Y / 00:00:08:686 / processor(03) / out / 10.00000 to top(01)
Message D / 00:00:08:686 / processor(03) / ... to top(01)
Message X / 00:00:08:686 / top(01) / done / 10.00000 to queue(02)
Message X / 00:00:08:686 / top(01) / solved / 10.00000 to transducer(04)
```

Figure 2. A fragment of a log stream.

CD++ simulator is message-driven, and each message represents an event with its associated timestamp. The simulation outputs can be recorded into a log. Using this stream as input, we can reproduce the state of each model, and we can use it to analyze model outputs. For instance, figure 2 shows an excerpt of a log stream showing different messages evolving in a simple DEVS coupled model, which represents a CPU connected to a Queue, and a Transducer that computes performance metrics. There are four different messages involved: X (inputs), Y (outputs), * (internal transitions) and D (done messages). In the first message of Figure 2, a model called *processor* is activated due to an internal transition. The model thus generates an output (the value `10.00000`, which is sent through the *out* port), and executes the internal transition function. After that, a *done* message is generated, including the scheduled time for the following internal event (in this case, infinity, represented as `"..."`). The Y message is translated into an input message (X) that is transmitted to two different submodels (*queue* and *transducer*).

By using these output streams, CD++ can generate a text stream representing the execution of Cell-DEVS models as a set of 2-dimensional slice. Figure 3 shows a fragment of the output generated by this facility (called **drawlog**) for a two-dimensional model of size 10x10.

```
Line : 1144 - Time: 00:00:03:050
              0    1    2    3    4    5    6    7    8    9
         +----------------------------------------------------+
        0| 23.6 24.4 24.5 24.4 24.2 24.0 23.4 22.8 22.3 22.8|
        1| 23.5 24.7 25.1 24.7 24.4 24.0 23.2 22.4 21.5 22.4|
        2| 23.3 25.1 29.1 25.1 24.5 24.0 22.8 21.5 20.3 21.5|
        3| 23.5 24.7 25.1 24.9 24.7 24.5 23.5 22.5 21.5 22.4|
        4| 23.8 24.4 24.5 24.7 24.9 25.0 24.3 23.5 22.8 23.2|
        5| 24.0 24.0 24.0 24.5 25.0 28.2 25.0 24.5 24.0 24.0|
        6| 23.8 24.0 24.0 24.3 24.7 25.0 24.5 24.0 23.5 23.7|
        7| 23.7 24.0 24.0 24.2 24.3 24.5 24.0 23.5 23.0 23.3|
        8| 23.5 24.0 24.0 24.0 24.0 24.0 23.5 23.0 22.5 23.0|
        9| 23.7 24.0 24.0 24.0 24.0 24.0 23.7 23.3 23.0 23.3|
         +----------------------------------------------------+
```

Figure 3. A fragment of an output stream

This fragment shows the results of executing a heat diffusion model. The cells at (2, 2) and (5, 5) are connected to a heating device (in this case, they have received a heat flow of 29.1 °C and 28.2 °C). The cells (8, 8) and (2, 8) are connected to a source of cold (i.e., two open windows). The initial temperature in the room is 24.0 °C. A cell's temperature value is measured computing the average of the temperature values in the cell's neighborhood. In models of three or more dimensions, the results can be shown as slices representing 2-dimensional planes, each of them shown as a matrix like the one in Figure 3. For instance, in a 3D dimensional space, the first plane corresponds to (x, y, 0), the second one to (x, y, 1), etc. Figure 4 shows a model executing a 3D simulation of the 'Life' game [13] with the original rules proposed by Conway. Each cell can be alive (1) or dead (0). A new cell is born when it has exactly three living neighbors. An existing cell survives if it has two or three neighbors that are alive. Otherwise, it dies.

```
Line : 247 - Time: 00:00:00:000
          0123456            0123456          0123456
         +-------+          +-------+        +-------+
        0|1      |         0|       |       0|1      |
        1|1 1  11|         1|11   11|       1|  111  |
        2| 1   1 |         2|   11 1|       2| 1 11  |
        3|       |         3|  1  11|       3|     11|
        4|  1  11|         4|  1 1  |       4| 1   11|
        5|  11  1|         5|   1 1 |       4| 11  1 |
        6|1  1  1|         6| 1   1 |       4| 1 11 1|
         +-------+          +-------+        +-------+

Line : 247 - Time: 00:00:00:100
          0123456            0123456          0123456
         +-------+          +-------+        +-------+
        0| 1    1|         0|11    1|       0| 1    1|
        1|1 1   1|         1|1     1|       1|1 11  1|
        2|11  1 1|         2|1    1 |       2|11   11|
        3|    111|         3|  1 1 1|       3|    1 1|
        4|       |         4|     11|       4|       |
        5|1  111 |         5|1 111 1|       4|1  11 1|
        6|       |         6| 1     |       4| 1  1 1|
         +-------+          +-------+        +-------+
```

Figure 4. A fragment of a 3D model.

3. RELATED EFFORTS

At present, there are a number of efforts devoted to develop DEVS models and others focusing in cellular models, but none of them meet our requirements: we intend to provide tools to build complex physical systems that can be executed remotely in a high performance platform, being able to visualize the results in local workstations. Several existing tools are devoted model and simulate CA. Some of them provide good visualization facilities; others enable distributed execution of the models. At present, dozens of CA toolkits have been developed. We will briefly describe of the existing tools following.

- **MJCell** [014] is a Java applet to simulate CA. Its main purpose is exploring existing and creating new rules and patterns of 1-D and 2-D CA. It can use rules from thirteen different CA rules families, and allow experimenting with new rules.
- **Cellsprings** [15] is a powerful 2D CA Java applet. More than seventy CA rules are predefined, and the users can define, run, and save their own arbitrary rules. The users also can change the size of the model and the color map, specify some characters of the model, then initiate and run the model.
- **Trend** [16], is a general-purpose 1D or 2D CA simulation system. It is flexible about the space sizes, cell and neighborhood structures and CA rules. It also has a backtracking feature that simplifies rule set development.
- **SpaSim** [17] allows the user to build, simulate and perform spatial and spatial-temporal analysis on the same environment using a friendly user interface. It has no real 3D visualization, but a dynamic window dialog containing several tabs, one for each of the automata layers.
- **Capow** [18], is a program for evolving 1D and 2D CAs. The user can control the simulation and the visualization with color, selecting the type of view and 3D view details. However, for 3D visualization, the tool needs to create a VRML output file with the result. In addition, it only displays the surface of the 3D graph.
- **PascGalois** [19] can produce innovative 3D visualization of 2D CA. It lays different results over together, or changes the 2D graph to 3D graph (rolling a 2D graph to implement the idea in 3D).
- **CASim** [a20] is an environment for simulating 1D, 2D and 3D cellular automata. The user designs the model by giving the names and number of states, state transition rules, colors and icons.

Besides the lack of visualization facilities or support for distributed execution, these tools also have problems related with the synchronous execution of the cellular models. In [021] we showed that the use of a discrete time base poses restrictions in the precision and efficiency of the simulated models. If complex CA are considered, higher precision can only be achieved by reducing the activation period for each time step. Therefore, large amounts of compute time will be wasted to obtain the desired results. Furthermore, most cells do not need to be updated in each time step. The existence of these "quiescent" states allowed defining different modifications in which the automaton advances using instantaneous events that can occur at unpredictable times.

The Cell-DEVS formalism is one of the existing approaches providing the advantages of asynchronous execution. As DEVS and Cell-DEVS are discrete event formalisms, higher precision and speedups in the simulations than the discrete time approaches used by traditional CA. In [22], the authors showed that DEVS combined with parallel simulation techniques can produce speedups of up to 1000 times. In [21], it was showed that Cell-DEVS model execution also

provides execution speedups when compared with CA. these advantages. Besides this, Cell-DEVS models enable integration with other models defined with different techniques, improving model definition.

Therefore, we also investigated existing DEVS tools, in order to see the possible integration with executing Cell-DEVS models. At present, there are different DEVS modeling tools, but most of them do not provide facilities for execution of cellular models. Some of them do not even provide visualization facilities or distributed execution services. All of them provide much flexibility for the users to develop their own models. We summarize the main features of some of these tools following.

- **ADEVS** [023] provides a C++ library based on the DEVS formalism. No distributed environment, or visualization tools have been implemented. To use it, the users should have basic familiarity with DEVS, and use the classes in the library to construct their own models. The users should decide how to output the result files, and design the corresponding visualization tools.
- **PyDEVS** [24] uses the ATOM3-DEVS tool to construct DEVS models. The models are represented as a graph used to generate Python code. The users can add nodes, ports and links, and edit them according to the real system. The model files are saved in directory structure matching the hierarchical structure of the model. For each atomic or coupled DEVS model, a Python file is created. Python DEVS deals with the graph model, the code generation and execution, and does not introduce remote execution environment or visualization of the results.
