Abstract

Simulation is becoming increasingly important in the analysis and design of complex systems. CD++ is a modelling tool for simulation of complex physical systems, which can be used to simulate a variety of models. In order to enhance the usability of this tool, this thesis introduces many facilities, which are organized as a simulation client. The simulation client provides users with the ability to create a simulation model, send the simulation model to a remote CD++ server for execution and visualize the results with easy-to-use 2D and 3D interfaces locally. This client component also can support multi-view visualization and run several different models simultaneously. The sophisticated user graphical interfaces in these facilities improve the analysis of simulation models. The simulation server can now be used by various users around the world to perform multi-observer simulation using remote execution of the models.

Acknowledgement

Firstly, sincere thanks go to my academic supervisor Professor Wainer, G. He provided me with a substantial amount of help and suggestions throughout my work.

Special thanks also go to the students in the RADS laboratory, Department of Systems and Computer Engineering, Carleton University, for many helpful discussions.

Lastly, and mostly, sincere gratitude goes to my wife for supporting me throughout my graduate studies.

I am also grateful to my friends for helping me with their encouragement and suggestion.

Table of Contents

List of Figures

Chapter 1: Introduction

Simulation is becoming increasingly important in the analysis and design of complex systems. Scientists and engineers have long used models to better understand the systems that they are studying: models have been used for analysis and quantification, design, prediction and the understanding of different complex phenomena. The simulation process begins with a practical problem needed solving or understanding. It might be the case of a transportation company trying to develop a new strategy for cargo and truck usage before putting it into effect, or a chemist trying to understand a complex chemical reaction taking years to complete.

In most cases, these models can be defined as mathematical representations, and can be analyzed using mathematical techniques. However, at times these methods were also proved infeasible in studying some recent complex artificial systems, such as traffic controllers, digital systems, automated factories, robots, etc. Likewise, the complexity of the natural systems under analysis is growing, making it impossible to use analytical methods.

The appearance of digital computers provided scientists and engineers with alternative methods of analysis. Since the early days of computing, they have started to translate their analytical models into computer simulations. A simulation process starts with the observation of a real system**.** Entities in the system are identified, and an abstract

representation (a model) is created with some modeling technique. Computer simulation enables scientists and engineers to experiment with "virtual" environments, elevating the analysis of natural and artificial systems to a new level of detail unknown in earlier stages of scientific development, and providing great help in the design and analysis of complex systems. Simulated models also can be used for training and many other purposes because they provide cost-effective and risk-free solutions.

At present, there are a large number of modeling and simulation techniques, and various types of simulation tools have been developed to deal with complex systems and the interactions among their constituent parts. A formalism that is gaining popularity in recent years is called DEVS (Discrete Event Systems Specification) [41, 42]. DEVS provides a framework for the construction of discrete-event hierarchical models in a modular manner, which allows pre-defined models to be reused in new models to reduce development time. In DEVS, basic models (atomic models) are specified as black boxes with a state and duration for that state. When the duration time for the state expires, an output event is sent, an internal transition takes place and the model changes its current state. A change of state also can occur when an external event is received. An atomic model is defined by a set of states of the model, the internal and external transition functions, the output function and the state duration function. Several DEVS models can be integrated together to form a hierarchical structural model (coupled model).

Cell-DEVS [32] extends the DEVS formalism and allows simulating discrete-event cell spaces. This approach extends traditional Cellular Automata (CA) [29], defined as a lattice of cells updated synchronously and simultaneously. Each cell in a CA holds a state

variable and can be in one of a finite number of possible states. The new state of a cell is computed based on the current state of the cell and the states of its 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.

The CD++ tool [33] can be used to simulate DEVS and Cell-DEVS models. It has been used to create a variety of models in many different areas: biology (watersheds, fire spread, ant colonies), physics (crystal growth, lattice gases, heat diffusion), chemistry (solution diffusion in moving fluids), and artificial systems (autonomous robots, heat seekers, urban traffic) [2, 3, 30].

While executing simulation models for these complex applications, it was found that the computing power provided by personal computers was not enough when the model size increased. Despite this fact, end users might not have access to high performance computing resources, or they might prefer to use personal computers with standard software packages for analysis and development. A solution to these problems is to enable the users to execute the simulation models in remote high-performance computers, while using their personal computers for development and analysis. In these cases, client/server architectures provide a very good solution for the remote execution of the model. The simulation software can be designed as a server and execute many simulation models simultaneously, and the users can communicate with this server through a network to request simulation services. The CD++ simulator was recently modified following these ideas and transformed into a simulation server. It can run on high performance computers, accepting the requests from the users and executing many

simulation models simultaneously. However, although the CD++ simulator can be used as a server, until recently it has only been used as an application running on a local machine because of the lack of a client application.

Another problem of using CD++ for complex system analysis was the lack of adequate visualization mechanisms, that is, any means by which the users use simulation results to construct useful 2D or 3D images. Visualization tools are crucial in helping to better understand 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 color classifications, shapes, or symbols [20, 22]. As illustrated in Figure 1.1 [10], visualization has become an integral part in modeling and simulation. Effective simulation tools must include good corresponding visualization tools. The user can do the observation and analysis of the real system or just modify the model again if he finds the result of the simulation is not correct.

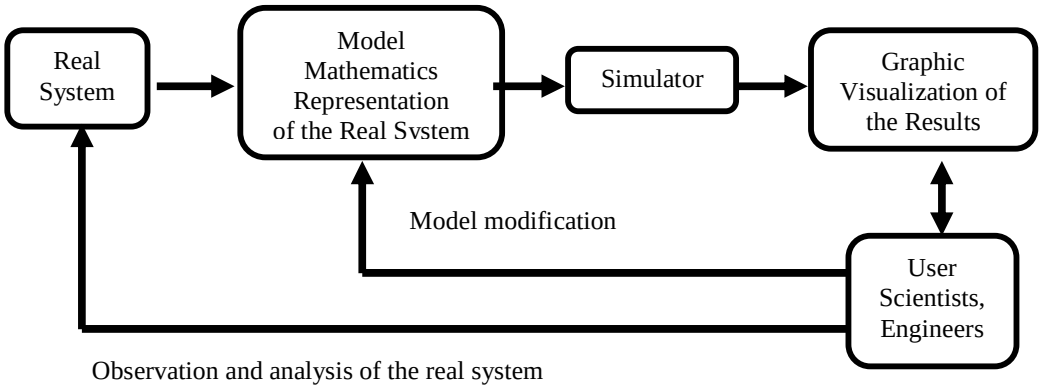


Figure 1.1 Flow-chart of a typical computer simulation

The goal of visualization is to provide a deeper understanding of the real 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. Therefore, visualization has been now considered as essential to theory, modeling and experimentation.