- **SimBeams** [25] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of components that can be used in model creation, result output, analysis and visualization using DEVS. For actual simulation applications, the users need to select suitable components and place them on a worksheet to build models and connect them with external events. No remote execution facility is introduced, and the users should realize their own special-purpose simulation environments for particular application domains. All the components are displayed in 2D graphs.
- **JDEVS** [26] is a DEVS simulation engine written in Java. It enables general purpose, component based, GIS connected, visual simulation model development and execution. It was developed mainly to interact with Geographic Information Systems. It also provides easy-to-use 2D and 3D visualization tools. However, it has no powerful navigation functions and visualization edition functions to better check the visualization contents. In addition, no remote execution is considered here.

Some of the existing DEVS environments could be suitable to meet our goals. Unfortunately, none of these environments is able to run Cell-DEVS models in remote environments, or to display the results of executing cellular models in 3D. Some of them were provided with extensions for visualization of particular problems, but no generic visualization facilities are provided. **DEVS/C++** [027] is a DEVS-based modeling and simulation environment written in C++, which supports parallel execution. It provides classes for the users to implement their own DEVS models. The tool does not include 3D visualization for cellular models. **DEVS/Java** [28] is a DEVS-based modeling and simulation environment written in Java that supports parallel execution. It provides classes for the users to implement their own DEVS models. The users can use the interface to visualize the state of the components in the model, their ports and couplings. A model can execute in a web browser, but it does not provide client/server facilities. **DEVSim++** [29] is an object-oriented tool to develop DEVS models using C++. It includes a graphical user interface, but no support for the execution of 3D cellular models.

DEVS/HLA [30] is based on the High Level Architecture (HLA) [31] and DEVS. It is used to demonstrate how an HLA-compliant DEVS environment can significantly improve the performance of large-scale distributed modeling and simulation environments. HLA has been proposed and developed to support the reuse and inter-operation of simulations, and establish a common technical framework facilitating the inter-operability of all types of models and simulations. The user should implement **DEVS/HLA** models using a standard programming language, such as, C++. The tool does not provide visualization facilities, but it has been integrated with powerful visual displays [32]. The extensions of our toolkit were designed to make easy the future integration with these tools. If a common standardized representation for DEVS models is defined, we would be able to analyze the execution results for Cell-DEVS models while making the models ready to interact with others developed under DEVS/HLA, DEVS/C++ or DEVS/Java, thus making possible interoperability between these advanced environments.

As we are interested in building a web-based environment, the use of VRML (Virtual Reality Markup Language) [33] appeared to be a good choice to build our visualization interface. VRML is a web-based graphics language for creating 3D models. It provides a file format for describing 3D objects and worlds, and it allows user interaction within a scene through viewpoints, movement, and rotation. VRML worlds are created with a scene-graph structure. Scene graphs are comprised of various groups of nodes, which are responsible for displaying shapes, interact, and navigate through the world. The External Authoring Interface (EAI) [34] is a Java API, which enables a currently running applet to interact with, and update a 3D VRML scene, letting the users to create dynamic VRML worlds.

We used some of the basic VRML constructions in order to develop our visualization facilities. The basic standalone structure in a VRML file is called a *node*, and it describes shapes, colors, lights, viewpoints, sub-scenes, sensors (used to sense user input), position and orient shapes, etc. A VRML scene is composed of various groups of nodes, which create a hierarchical architecture to represent a VRML scene [34]. **Grouping** nodes are used to band other nodes into a common entity. They have a children field, which can contain a list of nodes, and events (methods) to add or remove nodes from this field. All the nodes in the children field can treated and manipulated as a whole. **Group** nodes simply serve as a container for children nodes. **Transform** nodes allow manipulating the entity's size, position, and orientation. **Inline** nodes are used as children of any grouping node. **Defining** nodes define names for nodes in the scene, in order to facilitate the access to it. **Sensors** are used to generate events based on user actions, such as a mouse click or navigating close to a particular object. **Geometry** nodes should be contained by a shape node in order to be visible to the user. A shape node contains exactly one geometry node in its field. Each shape can be manipulated by changing its parameters, such as, the radius of a sphere, the height and width of a box, the color of a cone. **Navigation** nodes are used to set up predefined "camera" viewpoints within a virtual world. **Viewpoints** are used to predefine a position and orientation that automatically takes a viewer to that location when first entering the world. Additional viewpoints can be defined that enable the viewer to easily navigate to predetermined areas throughout the world. We will show how to use these facilities to build our visualization environment in section 4.

4. DESIGN OF A SIMULATION CLIENT/SERVER PLATFORM FOR CD++

Considering the background information presented in the previous sections, we decided to define a set of tools to improve the definition of Cell-DEVS simulation models, visualizing and analyzing the results, and accessing to a remote server for

execution. The idea was to incorporate all of these components together as a client/server simulation engine. Here, we will describe the design aspects and the inter-relationship of the components in this engine.

Client/server architectures are popular in distributed systems, in which the resources are spread over various computing platforms and in distant locations. The inherent modularity of client/server systems leads to greater overall robustness, since computing responsibility is no longer concentrated in one computer. Client/Server applications typically distribute the software components so that the data source resides on the server, the user interface resides on the client, and the logic resides in either, or both sides.

Following these ideas, the CD++ simulator was modified to run as a stand-alone application or as a server. In stand-alone mode, CD++ can be seen as integrated by the components presented in Figure 5. As we see, there is a separation between model definition, simulation execution, and visualization tools, and the interaction is done through input/output streams. Users run CD++ on the local machine using the CD++ Modeler to specify a model. This model is specified according to the DEVS and Cell-DEVS specifications, and it can be executed by a DEVS simulation engine. After the simulation is over, a log stream like the one in figure 2 is generated.

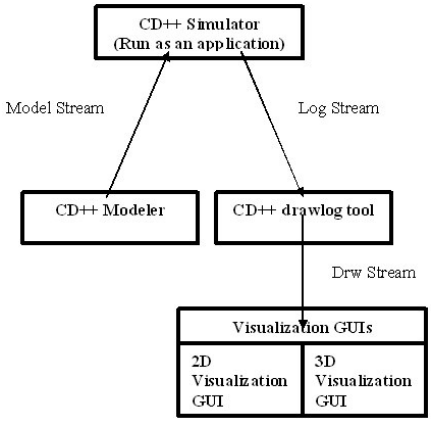


Figure 5. CD++ running as stand-alone application.

In order to achieve high-performance execution, the original abstract simulator mechanism (which was based on the techniques described in [35]) was extended to enable parallel execution using a distributed memory approach with message-passing. In this abstract simulator, DEVS processors are specialized into two different simulation engines, *simulators* and *coordinators.* The structure of these hierarchical processors mimics DEVS models hierarchy. The role of a *simulator* is to invoke an atomic model internal and external transition functions. On the other hand, a *coordinator* is attached to a coupled model and it has the responsibility of translating its children output events and of keeping the time of the next imminent dependants. Figure 6 shows the relationship between models and simulators. As we can see, the root coordinator (in charge of managing the higher level of the simulation, creating input/output streams and managing time and end conditions) is connected to a coordinator which is associated with the top level coupled model (*TOP*). The *CoupledModel#2* is associated with a coordinator, while each atomic model is associated with a simulator.

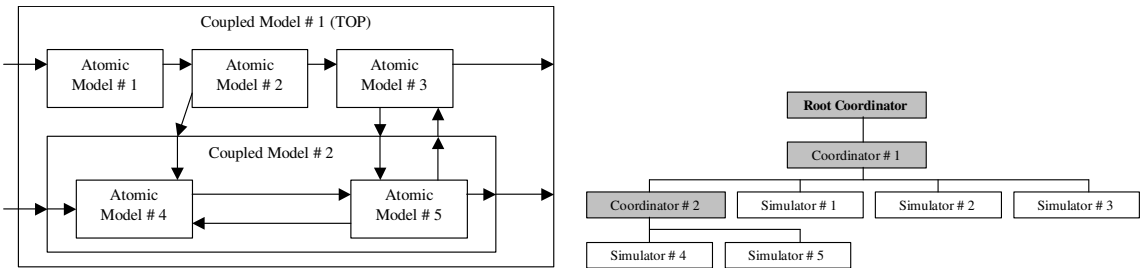


Figure 6. Models/Processors relationship.

This modular definition enabled us to extend the CD++ simulation engine in order to run DEVS models in a distributed platform, as shown in Figure 7. As we see, the simulation engine was replaced by a new application, able to run parallel simulations of DEVS and Cell-DEVS models in a high performance environment.

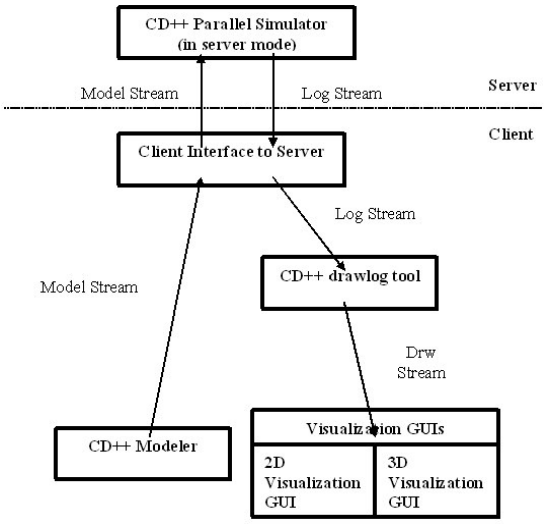


Figure 7. CD++ running as client/server application.