A useful visualization system should attempt to meet the following goals [36]:

1. Display as much information as possible on the screen.
2. Show relationships among different components.
3. Show necessary parts of all the information on the screen: sometimes, attempting to display all the details to the users is not necessary or difficult, as there may be too many details. Moreover, it is hard for the user to keep these details in mind at one time. Visualization tools attempt to address this by allowing the user to only display the important or necessary parts of the results.
4. Show the results with the same sequence as the simulation does: stop and resume the display, or go to any point of the display: the users, seeking to better understand a system, also want to visualize the simulation behavior by animating the progress of the simulation processes. The users may also want to check the simulation by moving the logical processes in any direction, and beginning the continuous display at any logical point.

This thesis challenges the issues in CD++ simulator mentioned above by providing the users with a series of tools, including a CD++ modeler, an interface to connect the CD++

simulator on a remote server, and sophisticated visualization facilities. The following visualization facilities are provided to reach the above visualization goals [36]:

1. *Display Methods*: they allow the users to visualize the results at any simulation time and in any direction of the simulation process. It may be forward or backward, and they allow the users to visualize the results continuously or step by step, or stop at a special time for detailed examination of the result.
2. *Context and Detail*: they allow the users to focus on the details of a particular part of the result because sometimes the users only want to check an interesting part of the whole result.
3. *Navigation:* when the model or result is too large for the screen, they allow the user to check the results more efficiently by moving through the scene. The display should change continuously, so the user can move to any position and orientation of the scene for detailed examination of the interesting part.
4. *Multi-View*: they allow the users to visualize the results with multiple views in several different windows with different viewpoints at the same time, in order for them to investigate the simulation results from several different views.
5. *Remote multi-observer*: many users can participate, and the simulation results can be distributed among many users as needed. Sometimes, the users in different places with different technical backgrounds need to participate in the analysis of the simulation result.
6. *Access to a Remote Server*: the users can send model files to a remote server, receive and visualize the results locally. In this way, if the simulation needs computing power

that normal personal computers do not provide, the software running on a remote computer can be used.

The thesis work started by upgrading significantly an existing 2D GUI (Graphical User Interface), the CD++ Modeler [38]. Various checking rules were introduced to validate the new model design to ensure the design conforms to the DEVS formalism. New features were added, and the 2D visualization GUIs were expanded to different types of result files (including 3D result files).

A sophisticated 3D GUI was developed using VRML (Virtual Reality Makeup Language) [4, 19] and Java. The users can select different geometries to represent the results, assign different color classifications and navigate in the field of visualization. They also can edit all the nodes in the scene as a whole, or even individual layers and nodes in the scene to better understand the result. A multi-view GUI is provided with the goal of displaying multiple views of the result. Different viewing areas can be selected, so different areas of the same result file can be displayed and visualized simultaneously.

Finally, the CD++ simulator was transformed into a client/server engine, able to provide visual simulation results and remote access to a high performance DEVS simulation server. The simulation server can receive model specifications from the clients, and send 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 the models to a remote CD++ server executing in a high

performance platform.

The client Interface enables the users to access a remote simulation server. It can send model files to the remote CD++ server, receive the result on the local machine, and then change the result format of the result stream to be visualized with the new visualization facilities. The design of this interface enables easy extension of this simulation system to a multi-observer remote simulation system in various kinds of environments. Using one client as the interface client to the server, many other clients can work together to act as multi-observers. Other clients can send models to the server through this interface client, which then sends models to the server, receives the results and distributes the results among other clients as required. In addition, all the joint clients can communicate with each other through this interface client.

The provision of these tools enables to have a fully functional modeling and simulation environment for the CD++ simulator. It enables model definition with graph-based notations, 2D and 3D result visualization, and remote simulation execution. In DEVS-based environments, models are completely independent from the simulation engines, and the simulators can be exchanged without doing any modifications to existing models. This feature facilitates all these tools to be organized together as a simulation client to be applied in the CD++ simulation environment.

The contributions of this thesis are summarized as follows:

1. Contributions to the knowledge

   This thesis developed methods to map visualization entities with the simulation model. Different algorithms were developed to directly manipulate the visualization entities, update and navigate the simulation results. Algorithms to transform the 2D models to 3D models were also developed. The mechanisms for remote execution of simulation models were defined

2. Practical contribution

   This thesis introduced a variety of tools to work as a full-functional simulation client. It extended the existing tool, CD++ Modeler with new methods to check the model design to ensure that it conforms to the DEVS formalism. 2D and 3D visualization GUIs was developed for simulation result visualization. Client Interface was developed for the access to a remote simulation server.

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 the design and implementation of this remote simulation environment. Finally, we show several examples presenting the visualization facilities.

The thesis is organized as follows.

- Chapter 2 introduction to the DEVS formalisms, the existing facilities of the CD++ tool, and a brief introduction to VRML. This chapter also gives a short review of the related research, and the main research contributions of this thesis are illustrated.
- Chapter 3 describes the general design aspects of the client.
- Chapter 4 describes the features of the client.
- Chapter 5 describes the details of the design and implementation of the client, and explores the design goals.
- Chapter 6 presents several simulations for the new client.
- Chapter 7 presents conclusions, suggests directions for future work.

Chapter 2: Background

The DEVS formalism provides a framework for the construction of hierarchical modular models [41]. This chapter introduces the DEVS formalism and we describe atomic models, coupled models and Cell-DEVS models. Then, we introduce the CD++ toolkit, and how to define DEVS and Cell-DEVS models. Several examples are introduced for better understanding of the CD++ tool. This chapter also briefly introduces the main features of the Virtual Reality Modeling Language (VRML), and the reasons why we choose it as our development tool. This chapter also gives a review of related research on DEVS tools, and visualization facilities for cellular models. Finally, the main research contributions of this thesis are presented.

2.1 The DEVS Formalism

The DEVS (Discrete Events Systems Specifications) formalism [41] was originally defined in the ‘70s as a discrete-event modeling specification mechanism. It is a theoretical approach that allows the definitions of hierarchical modular models that can be easily reused. A real system modeled with DEVS is composed 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, states, inputs, outputs, and functions to determine the next states and outputs. Tested models can be integrated into a modeling hierarchy, allowing model reuse, reducing testing time, and improving productivity.

Atomic Models

A DEVS atomic model is described formally as:

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

Where

X the set of input events set;

Y the set of output events;

S the state set;

δ_{int} internal transitions;

δ_{ext} external transitions;

λ the output function;

D the duration function.

The interface of the model consists of input (*X*) and output (*Y*) ports to interact with other models. Each state in the model has a corresponding lifetime defined by the duration function. The internal transition is activated to produce an external state transition after the model spends the corresponding lifetime on the present state. Before changing to the new state, the model generates the outputs using the current state values through the output ports. External input events from other models arrive through its input ports, and trigger an external transition specified by the external transition function. This function computes the new state of the model using the present state, the input values, and the time elapsed since the last event. The transition generates an internal state change after the results are sent out through the output ports. Every time a transition function is activated, a new lifetime is associated with the new state.