The new extensions were built on top of a modified version of the Warped kernel [36]. The Warped project was dedicated to the implementation of a simulation API to support different parallel simulation kernels. We mainly used two of the kernels provided in Warped: an optimistic kernel that implements the TimeWarp protocol and a NoTime kernel that uses no synchronization. A conservative kernel that complies with the Warped API was introduced in [37]. Having three different simulation kernels with the same API, Warped proved to be ideal for implementing the parallel CD++ simulator.

The Warped kernel presents an interface in which objects are modeled as entities that send and receive events to and from each other, and act on these events by applying them to their internal state. The kernel provides basic functions for the application to send and receive events, with periodic state saving for a potential rollback and recovery process. All DEVS processors have been defined as Warped objects. In order to define new atomic models, CD++ provides an *Atomic* abstract class that a modeler must extend, which is described in Figure 9. A new atomic model is created by including a new class derived from *Atomic.* In doing so, the following methods may be overloaded:

- ***initFunction***: this method is invoked when the simulation starts. It allows to define initial values and to execute setup functions for the model.

- ***externalFunction***: this method is invoked when an external event arrives from an input port.
- ***internalFunction***: this method is started when an internal event occurs (that is, the value of *sigma* is zero).
- ***outputFunction***: this method executes before the internal function, in order to generate outputs for the model.

```
class Atomic {
// Methods the user should define
Model& internalFunction();
Model& externalFunction (MessageBag&)
Model& outputFunction();
Model& confluentFunction();
ModelState* allocateState();

//Simulation kernel services
void sendOutput( Port&, BasicMsgValue* );
const Vtime& lastChange();
void holdIn( state, Vtime );
};
```

Figure 9. The Atomic class.

After defining the model and the set of available processors, it only remains to define how the models will be distributed. The modeler must create a partition file that tells CD++ in which machine to run each atomic model. Using this information, CD++ decides the location of the *coordinators* and it creates them. The hierarchical structure of simulators and coordinators is such that every coordinator has a set of child DEVS processors. Nevertheless, when a simulation runs in a parallel environment, coordinator's children might run on different processors. As every coupled model is associated to only one coordinator, every message sent to its child processors on a different CPU will require interprocess communication. Figure 10(a) illustrates this case. The coordinator in CPU 0 sends messages to its 8 children distributed on two CPUs. Four interprocess messages are required for the four children running on CPU 1. This can be avoided if every coupled model has more than one coordinator, each on a different processor. Figure 10(b) illustrates this case. For the same coupled model, there are two coordinators, and only one message is sent over the network.

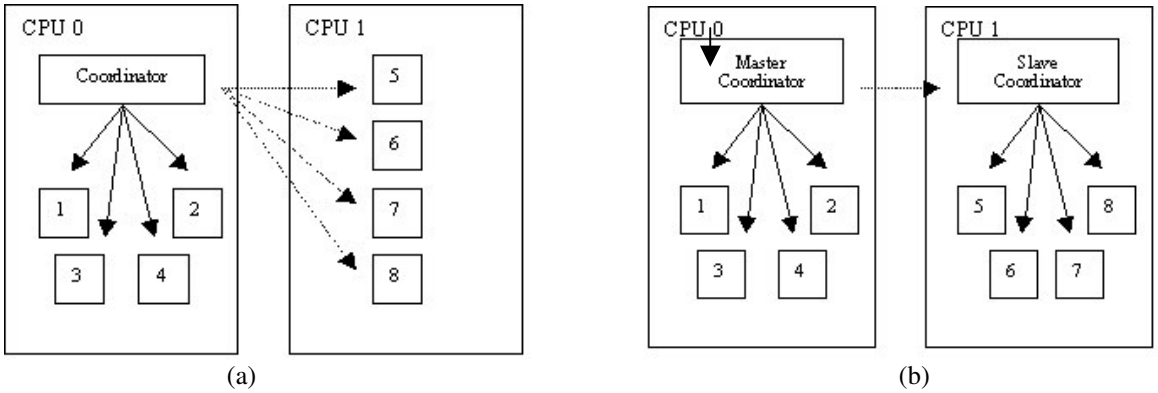


Figure 10. (a) Single coordinator sending messages to its children processors. Dashed lines = interprocess messages. (b) A master-slave pair sending messages to the children processors [38].

A coordinator will be created on each CPU with a child processor. Children processors will send messages to the local coordinator, which will decide how to handle them. Upon receiving a message from a child, a coordinator could forward it to other coordinators in other CPUs for the model. This requires all coordinators to know about the others. For instance, if coupled model *A* is a child of coupled model *B*, then *B´s* coordinators have to interact with *A´s* coordinators. If not handled carefully, this communication can turn out producing the same number of interprocess messages we wanted to

avoid. In such scenario, a way of keeping the number of interprocess messages to a minimum is to have only one of the coordinators to handle all messages to the parent model local coordinator. This specialized coordinator will be known as a *master coordinator* and all other model coordinators will be *slaves.* The *master coordinator* for model *A* will then be the only one that can receive or send messages to *B´*s local coordinator [39].

With the exception of the top level DEVS processor, known as *root coordinator*, all DEVS processors will have a parent *coordinator*. To set the parent-children relationship on a parallel environment, the following rules apply:

a. for each *simulator*, the parent coordinator will be the parent's model local processor (it is guaranteed that it will exist)

b. for each *slave coordinator*, the parent coordinator will be the model's *master coordinator.*

c. For each *master coordinator*, the parent coordinator will be the parent's model local processor; just as if it were a *simulator*.

When CD++ is running as a server, it expects receiving a model specification on a given TCP socket. The clients needing service will communicate with it and will send model streams through this input socket. The server was built to service several clients at the same time, and whenever it accepts a simulation requirement, a child process is created to serve the specific requirement. The most recently created process takes the place of its parent and waits for the next request in the input socket. When a new incoming simulation request arrives, the described process is repeated; a copy of CD++ server is created as a child process, and the simulation request is satisfied. When the simulation is over, the process is terminated. In the case of concurrent requests, multiple requests can be serviced concurrently.

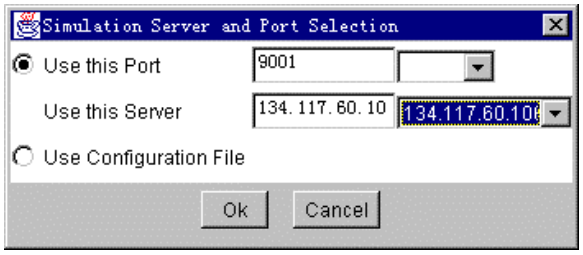


Figure 11. Setting up the CD++ server.

In order to execute a simulation, the client must send a *model* file like the one defined in figure 1, an optional *event list* and an optional *stop time* through the network. When a request is received, the CD++ server executes the model, and it returns the result through the same TCP port, with the format described in figure 2. The CD++ server will send back this log stream, and the client will be able to manipulate it. In our case, the client will save the results on the local disk, and later we will activate the CD++ drawlog to change its format into a text stream that can be used with visualization purposes. A user willing to run a model using these facilities must follow these steps:

a) **Set a Configuration File**: this file stores the server address, the local port number to be used, and a directory to save this default information. When the client starts, this file will be read and the default server, port and directory will be set.

b) **Select the model stream(s)**: the user must pick the model stream(s) that will be sent to the server for simulation.

c) **Change the Server and input socket**: a user can change the server and port addresses to be different from those defined as default in the configuration file (see Figure 11).

d) **Connect with the Server**: the client sends the chosen model file(s) to the server, receives the execution results, and then saves them in the default directory.

The main classes related to these procedures and their relationships are shown in Figure 12. Once a model file is sent to the remote server, the client waits for a response by listening to the input port defined in the configuration file. The results are stored into a buffer (*Queue*), which is saved to a file after some time. In order to do this, three different threads are used:

- Listening thread: it is always ready to receive data in the input port. Once a result arrives, it starts a read thread.

- Read thread: it reads the result, which is stored in a buffer. When the amount of data in the buffer reaches a threshold, it starts a saving thread and it resets the buffer.

- Saving thread: it appends the buffer data to the result stream.

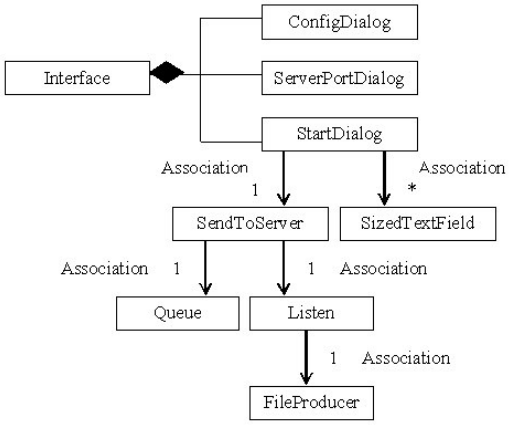


Figure 12. Class diagram of CD++ client interface.

5. USER VISUALIZATION ENVIRONMENT

This new client/server simulator was integrated with different modeling and visualization facilities. One of the basic components is the *CD++ Modeler*, which consists of a set of utilities to let the users to define DEVS models using graphical notations. This application can be used to create atomic or coupled models that can be executed by the CD++ simulator. The basic functions of the Modeler include creating atomic models using DEVS graphs [40], and coupled models using directed graphs [2]. The Modeler also includes a text editor to write and modify Cell-DEVS models and to run the simulation engine locally or remotely. The tool, coded in Java, enables execution on various environments. The main screen of the application is a design space, which is depicted in Figure 13.

Before creating a model, the user should select the proper design space (for defining atomic or coupled models). The user can refer to models previously coded in C++, as the ones presented in section 3, after being added to the CD++ model base. Besides this, Atomic models can be defined as DEVS graphs. Every state in a DEVS graph is composed by a state identification and a lifetime.

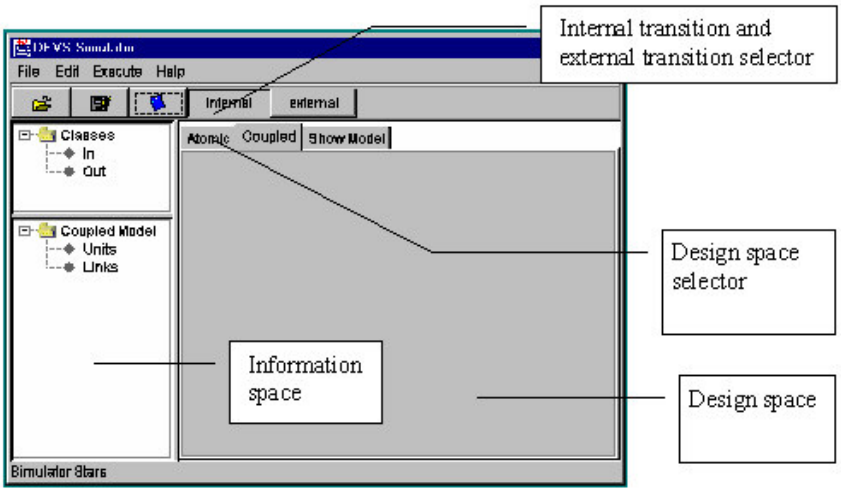


Figure 13. CD++ Modeler design space

Figure 14 shows the definition of a simple DEVS graph. The states are defined as circles with a name and a lifetime. For instance, the state *end* has a lifetime of 10 time units. Every state change can be associated with input/output activities, thus, input/output ports might need to be associated to each state. For instance, the *start* is associated with two input/output ports: *in* and *out*.

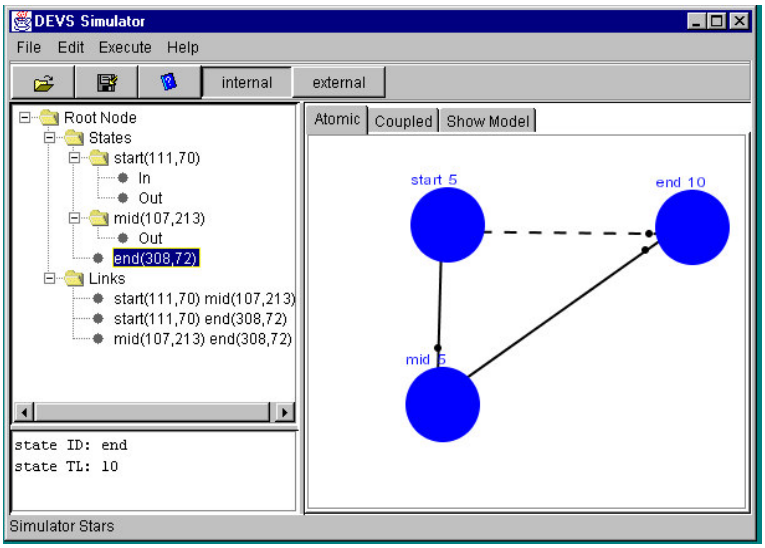


Figure 14. CD++ atomic model definition.

States are interconnected using different links, representing the transition functions. Internal transitions are represented by full lines, and external transitions by dotted lines. External transitions can be associated with an input port, which represents an input for the model. For instance, Figure 14 shows that every time an input is received through the *in* port and the model is in *start* state, the model will execute an external transition and will change to the *end* state. Internal transitions are associated with output ports (representing execution of the output function). For instance, if the lifetime of state *mid* is consumed (e.g., 5 time units have been elapsed), the output function is executed and the current state and time of the model are sent through the *out* port. After, the internal transition executes, and the model state changes to *end*.

The tool also permits creating coupled models on the coupled model workplace, which are defined as graphs connecting internal components or input/output ports. The first step to build a coupled model is to select the Atomic or Coupled subcomponents, which can be chosen from the ones previously added to the model database. Then, we can give a new name to an instance of the chosen model. Models within coupled models are represented as squares, as it can be seen in Figure 15. For instance, the *queue* model is an instance of the *Queue* atomic model previously defined. The user can define different instances of the same model. Once these components have been created, we can establish links connecting the input/output ports to each component. For instance, Figure 15 shows that the output port *out* in the *generator* model is connected to the *in* port in the *queue* model. Finally, we can establish links between output ports in a component, and input/output ports of the coupled model under definition. For instance, the *throughput* port in the *transducer* model of Figure 15 will be connected to the *throughput* port of the coupled model being defined (represented by a circle).

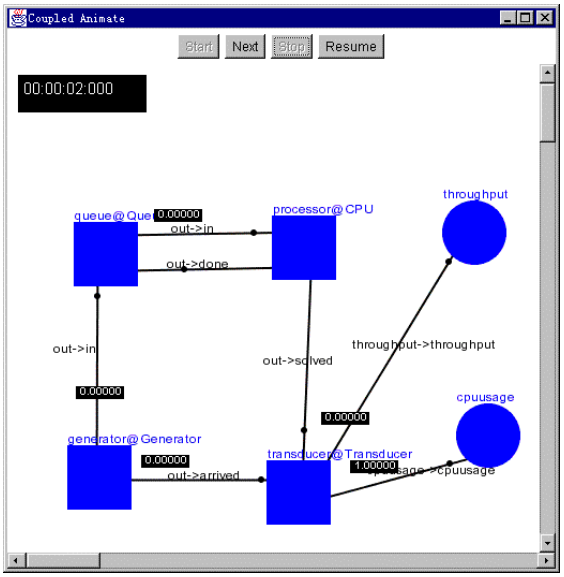


Figure 15. CD++ coupled model definition and execution.

The modeler can launch the local or the remote simulators. Once the simulation finishes, the user can analyze the execution results using different visual tools to enable better model validation. One of these facilities enables analyzing the input/output trajectories of an atomic model by studying the values transmitted into each of its input/output ports. Usually, a DEVS coupled model includes many atomic components, and the information for each of them is collected in the log stream during the simulation. Accordingly, every message value sent or received by a specific atomic model can be extracted and visualized. The execution results of this application can be seen in Figure 16. Besides this, execution of coupled models can be visualized by associating the digraphs representing coupled models, with the log stream generated during the execution of the coupled model. The graphical specification for a coupled model can be defined using the CD++ Modeler, and the log file will contain the information needed for displaying. In this case, we are able to see the input/output values transmitted during the simulation within a structural model, as it can be seen in Figure 15.