Coupled Models

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

$$CM = \langle X, Y, D, \{Mi\}, \{Ii\}, \{Zij\}, select \rangle$$

Where

X the set of input events set;

Y the set of output events;

D an index of components;

Mi a basic DEVS model (atomic or coupled model), $\forall\ i \in D$;

Ii the set of influences of model i (the models that can be influenced by outputs of model i), $j \in Ii$; and

Zij the model i to model j translation function;

select the tie-breaking selector.

Each coupled model consists of a set of basic components (atomic or coupled model), which interact with each other through the model's interface. Each model *i* has its own set of influencees $\mathbf{I_i}$, defined as the models to which its output values must be sent. For each influencee j in model i, a translation function is defined as $\mathbf{Z_{ij}}$. It defines how the outputs of model M_i will be converted into inputs for model M_j. When two sub-models have simultaneous events, the *Select* function defines which one should be activated first.

Cell-DEVS

The Cell-DEVS formalism extends the basic behavior of DEVS models to allow the implementation of cellular models with timing delays [32]. A Cell-DEVS model can be

defined as an infinite n-dimensional lattice of cells. The state value of each cell in these spaces is updated according to a local rule, which considers its own state and those of a finite set of its nearby cells (called its neighborhood). Each cell is defined as an atomic model with timing delays, and can be integrated to a coupled model to represent a cell space.

Cell-DEVS defines cells as atomic models. A Cell-DEVS atomic model is defined by:

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

where

X	a set of external input events;
Y	a set of external output events;
I	the set of states for the input events;
S	the set of sequential states for the cell;
θ	the cell state definition;
d	the delay for the cell;
δ_{int}	the internal transition function;
δ_{ext}	the external transition function;
τ	the local computation function;
λ	the output function; and
D	the state's duration function.

A cell uses the input values I to obtain its next state by executing the local computation function τ. A delay function is associated with each cell, delaying the computed result to be sent to the neighbor cells. There are two types of delays: inertial and transport delay.

For the transport delay, the next value will be added to a queue sorted by output time, therefore, the results can be stored until they have been sent out. On the contrary, inertial delay uses 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, the whole cell space can be constructed by building a coupled Cell-DEVS model, defined by:

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

where

Xlist the input coupling list;

Ylist the output coupling list;

I the definition of the interface for the modular model;

X the set of external input events;

Y the set of external output events;

{m, n} the dimension of the cell space;

N the neighborhood set;

C the cell space, where $C = \{C_{ij} / i \in [1,m], j \in [1,n]\}$, C_{ij} is a Cell-DEVS atomic model;

B the set of border cells;

Z the translation function; and

Coupled models are built as an array of atomic cells. $\mathbf{X}_{list}$ and $\mathbf{Y}_{list}$ are the input/output coupling lists, and 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 scope. The cell space C, the set of border cells B, and the translation function Z define the cell space. Each cell is connected to the cells in its neighborhood. Since a cell space is finite, the cells on the borders should have a different neighborhood than the rest of the space. They can be "wrapped", that is, cells on the border are connected to the cells in the opposite one. Otherwise, the border cells need to be provided with a behavior different from those of the rest of the model. Finally, the Z function defines the internal and external coupling of the 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.

2.2 The CD++ Toolkit

The CD++ environment [33] was built to implement DEVS and Cell-DEVS theories. The toolkit includes a set of independent software pieces running in different platforms to facilitate modeling and simulation.

The tool allows defining models according to the specifications introduced in the previous section. The models are built as a class hierarchy, and each of them is related with a specific simulation entity, which is activated whenever the model needs to be executed. New atomic models can be incorporated into this class hierarchy by writing DEVS models in C++. They can be defined by overloading the basic methods representing DEVS specifications: external transitions, internal transitions and output

functions. After an atomic model is tested, it can be stored in a model database and re-used to build a multi-component model (coupled model).

In CD++, coupled models are defined using a specification language specially defined with this purpose. The language was built by following the formal definitions for DEVS coupled models. The language is illustrated in the following sections.

2.2.1 Definition of Coupled models in CD++

Model files are used to define coupled and Cell-DEVS models within the CD++ tool. A model file consists of a set of groups and definition clauses within these groups [34]. A group is identified by its name between two square brackets at the beginning of the definition. Every model file must have a **top** group, which identifies the top level coupled model. Each coupled model, is defined using four different parameters:

Components with the following syntax**:**

```
component : name1[@coupled_model1][name2[@atomic_class2] …
```

This construction lists the component models of the coupled model under consideration. A coupled model can have atomic models or other coupled model as its components. For atomic components, an instance name and a class name must be specified. This allows a coupled model to use more than one instance of the same atomic class. For coupled models, only the model name must be specified. This model name must be defined as another group within the same file.

Out, with the following syntax**:**

```
out : portname1, portname2 …
```

This construction represents the model's output ports. This clause is optional

In, with the following syntax**:**

```
in : portname1, portname2 …
```

This construction represents the model's input ports. This clause is also optional

Link, with the following syntax:

```
Link: source_port[@model]  destination_port[@model]
```

This construction represents the links between components in the coupled model. If the name of the model is omitted, it is assumed that the port belongs to the coupled model being defined.

A coupled model example is shown in the following figure 2.1:

```
[top]
components : queue@Queue processor@CPU
transducer@Transducer generator@Generator
Out : throughput
Out : cpuusage
Link : out@generator arrived@transducer
Link : out@generator in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor solved@transducer
Link : throughput@transducer throughput
Link : cpuusage@transducer cpuusage
```

Figure 2.1 A Coupled-DEVS specification in CD++

As we can see in Figure 2.1, there are four basic DEVS models: *queue, processor, generator and transducer*, each of which is an instance of an existing model. For

instance, *queue* is an instance of the existing *Queue* atomic model. Links are established to connect the components and their influencees. For instance, the output port *out* in the *generator* model is connected to the *in* port of the *queue* model. Two output ports, *throughput* and *cpuusage*, are connected to the output ports of *transducer*.

CD++ also can 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 timing delays. In addition, it allows defining the size of the cell space, the border and the initial state of each cell. This language was defined by 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 these 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.

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 [35] for details). The user only needs to define the τ function (defined by the local transition) and the delay (defined by *delay* and the delay values in each rule). For instance, Figure 2.2 shows an example for 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*, and specify the size of the cell space (in this example, m = 20, n = 40). The neighborhood set N is defined by the lines starting with the *neighbors* keyword. The border (B) can be wrapped or no 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.

```
[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 2.2 A Cell-DEVS specification in CD++

In this example, the first several lines define the dimension parameters (20 × 40) of the cell space. Then the kind of the delay and the shape of the neighborhood are included. The border is defined as wrapped, so the cells in the border can use the same neighborhood and computing function as the others. Finally, the local computing function is included. 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 spread to the neighbors after 150 ms. As we can see, cells evolve using a discrete-event approach.

The CD++ simulator is message-driven, and each message represents an event being executed in a model with an associated timestamp. The simulation outputs can be recorded into a result file, which keeps a record of all the messages sent between components. Each line of the file shows the name of the component that received the message, the message type, the time of the event, the sender and the receiver. Using this file as input, we can reproduce the state of each model, and we can use it to analyze model outputs. For instance, figure 2.3 shows a fragment of a result file with different messages evolving in the DEVS coupled model defined in Figure 2.1 (a *CPU* connected to a *queue*, and a *transducer* that computes performance metrics).

```
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 2.3 A fragment of an example result file

There are four kinds of messages: X (inputs), Y (outputs), * (internal transitions) and D (done messages). In the second message of Figure 2.3, a sub-model called *processor* is activated due to an internal transition. The model generates an output (the value `10.00000`, which is sent through the *out* port), and then 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 2 different sub-models (*queue* and *transducer*).