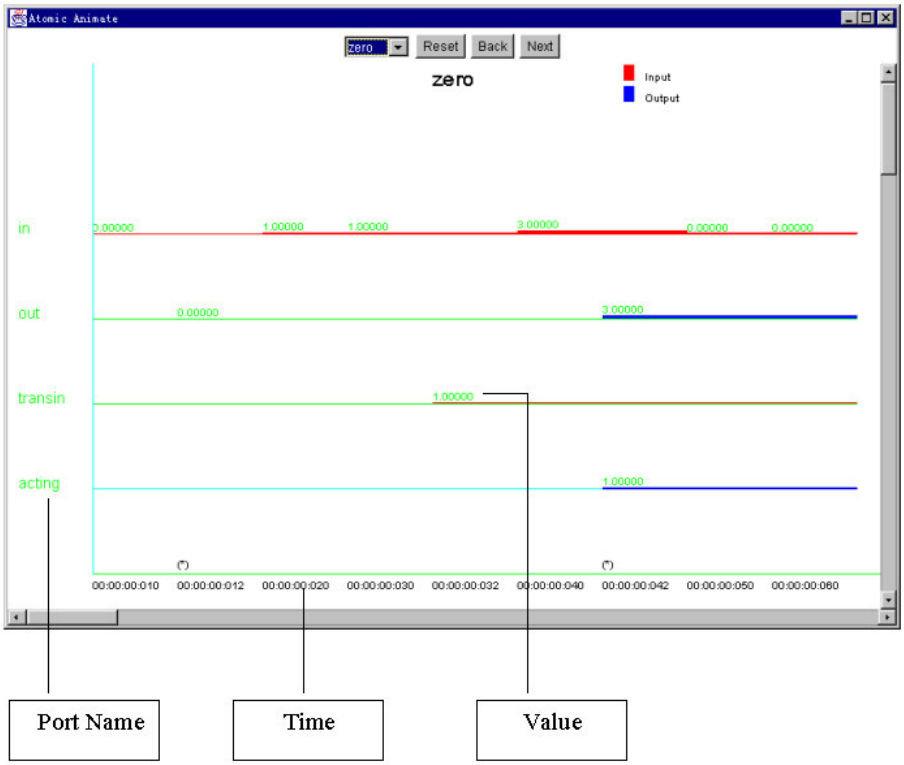
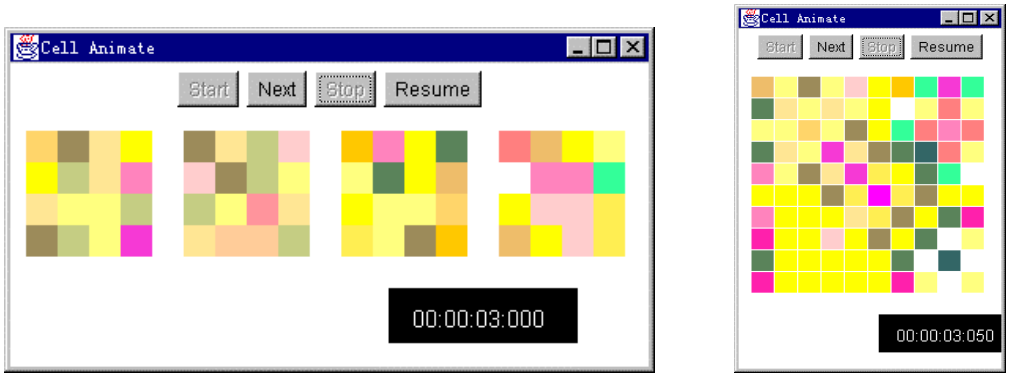


Figure 16. CD++ atomic model execution.

Cell-DEVS models can be created and modified using a built-in text editor, in which the user can specify model rules (as the ones presented in figure 1). In order to better visualize the execution results of Cell-DEVS models, we also added a new facility in order to visualize the outputs generated by the drawlog tool with a graphical interface. Simulation results for 2D Cell-DEVS models are shown in one plane by giving different colors to the different cell values (as seen in Figure 17.a.). Simulation results for 3D models are showed by displaying the values of all of the planes comprising the model simultaneously. In addition, different colors are given to different cell values in order to improve model visualization (as seen in Figure 17,b.)



(a) (b)

Figure 17. Cell-DEVS model execution. (a) 2D model (b) 3D model

A main addition to the visualization capabilities of the toolkit was to define a 3D visualization GUI in which the results can be seen in a 3D environment. In this application, the results are represented by nodes in a VRML scene. The user can

navigate in the scene, and edit the nodes for more convenient investigation of the results.

This visualization tool was built using several of the predefined VRML classes. We will explain some basic aspects about them here, but further information can be found in [33].

The **VRML root** file is embedded in an HTML file, and it is loaded as an empty scene to hold the nodes representing the result. Any of these nodes, which are used to represent the simulation results, should be authored as a child of the root file, so they can be added or removed by the applet according to the current results. In order to facilitate to identify the nodes in the scene and to visualize results, two nodes were also included in the root file.

- Background node**:** the background node allows defining the background of the VRML world.

- Viewpoint node**:** a viewpoint describes a predefined viewing position and orientation in a VRML world, acting as a camera in the real world. A VRML world can have any number of viewpoints (or cameras), that is, interesting places from which the user might wish to check the world. To facilitate the result visualization, we define a Viewpoint named *UserEye*, and a group of *viewpoints*, which will be used to switch to different viewing areas of the scene.

The **VRML** node class is used to create visible nodes in the scene representing the simulation results. The *Transform* nodes include every attribute needed to present a node in the scene. It allows manipulating a node's size, position, and orientation.

The **Inline** node is used to translate and rotate nodes in a VRML scene. It can be inserted as a child in a Transform node. Inline nodes have predefined shapes that already exist in VRML; only translation and rotation operations can be applied to these shapes.

The application starts with an empty VRML world that is embedded in an HTML file as a root file. This file also includes an applet to control the scene, having a variety of functions such as:

- Add or remove a node from the scene
- Change the shape and colors of the nodes, or hide the nodes
- Select the colors for the value ranges of the nodes
- Change the scale of the nodes, and the interval between the nodes
- Navigate in the Scene
- Edit the scene and the individual node
- Load a result stream to be displayed

These functions were classified into several categories, each of them implemented in a different panel, which are organized as in Figure 18. The **ReadDrwFile** class is in charge of reading the values to be displayed from the result stream. It is activated by the NavigatePanel when the user decides to view the next result. The result can be chosen from: a) the result with the next timestamp in the result stream; b) the result with the previous timestamp; c) the result with a user-defined timestamp. The **InfoPanel** class is shown when the application starts. It includes methods to select the result stream to be displayed and an associated color palette. These streams are checked for consistency, and a text area displays

debugging and status information.

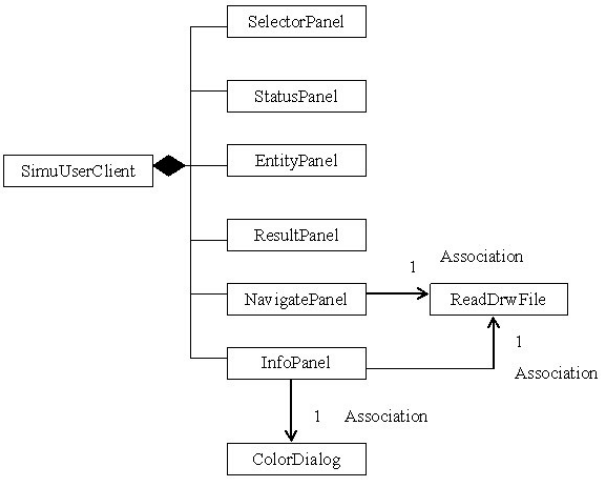


Figure 18. Class diagram of the visualization applet.

NavigatePanel is the main class in the application. It stores the currently displayed result, the recent displayed nodes, and the names of every displayed node. This information changes whenever a scene is updated with a new result, or the color palette is modified. It first initiates the scene as a block of nodes corresponding to the size of rows, columns and layers. Then, it associates each value in the result stream with a node in the block. The class includes methods to add or remove nodes in the scene, to change the shape, color, and size of the nodes, and to check the results from a favorite viewpoint.

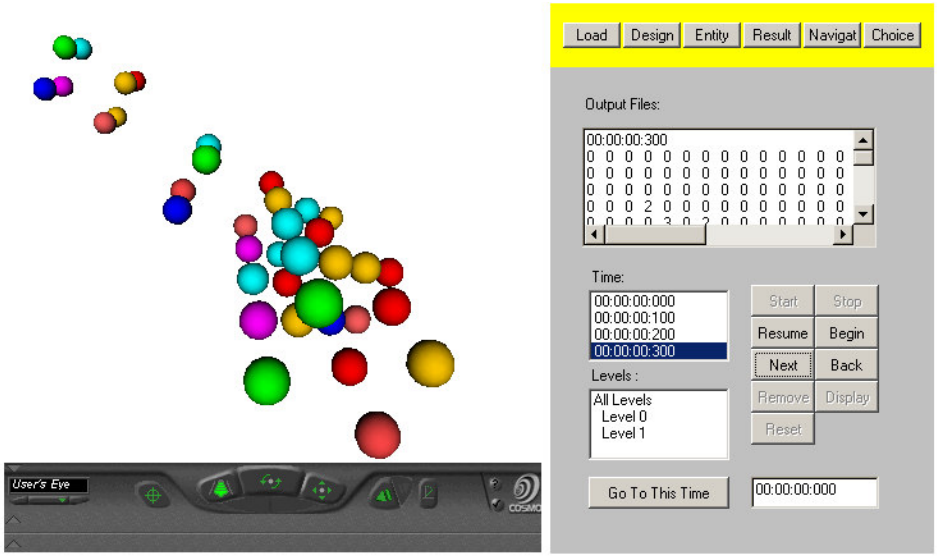


Figure 19. NavigatePanel execution.

This class also provides the functionality for *navigation*, which is implemented as the possibility of having different viewpoints. A viewpoint can be defined as a child of Transform nodes, therefore, its position and orientation change with those in the parent Transform node. The user can select different viewpoints, and if another viewpoint is bound, the active viewpoint will be unbound automatically, enabling the user to see the scene corresponding to the newly bound viewpoint.

It also enables to choose a node as the current node, and a method to remove layers in the scene or add the removed layers. A reset method for the scene enables redisplaying all the layers. Figure 19 shows the results obtained when the NavigatePanel class is executed.

The **EntityPanel** class allows editing individual nodes in the scene. A list of entities is populated by the NavigatePanel class every time it updates the nodes in the scene. A node can be picked up to become the current node. After being selected, it can be edited by calling the related methods in the node and NavigatePanel classes. The functions included in this class permit to change the shape, color, and size of the selected node, add or remove the individual node in the scene.

The **ResultPanel** is used to navigate in the VRML world. Different methods are defined to control the navigation, including: a) a start method to start the execution; b) a resume method to continue if stopped; c) a go back method to go to the previous timestamp; d) a go next method to go to the next time; e) a stop method to stop the visualization at a given time; f) a continuously display method (this iteration will end only at the end of file); g) a method to go to any selected timestamp. These methods call the corresponding methods in the ResultPanel Class, which activates the corresponding methods in the NavigatePanel Class.

CD++ generates text streams with the structure shown in figure 3, which represents the coordinates of all the nodes in the cell space, and the two end points of all the links between these nodes. The 2D node positions need to be transformed into a 3D VRML node. To do so, if we call (x_2, y_2) to a position in a 2D stream, and (x_3, y_3, z_3) for a position in a 3D file, we use the position x_2, y_2, and transform them into the x_3, y_3 positions of a VRML 3D space. Then, we use the remaining layer for the z_3 value. As the nodes have a non-zero size, in order to allocate the center of the 2D graph to the center of a 3D VRML scene, the coordinate x and y need following transformations:

$$x_3 = - (\max x_2 + \min x_2) / 2 + x_2$$

$$y_3 = - (\max y_2 + \min y_2) / 2 + y_2$$

For each link in the 2D model, we must:

(1) Get the corresponding 2D coordinate for the two end nodes $(x_{2\_start}, y_{2\_start})$, (x_{2_end}, y_{2_end}), computing y_2 with the above transformations.

(2) Calculate the length of the link l_{link}, which will be used in *scale* attribute in the Transform node, to let the link in 3D have the length l_{link}

$$L_{link} = \sqrt{(x_{2\_end} - x_{2\_start})^2 + (y_{2\_end} - y_{2\_start})^2}$$

(3) Calculate the link α_{link} and orient the link with θ_{link}, as illustrated in Figure 16.

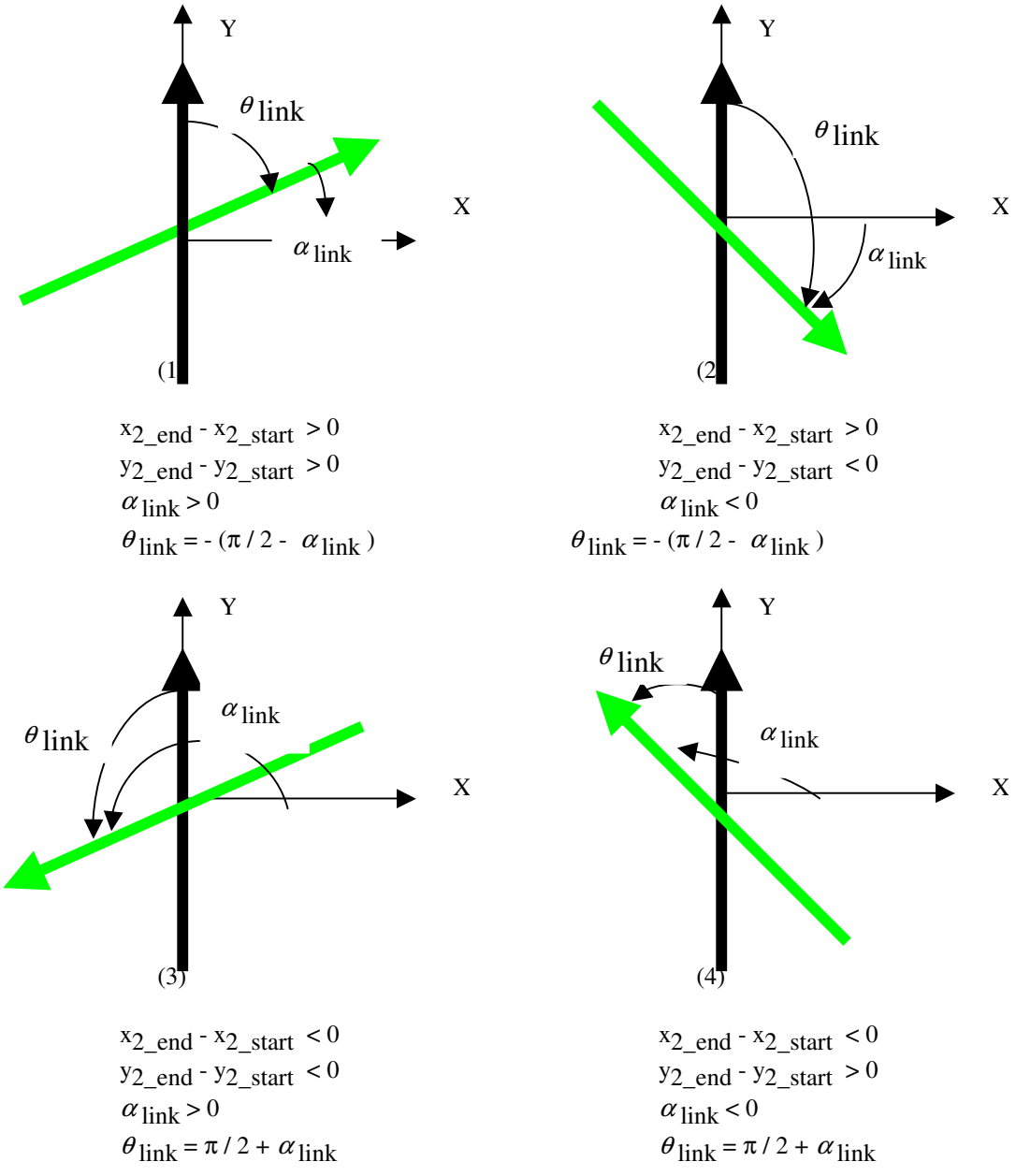
$$\alpha_{link} = \tan^{-1}((y_{2\_end} - y_{2\_start}) / (x_{2\_end} - x_{2\_start}))$$

(4) Translate the center of the link to the new center (x_{center}, y_{center})

$X_{center} = (y_{3\_end} + y_{3\_start}) / 2$,

$Y_{center} = (x_{3\_end} + x_{3\_start}) / 2$

These procedures are summarized in Figure 20.



(1) $x_{2\_end} - x_{2\_start} > 0$, $y_{2\_end} - y_{2\_start} > 0$, $\alpha_{link} > 0$, $\theta_{link} = -(\pi / 2 - \alpha_{link})$

(2) $x_{2\_end} - x_{2\_start} > 0$, $y_{2\_end} - y_{2\_start} < 0$, $\alpha_{link} < 0$, $\theta_{link} = -(\pi / 2 - \alpha_{link})$

(3) $x_{2\_end} - x_{2\_start} < 0$, $y_{2\_end} - y_{2\_start} < 0$, $\alpha_{link} > 0$, $\theta_{link} = \pi / 2 + \alpha_{link}$

(4) $x_{2\_end} - x_{2\_start} < 0$, $y_{2\_end} - y_{2\_start} > 0$, $\alpha_{link} < 0$, $\theta_{link} = \pi / 2 + \alpha_{link}$

Figure 20. Computing the Orientation Angle

5. VISUALIZING CD++ SIMULATIONS

In this section, we will show some examples of execution of models using the VRML 3D interface. We will not focus in the model definition or the execution results, but in the client functionality, including viewpoints, geometry, scales and

colors (see [56] for further details about model definition). In the following examples, we will use a 3D version of the heat diffusion model presented in figure 3.

As previously explained, we can use different geometries to represent the objects in the result space. The user can select boxes, spheres, cones or cylinders for the nodes in the scene. Two examples can be seen in Figure 21. The original result matrix is now shown as a 3D VRML model consisting of colored nodes with the same size. Each node corresponds to a value in the result matrix, and the color of the node is specified by its value and set with a color selection.