In order to better visualize the execution of Cell-DEVS models, the *drawlog* utility [34] permits to view the state of the complete cellular model after each simulation cycle. *drawlog* utility is an existing tool of the CD++ to parse the messages in the result file. Using the result file as the input, *drawlog* parses the Y messages to get the state of every cell in the model, and stores all the cell states in another output file with different format. The output format of *drawlog* depends on the number of the dimensions of the cellular model, which can be two, three or more dimensional. Figure 2.4 shows a fragment of the output file generated by *drawlog* utility for a two-dimensional model of size 10 x 10.

```
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 2.4 A fragment of an output file generated by *drawlog* utility

This fragment shows the results of a heat diffusion model in a surface. The cells at (2, 2) and (5, 5) are connected to a heating generator (now they are receiving 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. The

initial temperature of all the cells is 24.0 °C. A cell's temperature value is obtained by computing the average of the temperature values of the cell's neighborhood.

For the models with three or more dimensions, the results can be shown as matrixes, each of them representing a 2-dimensional plane in the model, and looks like the one in Figure 2.4. For instance, in a 3D dimensional space, the first plane corresponds to (x, y, 0), the second one to (x, y, 1), etc. Figure 2.5 shows a model for the 3D simulation of the 'Life' game [18] with the original rules proposed by Conway. In this simple model, there are cells, which 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 2.5 A fragment of a result file of a 3D model

2.3 Virtual Reality Modeling Language (VRML)

VRML is a Web-based graphics language for building 3D models. VRML allows the users to interact with a scene through a variety of methods, such as viewpoints, movement, and rotation. It is an ISO standard designed for use on the World Wide Web. VRML is a scene description language. Though VRML is a computer language, it is not a programming language. VRML files are simple ASCII text files, which are not compiled, but parsed by a VRML interpreter. These interpreting programs are often called VRML browsers.

VRML worlds are created using a scene-graph structure. Scene graphs are simply a hierarchical decomposition of components that will be rendered in a **scene**. Scene graphs are comprised of various groups of **nodes**, which together form a virtual world. These nodes are responsible for displaying shapes, interaction, and movement through the world. VRML worlds can be viewed with any VRML-capable browser such as Cosmo Player. Using Java and EAI [23], users can have full control of VRML World to create a dynamic 3D VRML World. Therefore, VRML is a file format for describing interactive 3D objects and worlds.

VRML has been successfully used in a variety of application areas, such as, engineering and scientific visualization, multimedia, entertainment, education, and shared virtual worlds. We decided to use VRML as the development tool, because of the following attributes:

1. VRML is the ISO standard designed for use on the World Wide Web, which makes it easy to be used on the Internet or local clients.
2. It is platform independent, and it can be used in various operating systems and hardware configurations.
3. It is scalable, enabling nodes to be dynamically added to or removed, thus building arbitrarily large dynamic 3D worlds.
4. It is extensible: a user can introduce new node types.
5. It is reusable: a previously saved VRML world can be used in a new VRML world.
6. It is event enabled: the nodes can respond to the users' action, and events on one node can be spread to other nodes in the world.

These attributes make the VRML as a perfect tool to develop simulation visualization software to be used on the Internet, Intranet, and local clients. However, VRML has some disadvantages, such as

1. Relatively slow rendering speed because the Java program controls the VRML world through EAI.
2. A VRML world is controlled with an applet, which not allowed accessing local file and making connection to other computers.

Therefore, effective algorithms should be developed to facilitate the scene rendering, and a standalone application is needed to communicate with the remote computers.

For more information about VRML, how to create a VRML scene and interact with Java programs, please refer to on-line report [6].

2.4 Related Work

At present, a number of efforts have been devoted to develop DEVS models and cellular models, but none of them meets our requirements. We intend to provide tools for the users to build the models of complex physical systems and visualize the results locally with basic workstations, while executing the models remotely in a high performance platform anywhere in the world.

A number of efforts also have been devoted to build tools for modeling CA. Some of them provide good visualization facilities while others enable remote execution of the models. Some of the existing tools are described following.

- **MJCell** [40] is a Java applet to simulate CA. Its main purpose is to explore 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. The users can select one of the eleven available families of rules, and the desired rule. They also can change the size of the model, initiate and run the model. It includes advanced editing features and many analysis tools.
- **Cellsprings** [13] is a powerful 2D CA Java applet. It comes in two editions, Cellsprings/Web, a Java applet, and Cellsprings/DT, a Java desktop application. More

than seventy CA rules are predefined, and the users can define, run, and save their own arbitrary rules. The applet version saves the new rules in the server, so they can be accessed by other users. The users also can change the size of the model and the color palette map, specify some characters of the model, then initiate and run the model.

- **Trend** [9], is a general-purpose 1D or 2D CA simulation system. It is very flexible about the space sizes, cell and neighborhood structures and cellular automata rules. It also has a smart backtracking feature that simplifies rule set development by allowing users to return to previous stages of the simulation.
- **SpaSim** [24] allows the user to build, simulate and perform spatial and spatial-temporal analysis on the same environment using a friendly user interface. To visualize the 3D model, it includes a dynamic window dialog containing several tabs, one for each of the automata layers. For each layer, the user selects the time and color palette to be used, and the layers can be exported, imported or saved as independent layers. Therefore, it actually is a 2D visualization tool.

Some of the existing tools enable 3D visualization of the executing cells. Some of them are described following:

- **Capow** [27], is a program for evolving 1D and 2D CA. The user can control the simulation with parameters, control the visualization with color classifications, and select the type of view and 3D view details. However, for 3D visualization, it needs to

create a VRML output file used to visualize the result. In addition, it only displays the surface of 3D images, so the user cannot see the inside states.

- **PascGalois** [17] can produce innovative 3D visualization of 2D CA. It lays different results over together, or changes the 2D raster images to 3D (rolling a 2D (graph) raster images to implement the idea in 3D).
- **CASim** [14] 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, color classifications and icons. After initializing some selected cells, the user can run the model.
- Different 3D tools have been applied to simulate 3D versions of the 'Life' game. For instance, **3-D Life Visualization** [31] simulates a 3D-life game developed with OpenGL. A 3-D Life is played on a three-dimensional grid of cubic cells. It tries to visualize the development of the cells from generation to generation with appropriate graphics techniques. **The Game of Three Dimensional Life** [5], is also sophisticated software to simulate the 3D-life game developed with 2D images. The user can design the game, and change the color classifications of the sides of the boxes. The user also can rotate the image, and see the game in different viewpoints. In spite of these, the images are actually 2D, so no navigation can be made through the images. In **3D Cellular Automata** [1], the user can select the initial state, grid size, delay between generations, and growing algorithm. The image is actually 2D, and no navigation can be made through the graphics.

These tools do not meet our goals for visualization and remote execution, and they have problems related with the definition of the cellular models. In [35], it was demonstrated 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, in several cases most cells of the automaton do not need to be updated in each time step. These "quiescent" states allow defining modifications in which the automaton advances using instantaneous events that can occur at unpredictable times.

As mentioned in section 1, using DEVS as a basic formalism, we can improve performance execution. As DEVS and Cell-DEVS are discrete event formalisms, they provide higher precision and speedup in the simulations than the discrete time approaches used by CA. In [43], the authors showed that DEVS combined with parallel simulation techniques can produce speedups of up to 1000 times. In [35], the authors showed that Cell-DEVS also provides 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 if any of the existing tools completely satisfy our requirements. At present, many tools have implemented DEVS formalism, but none of them is able to meet our goals. Most of them do not provide facilities for the execution of cellular models. In addition, some of them do not

provide visualization facilities and remote execution service. Although they provide much flexibility for the users to develop their own models, they are not easy to use:

- **ADEVS** [25] provides a C++ class library based on the DEVS formalisms. No practical distributed environment, and 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 model. The users should decide how to output the result files, and design the corresponding visualization tools.
- **Python DEVS** [11] uses the ATOM3-DEVS tool to construct DEVS models. The models are represented as a graph that is 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 a 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 tools.
- **Neuro-DEVS** [15] is an Object Oriented Modeling and Simulation environment that can be used to model a system whose behavior is unpredictable, and its knowledge is collected in empirical data. It introduces some items in the model description, such as, a learning function, but it does not introduce remote execution or advanced visualization tools.
- **DEVS/C++** [44] 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. No client/server simulation environment and

no result visualization are considered.

- **GALATEA** [12] is also a DEVS-based simulation platform that offers a language to model multi-agent systems. It describes a real system as interacting agents. The model program describes all the entities in the system, the propagation of events in the system and the relationship between agents. The simulator can trigger the events, and coordinate the execution of the pieces of codes in the related agents. The users analyze their systems and identify the entities and their relations, then use the provided specific language to build the model as multi-agent systems. The users should design their own visualization tools.

Other existing DEVS modeling tools provide some basic visualization tools:

- **SimBeams** [26] 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 images.
- **JDEVS** [16] 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.

Two of the existing DEVS environments are suitable to meet our goals. Unfortunately, none of these environments is able to run Cell-DEVS models in remote environments, and visualize 3D models. Some of them were provided with extensions for visualization of particular problems, but no generic visualization facilities are provided.

- **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.
- **DEVS/HLA** [45] is based on the High Level Architecture (HLA) 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. The 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 can be integrated with powerful visual displays.

2.5 Research description

As we indicated before, simulation is becoming increasingly important in the analysis and design of natural and artificial systems. DEVS is a formalism that is gaining popularity in recent years, and it has found applications in many areas. However, as we can see from the related work, no practical tool for 3D visualization has been developed in Cell-DEVS simulation. In addition, no practical integrated distributed DEVS simulation environment has been developed in DEVS simulation.

The contribution of this thesis is to introduce a practical integrated remote DEVS simulation and visualization environment for the users with functions in every aspect of the modeling and simulation, which includes:

1. A DEVS Modeler [38] to build DEVS models using a graph-based notation. New methods were added to this existing utility to check the model design to ensure that it conforms to the DEVS formalism.
2. 2D Cell-DEVS visualization tools
3. 3D Cell-DEVS visualization tools
4. An interface with the simulation server, enabling users to remotely invoke the CD++ simulation engine in server mode

With sophisticated user graphical interfaces in above four components, this simulation environment can be used by various users with varied expertise around the world. In

addition, with the design of this simulation environment, the users even can set up a remote simulation environment in their own computing environment, such as, the Intranet within their company.

The users may not be familiar with the simulation and visualization theory, but they can rapidly obtain the results and visualizations to assist in analysis for scientific and technical purposes. Nevertheless, with this simulation environment, they can build models locally, send the models to a remote CD++ server, and receive the results locally. Then the 2D and 3D visualization GUIs can be used to visualize the results. These visualization tools enable the user to navigate in the visualization with many ways, select the shape and color palette of the cells, and edit the cell matrix to check the results more effectively.

Chapter 3: Design of a Client/Server Simulation Platform for CD++

According to the research description information in the previous chapter, we decided to develop a set of tools for the CD++ simulator to facilitate the definition of DEVS simulation models, visualization of the results, and the access to a remote simulation server for execution. These toolkits provide easy-to-use graphical user interfaces that insulate the users from the requirements of knowing the simulation implementation details and much programming knowledge. The goal is to develop a full-functional client/server simulation environment for the users with all these toolkits.

As indicated before, the CD++ simulator was extended to run as a stand-alone application or as a server. This chapter will describe the main components and their inter-relationship in our simulation environment when CD++ simulator runs as an application and a server respectively. Finally, we will discuss the general design guidelines used in the design of our client/server simulation environment and user interfaces.