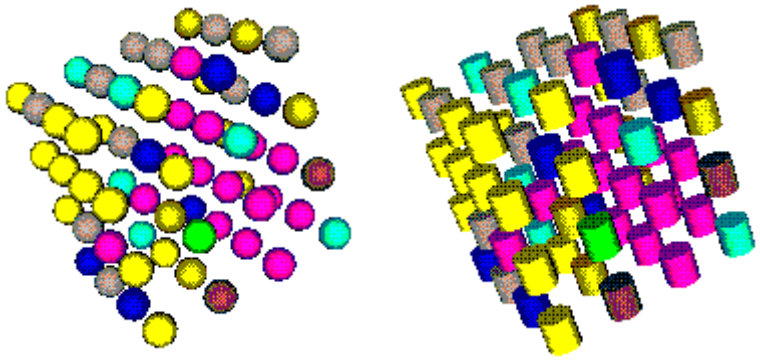


Figure 21. Changing geometries

Another of the facilities available enables selecting different viewpoints to visualize the results. This can be seen in Figure 22. A user can select any viewpoint defined in the VRML file to visualize the results. Here, we show the *User's Eye* and the *Side view* viewpoints. In addition, the user can select any viewing area, as shown in previous figures.

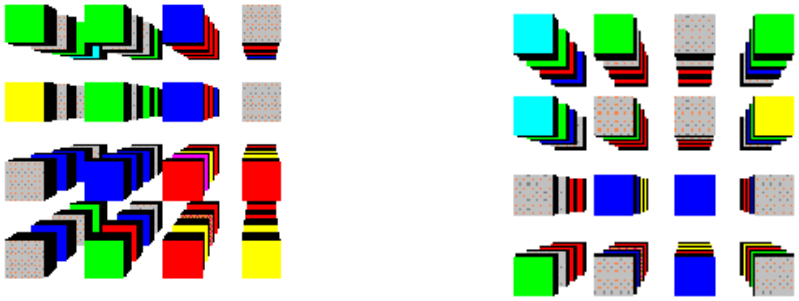


Figure 22. Different Viewpoints

The navigation facilities enable displaying the results following the sequence of the original simulation. A user can see the results continuously, advance one step at a time, move backwards, or advance up to a certain time following the simulation sequence. Figure 23 shows the execution results obtained using this facility.

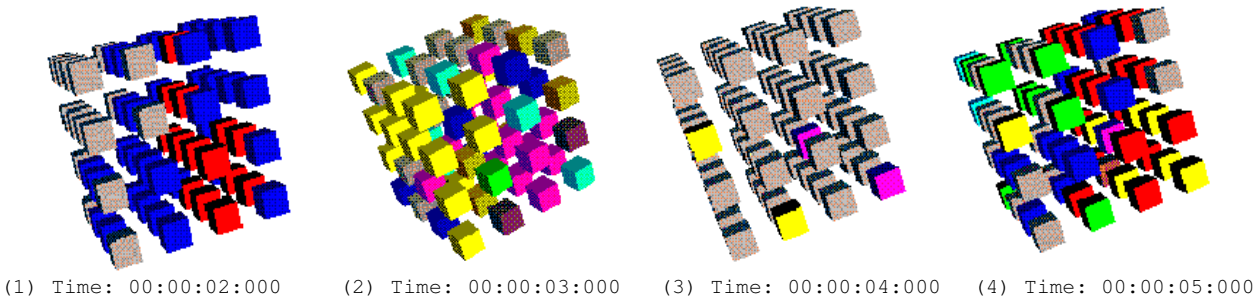


Figure 23. Continuous advance function.

The user can also edit a single node in the scene, changing its shape, color or position, as shown in the Figure 24. The edited node will keep the modified attributes. Therefore, the user can highlight the special nodes he wants to check. A user can modify a node with color, size, translation and rotation, or delete an edited node. The user also can redisplay a previously deleted node, then the display will be as in the original version, and all the modified attributes remain unchanged.

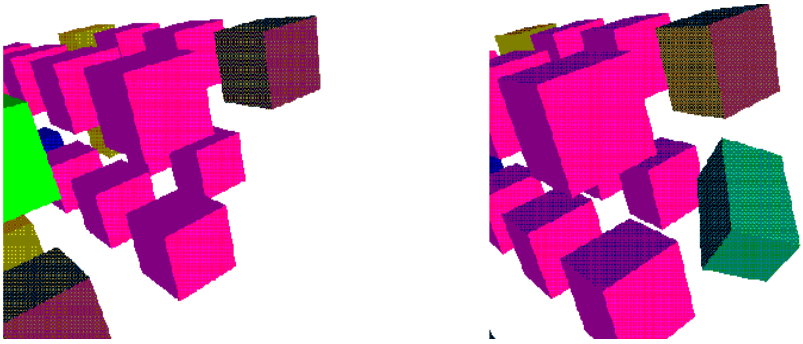
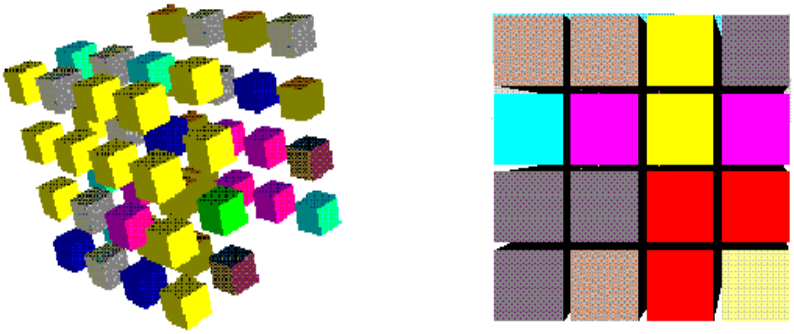


Figure 24. Editing single nodes

A user can also remove any layer in the display, in order to make easier the visualization of certain phenomena. In Figure 25.a), we show the previous examples, but level 1 was removed, which can be redisplayed later if needed. The nodes in the scene can be scaled up or down, as shown in Figure 25.b), where the nodes have been scaled to the minimum distance and cannot be scaled further. The nodes also can be scaled to smaller size.



(a) **(b)**

Figure 25. (a) Deleting layers (b) scaling nodes

Finally, as shown in Figure 26, multiple instances of the GUI can be activated to visualize the same result, using different viewing areas, as shown in the following figure. Likewise, different geometry or Inline nodes can be used, if needed.

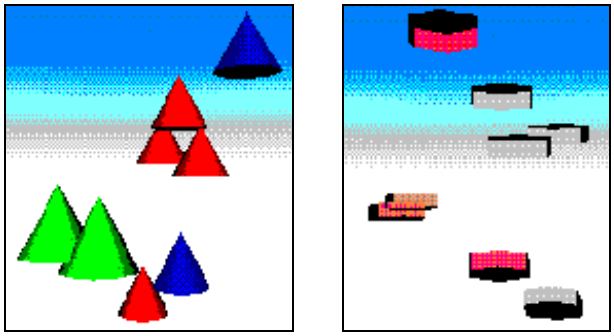


Figure 26. Multiview: different shapes from different viewpoints.

6. 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 using a client/server approach. This client provides a series of tools, including 3D simulation visualization and remote execution. The CD++ server was extended to provide high performance simulation by executing DEVS and Cell-DEVS models in a parallel fashion. The abstract simulator here defined keeps to a minimum the number of messages interchanged. This was possible by assigning each coupled model one *master coordinator* and zero, one or more *slave coordinators.* Messages that have to cross a processor boundary are always sent between *master* and *slave coordinators,* which then forward the received messages to their local dependants.

The 3D visualization GUI enables sophisticated visualization of Cell-DEVS models. To better understand the results, the user can select shapes to represent a node in the 3D space, select different colors, edit individual nodes and remove layers. The interface in the client can send models to a remote CD++ server, then receive, and visualize the results locally. This client also can support real-time multi-view, multi-user simulation, and run several different models simultaneously. The simulation server is installed for public domain. The simulator, visualization tools and client software can be obtained by accessing http://www.sce.carleton.ca/faculty/wainer/celldevs.

The current facilities have highly improved the use of the previously existing tools, thus enhancing the analysis experience of the modelers using the toolkit. At present, we are focusing in extending these results providing even more advanced visualization facilities, focusing in standard DEVS models and their interaction with VRML worlds. We also started to analyze the interaction between the CD++ toolkit and tools based on the HLA standard (such as DEVS/HLA), in order to facilitate interoperation of different DEVS models developed in different tools. The results will permit defining models using different DEVS environments, using them in a distributed collaborative environment based on HLA tools, and enabling advanced visualization using Virtual Reality tools.

REFERENCES

1. ZEIGLER, B. "Theory of modeling and simulation". *Wiley*, 1976.
2. ZEIGLER, B.; KIM, T.; PRAEHOFER, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". *Academic Press*. 2000.
3. WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modeling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", *Springer-Verlag*. 2001.
4. TOFFOLI, T. "Occam, Turing, von Neumann, Jaynes: How much can you get for how little? (A conceptual introduction to cellular automata)". *Proceedings of ACRI'94*. Rende, Italy. 1994.
5. WAINER, G. "CD++: a toolkit to define discrete-event models". G. Wainer. *Software, Practice and Experience*. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.