3.1 CD++ simulator works as a stand-alone application

When CD++ simulator works as an application (stand-alone mode), it runs on a local machine. The users build the model first, and then activate the CD++ simulator to execute the model. After the execution, a result file will be generated, and then the users launch the *drawlog* facility to change the result stream to another type of result stream,

which can be visualized with the visualization tools (2D and 3D visualization GUIs).

According to above description, when CD++ simulator works as an application, the simulation procedure involves following four steps:

1. The user builds model
2. The user activates CD++ simulator to execute the model
3. The user changes the result format
4. The user visualizes the result with 2D and 3D GUI

The components in the CD++ simulation environment and their relationships can be illustrated with the following figure 3.1.

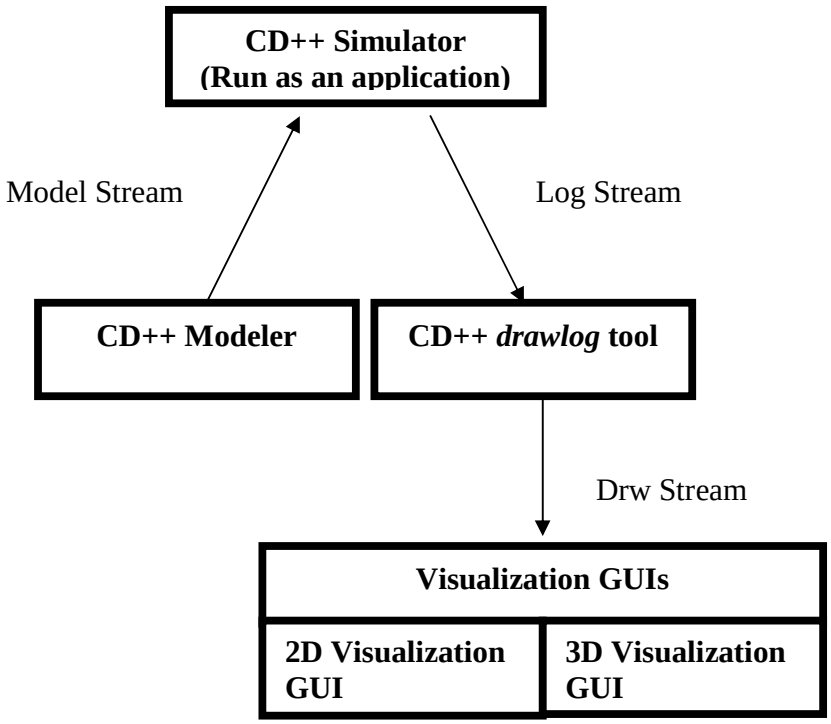


Figure 3.1 CD++ running as stand-alone application

As we can see, there is a separation between model definition, simulation execution, and visualization tools, and the interaction is done through input/output streams. A user can build a model with the CD++ Modeler according to the DEVS and Cell-DEVS specifications, and execute the model with the CD++ simulator running on the local machine. After the simulation is over, a result stream is generated. Using the *drawlog* facility, the result stream can be changed to a stream with different format that can be used to generate graphical outputs using different GUIs.

3.2 CD++ simulator works as a server application

When CD++ works as a server, it runs on the remote machine, and can accept simulation requests and provide simulation service for the clients. The client should send *model* file(s) through the network. When a request is received, the CD++ server executes the model and returns the result. The client will save the results on the local disk as a result file, and then activate the CD++ *drawlog* facility to change its format into another text stream that can be used with the visualization purposes.

According to above description, when the CD++ simulator works as a server, the simulation procedure involves the following steps:

1. The user builds a model
2. The client sends the model to the remote CD++ simulator, and the model is executed on the server

3. The client receives the result stream from the server, and save it in a result file on the local machine
4. The client changes the stream format
5. The user visualizes the results with 2D and 3D GUI

The components in the CD++ simulation environment and their relationship can be illustrated with the following figure 3.2.

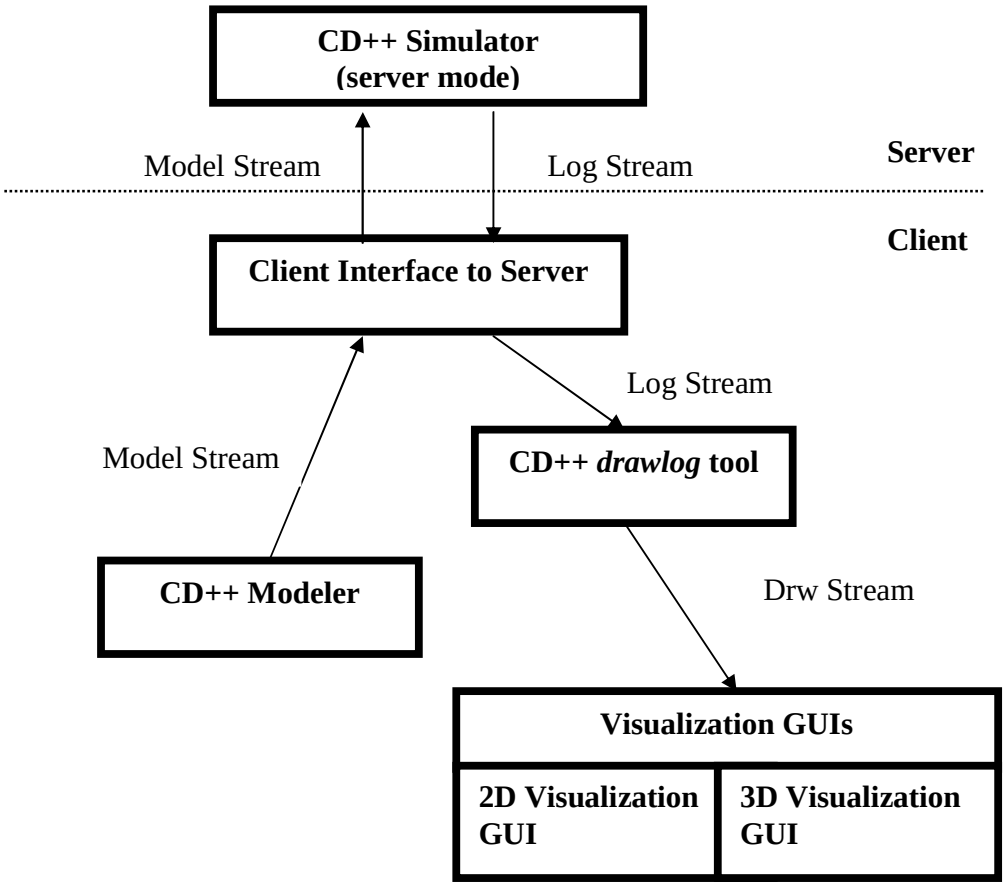


Figure 3.2 CD++ running as client/server application

According to the current design of the CD++ simulator, when it runs as a server, an

interface should be added in the client side to communicate with the server. A user can send models to the remote server for execution, and receive a result stream on the local machine. As illustrated in Figure 3.2, there is still a separation between model definition, simulation execution, and result visualization, and the interaction is done through input/output streams.

When CD++ runs as a server, it expects to receive a model specification on a given TCP port. Whenever it accepts a simulation requirement, a child process is created to serve the specific requirement. When a new simulation requirement arrives, the described process is repeated.

3.3 Guidelines for GUI Design

From above, we can see that the simulation client consists of the CD++ Modeler, the *drawlog* utility, the result visualization GUIs and the Interface with the remote server. The CD++ Modeler is responsible for the model input and model file generation. The visualization GUIs are responsible for the visualization process, deal with presentation issues and provide a visualization environment. The Interface is used to communicate with server. They ensure that the users are presented with a simulation system that consists of familiar and easy-to-use interfaces, and requires little training overhead. In this way, CD++ can be presented to a wider range of users around the world, and learning times and training costs can be significantly reduced.

The user interface is extremely important in a simulation and visualization system for all sorts of users. It has to be both simple and intuitive. The user interfaces provides a communication bridge between users and computer software. User interface designers should identify the types of the users of the application under development, and fully understand the purposes of the application, and design the interface with the user's attitudes in mind.

The whole aim of the GUI is to create user interface components that can be easily manipulated by the user. By building the operation procedures that take place in all the elements of the underlining software into the GUI, it can make complex systems easier to learn, and makes the users more productive.

We developed our simulation system to be used by various users around the world. With this idea in mind, the following provisions were considered:

- The end users have much more varied expertise, so although they may be familiar with the particular technical area the simulation system deals with, they are not familiar with the simulation and visualization system itself.
- The end users are interested in rapidly obtaining data visualizations to assist in analysis for scientific and technical purposes. They do not want to spend time on installing or learning special software.

The average end user usually conducts some research in a scientific area and requires simulation and visualization services to assist with the research and analysis of a real

system. These users may be quite limited in software programming, even in normal computer graphical tools. Therefore, when we develop our simulation system, the following criteria should be followed.

- Easy to use: The average users should not need special knowledge to install the software. They also need an appropriate visualization environment to promote rapid learning. End users should see the environment as an extension to the tools with which they are already familiar, allowing them to focus on the visualization task rather than on learning how to use the system.
- Interactive: End users should be able to control some important visualization parameters as well as directly manipulate and navigate the visualization.
- Multiple platforms: The average user can use any particular platform, to provide a public service. Therefore, the program is necessary to run on most popular operating systems and platforms.

Consequently, a visualization environment should rely as much as possible on standard interface conventions, and where appropriate, allow the user to interact directly with images that provide concrete representations of real-world objects rather than text or forms.

Besides above considerations, the following characteristics also should be considered in the visualization environment to facilitate the user.

- Interactive display: The visualization software should provide a responsive environment, and have functions for the navigation in the visualization, and the edition capabilities.
- Appropriate tools: provide tools for input, such as a slider for changing color classifications, not just text fields for user input.
- Message boxes: provide message boxes for communication with the user.