6. AMEGHINO, J.; TROCCOLI, A.; WAINER, G. "Modeling and simulation of complex physical systems using Cell-DEVS". In *Proceedings of 34th IEEE/SCS Annual Simulation Symposium.* Seattle, U.S.A. 2001.
7. AMEGHINO, J.; WAINER, G. "Application of the Cell-DEVS paradigm using N-CD++". J. Ameghino, G. Wainer. In *Proceedings of the 32nd SCS Summer Computer Simulation Conference*. Vancouver, Canada. 2000.
8. TROCCOLI, A.; AMEGHINO, J.; IÑON, F.; WAINER, G. "A flow injection model using Cell-DEVS". In *Proceedings of the 35th IEEE/SCS Annual Simulation Symposium.* San Diego, CA. U.S.A. 2002.
9. MUZY, A.; WAINER, G.; INNOCENTI, E.; AIELLO, A.; SANTUCCI, J.F. "Dynamic and discrete quantization for simulation time improvement: fire spreading application using the CD++ tool". In *Proceedings of 2002 Winter Simulation Conference*. San Diego, U.S.A. 2002.
10. KRAAK, M.-J.; MACEACHREN, A.M. "Visualization for exploration of spatial data". *International Journal of Geographical Information Science*, 13(4), 285-287. 1999.
11. MACKINLAY, J.; ROBERTSON, G.; CARD, S. "Rapid Controlled Movement Through a Virtual 3D Workspace", *Computer Graphics* 24(4), 1990.
12. WAINER, G.; GIAMBIASI, N. "N-Dimensional Cell-DEVS". In *Discrete Events Systems: Theory and Applications*, Kluwer. Vol. 12, No. 1. January 2002. pp. 135-157.
13. GARDNER, M. "The fantastic combinations of John Conway's New Solitaire Game 'Life'". *Scientific American*. 23 (4). pp. 120-123. April 1970.

14. WOJTOWICZ, J. “1D and 2D Cellular Automata explorer”. On-line documentation: http://www.mirwoj.opus.chelm.pl/ca/mjcell/mjcell.html.
15. ELLIOTT, J. M. G. “Cellsprings” On-line document: http://jmge.net/java/csprings/doc/
16. CHOU, H.; HUANG, W.; REGGIA, J. "The Trend Cellular Automata Environment for Artificial Life". Accepted for publication in *Transactions of the Society for Modeling and Simulation International*. 2002.
17. MORENO, N.; ABLAN, M.; TONELLA, G. “Spasim: A Software to Simulate Cellular Automata Models”. Proceedings of *IEMSS 2002 conference on Integrated Assessment and Decision Support.* Lugano, Switzerland. 2002.
18. RUCKER, R.; OSTROV, D. “Continuous-Valued Cellular Automata for Non-Linear Wave Equations”, *Complex Systems* 10 (1196) 91-119, 1997.
19. GUARRACI, B.; POTTER, J.; RITZ, B.; WINTER, D. “Modeling Grid Cellular Automata in 3-D”, *Technical Report, Department of Mathematics/Computer Science, Salisbury State University, Salisbury*, MD 21801 http://faculty.ssu.edu/~mjbardze/dan_vrml/index.html.
20. FREIWALD, U.; WEIMAR, J. “JCASim – a Java system for Simulating Cellular Automata”, *Theoretical and Practical Issues on Cellular Automata (ACRI 2000)*, S. Bandini and T.Worsch (eds.), Springer Verlag, London, 2001.
21. WAINER, G.; GIAMBIASI, N. "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation.". *Simulation*; Volume 76, Number 1. January 2001.
22. ZEIGLER, B.; MOON, Y.; KIM, D.; BALL, G. "The DEVS environment for high-performance modeling and simulation". *IEEE Computational Science and Engineering* , Vol. 4, No. 3. 1997.
23. NUTARO, J. "*Adevs*: User manual and API documentation". On-line document, http://www.ece.arizona.edu/~nutaro/adevs-docs/index.html.
24. DE LARA, J.; VANGHELUWE, H. “AToM3: A Tool for Multi-Formalism and Meta-Modeling”, Proceedings of *European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2002.
25. PRAEHOFER, H.; SAMETINGER, J. STRITZINGER, A. “Concepts and Architecture of a Simulation Framework Based on the JavaBean Component Model” Proceedings of *WEBSIM99, 1999 International Conference On Web-Based Modeling & Simulation,* San Francisco, California. 1999.
26. FILIPPI, J.B. BERNARDI, F, DELHOM, M. "The JDEVS environmental modeling and simulation environment". Proceedings of *IEMSS 2002 conference on Integrated Assessment and Decision Support*. Lugano, Switzerland. 2002.

27. ZEIGLER, B.; MOON, Y.; KIM, D.; KIM, J. "DEVS-C++: A High Performance Modeling and Simulation Environment". Proceedings of *29th Annual Hawaii International Conference on System Sciences (HICSS-29)*, Maui, Hawaii. 1996.
28. SARJOUGHIAN, H.; ZEIGLER, B. "DEVSJava: Basis for a DEVS-based Collaborative M&S Environment". Proceedings of *1998 International Conference on Web-Based Modeling and Simulation*. San Diego, California. 1998.
29. KIM, Y.; KIM, T. ,"Optimization of Model Execution Time in the DEVSim++ Environment". Proceedings of *European Simulation Symposium.* Passau, Germany. 1997.
30. ZEIGLER, B., HALL, S.; SARJOUGHIAN, H. "Exploiting HLA and DEVS To Promote Interoperability and Reuse in Lockheed's Corporate Environment". *Simulation* V. 735, pp. 288-295, 1999.
31. DAHMANN, J. "High Level Architecture for Simulation". *Proceedings of the 1st International Workshop on Distributed Interactive Simulation and Real-Time Applications (DIS-RT '97).* Eilat, Israel. 1997.
32. ZEIGLER, B.; HALL, S.; SARJOUGHIAN, H. "Exploiting HLA and DEVS to promote interoperability and reuse in Lockheed's corporate environment". *Simulation.* Vol. 73, No. 5. 228-295. November 1999.
33. AMES, A.; NADEAU, D.; MORELAND, J. "VRML 2.0 Source" (Second Edition). *John Wiley & Sons, Inc*. 1997.
34. ROEHL, B. ; COUCH, J. "Late Night VRML 2.0 with Java". *Ziff-Davis Press*. 1997.
35. CHOW, A.; KIM, D.; ZEIGLER, B. "Abstract Simulator for the parallel DEVS formalism". *Proceedings of AI, Simulation, and Planning in High Autonomy Systems.* IEEE Press, 157-163. Dec., 1994
36. MARTIN, D.; MCBRAYER, T.; RADHAKRISHNAN, R.; WILSEY, P. "TimeWarp Parallel Discrete Event Simulator". *Technical Report. Computer Architecture Design Laboratory, University of Cincinnati*. December 1997.
37. SZULSZTEIN, E.; WAINER, G. "Implementing a flexible environment for parallel simulation " (in Spanish). In *Proceedings of 28th Argentine Conference on Informatics and Operations Research (JAIIO).* Buenos Aires, Argentina. 1999.
38. TROCCOLI, A.; WAINER, G. "Performance results of parallel Cell-DEVS execution". In *Proceedings of 2001 Summer Computer Simulation Conference*. Orlando, FL. U.S.A. 2001.
39. TROCCOLI, A.; WAINER, G. "Implementing Parallel Cell-DEVS". In *Proceedings of Annual Simulation Symposium.* Orlando, FL. U.S.A. 2003.

40. ZEIGLER, B.; SONG, H.; KIM, T.; PRAEHOFER, H. "DEVS Framework for Modelling, Simulation, Analysis, and Design of Hybrid Systems". In *Proceedings of Hybrid Systems IV*. Lecture Notes in Computer Science 999, Springer. 1995.

Gabriel A. Wainer received the M.Sc. (1993) and the Ph.D. degrees (1998, with highest honours) at the Universidad de Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. He is Assistant Professor at the Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada). He was Assistant Professor at the Computer Sciences Dept. of the Universidad de Buenos Aires, Argentina, and a visiting research scholar at the Arizona Center of Integrated Modelling and Simulation (ACIMS, University of Arizona). He has published more than 70 articles in the field of operating systems, real-time systems and Discrete-Event simulation. He is author of a book on real-time systems and another on Discrete-Event simulation. He has been the PI of several research projects, and participated in different international research programs. Prof. Wainer was a member of the Board of Directors of The Society for Computer Simulation International (SCS). He is the coordinator of a group on DEVS standardization. He is Associate Editor of the Transactions of the SCS. He is also a Co-associate Director of the Ottawa Center of The McLeod Institute of Simulation Sciences.

Wenhong Chen received a B. Eng. in Aircraft Engineering from Northwestern Polytechnic University, China (1988) and a M. Eng. in Robotics from Tianjin University, China (1994). He was a researcher/lecturer in the Machine Intelligence Lab in the Tianjin Institute of Technology, and an R&D Engineer in Motorola Greater China. In 2003 he received a M. Sc. in Information Systems Sciences from the Department of Systems and Computer Engineering, Carleton University.