In our user interface design, all above rules were followed. The program, coded in Java, can run in various environments. The user can specify many parameters for the visualization, and even edit the graphical representation of the results. The user also can navigate in the visualization with many methods. The use steps are built into the program, so the user only needs to follow the sequence in the interface to use the program. In addition, various dialogs are included to communicate with the users and the slider components are used as many as possible for the value input in the interface.

Chapter 4: A Simulation Client for CD++

In this chapter, we will give an overview of a simulation client for the CD++ simulator. This client provides a series of capabilities to use the CD++ simulator as an application or as a server. It provides users with the following three main capabilities:

1. Building DEVS models.
2. Submit a model to a remote CD++ simulation server, receive the execution results locally, and change the result format with the *drawlog* utility.
3. Visualize the results.

The above three capabilities also illustrate the main components in the client. As illustrated in Chapter 3, the main components include the CD++ Modeler, the *drawlog* utility, the Interface between the client and the server, and the result visualization GUIs. Because the *drawlog* utility is an existing tool of the CD++, we just introduce the other three main components below.

4.1 CD++ Modeler

The CD++ tool uses model files to represent coupled models and Cell-DEVS models. As indicated before, the model file details the model components and their relationships. To improve model definition, the client includes a component for model input to create atomic or coupled models. This basic component is the *CD++* Modeler, which consists

of a set of facilities to enable the users to define DEVS models using graphical notations. This application can be used to create atomic or coupled models, which can be executed by the CD++ simulator. The basic functions of the CD++ Modeler include:

1. Building DEVS atomic models using DEVS graphs.
2. Building DEVS coupled models using directed graphs.
3. Saving the newly created models.
4. Loading previously saved models or integrating them as components of a new model.
5. Validating the design and ensuring it conforms to the DEVS rules.

The CD++ Modeler also includes a text editor to write and modify Cell-DEVS models. The application, coded in Java, looks like Figure 4.1 when it starts. There are four components for model input: the *Design Space*, the *Internal transition and external transition selector*, the *Information Space*, and the *Design Space Selector*. The most

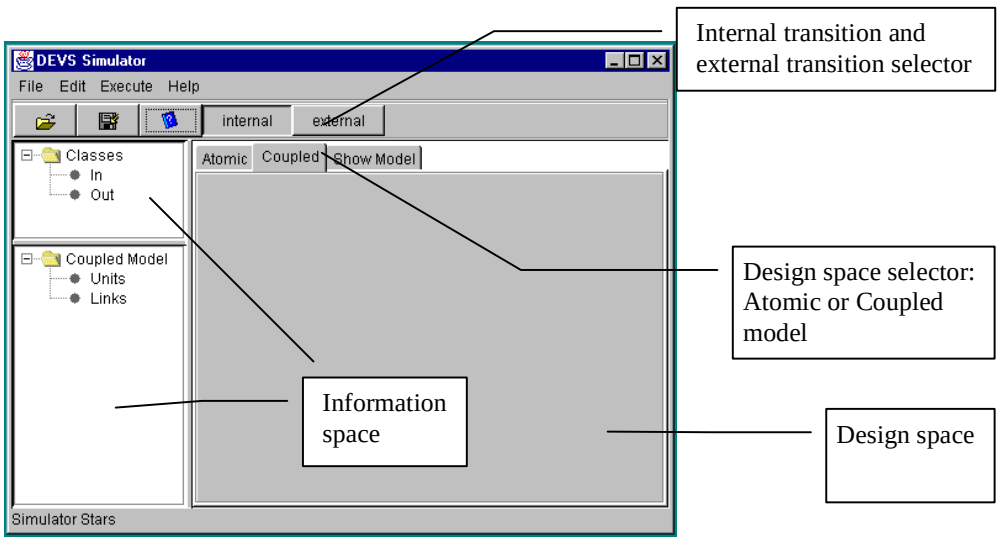


Figure 4.1 CD++ Modeler initial view

important component of the application is the design space where we can build our models.

Before creating the model, the user should select the proper design space according to which type of model (atomic or coupled) is needed. Atomic models can be defined using DEVS graphs, which specify all the states the model goes through, and the state transition relations. Every state in a DEVS graph is specified by an identification and a lifetime as shown in Figure 4.2. The user should assign the state a name for its identification and a value for its lifetime when adding it to the model. As we can see, the states are defined as circles with a name and a lifetime, and their coordinate values in the design space are shown just behind them in the Information space. For instance, the state assigned a name as *end* has a lifetime of 10 time units. Every state transition can be associated with input/output activities using specified ports that can be associated to each state. For instance, the state assigned a name as *start* is associated with the input port *in* and the output port *out*.

States are interconnected using different links to represent the transition relations. Internal transitions are represented by full lines and external transitions by dotted lines. An external transition is activated when an input is received, and it can be associated with an input port, which represents an input event for the model. For instance, figure 4.2 shows that whenever the model receives an input through the *in* port and the model is in *start* state, the model will execute an external transition and change to the *end* state. Internal transition is activated after the model stays in a state for its corresponding

lifetime, and can be associated with an output port to represent the execution of the output function. For instance, if the lifetime (5 time units) of state *mid* is consumed, the output function is executed and the current state and time of the model are sent through the *out* port. After the internal transition completes, the model state changes to *end*.

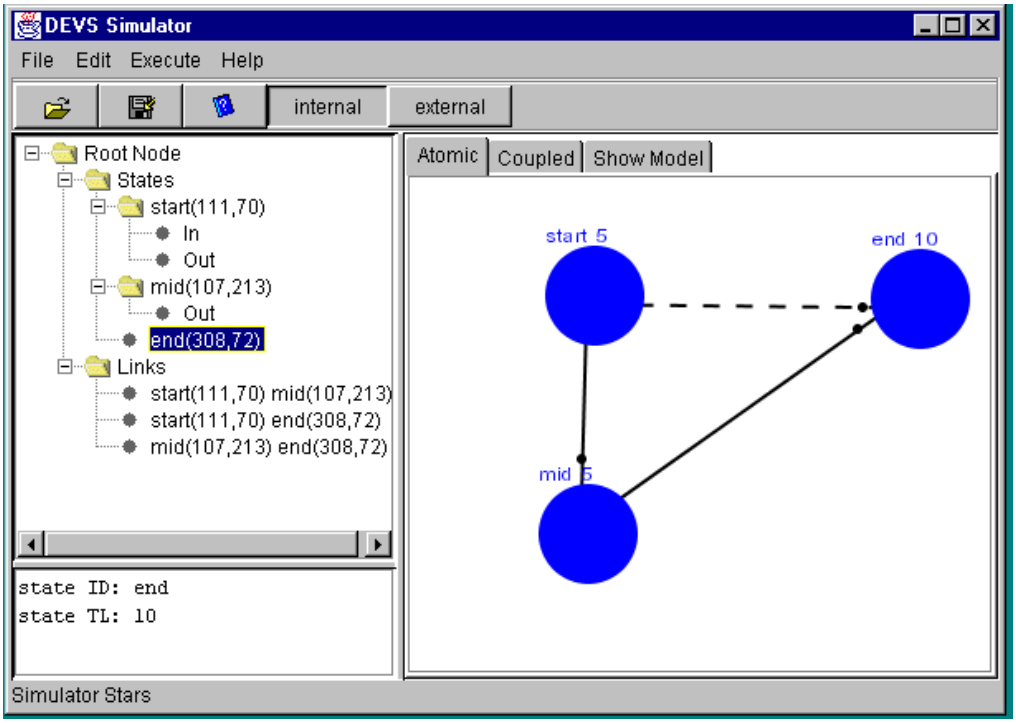


Figure 4.2 CD++ atomic model definition

The user also can refer to models previously coded in C++ after they have been added to the CD++ model database. Once atomic models have been created using any of these methods, the tool permits defining coupled models on the workplace. Coupled models are defined as directed graphs connecting internal component models and the input/output ports of the new model. The first step to build a coupled model is to select the atomic and/or coupled sub-components for the coupled model being built. These sub-component

models can be chosen from the ones previously defined and added to the model database. As it can be seen in Figure 4.3, the component models within the coupled model under definition are represented as squares and the input and output ports of the new model are represented as circles, and their names are shown just beside their corresponding figures. 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 all the component models have been created, we can establish links to connect the input/output ports of these component models to indicate the message transformation relations. For

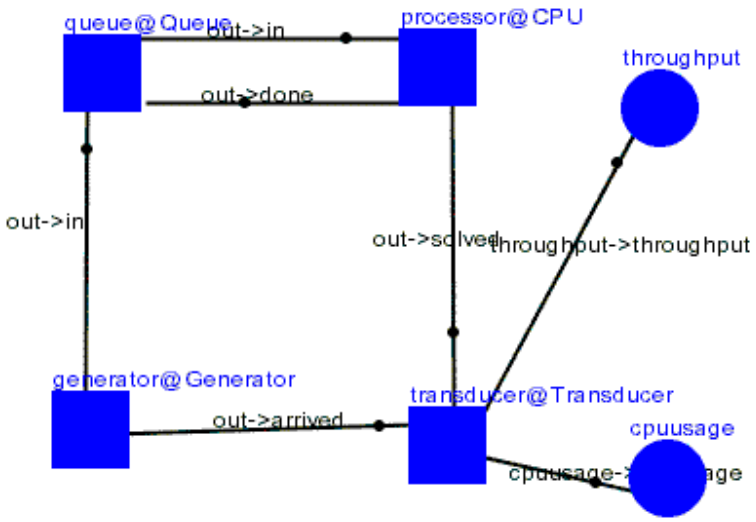


Figure 4.3 Graph representation of a coupled model

instance, Figure 4.3 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 the input/output ports of the coupled model under definition. For instance, the *throughput* port in the *transducer* model of figure 4.3 is connected to the *throughput* port of the coupled model being defined (represented by a circle).

After the graph representing the coupled model is finished, it can be exported as a model file. To ensure that the design conforms to the DEVS formalism, we check the design with the following rules:

1. There is no isolated nodes or links.
2. Every port is connected with at least one link.
3. Each end of a link should be connected with just one port.
4. A link should start from an output port to an input port.

If there is any violation of these rules, a dialog will prompt and show what and where the violation is, so the user can correct the error easily. This checking is done only when the file is exported to a model file to be run in CD++. If the design is just saved to a file, no check is performed because the user may need to continue defining the model later.

4.2 Interfacing with CD++ Server

To execute a simulation model in a remote server, the client must send the model file, an optional external event list and an optional stop time to a remote server through the network. When a request is received, the CD++ server uses a system call to produce a child process before running the specific simulation. Therefore, the server can execute many models simultaneously. The server returns the execution results through the same port. The client will save the results on a local result file, and then activate the CD++

drawlog facility to change its format into another that can be used with visualization purposes.

We developed an Interface on the client side to enable the users to specify the IP addresses of the simulation providers, and send models to them for execution. The following Figure 4.4 shows the *StartDialog* of this interface. We can see that the model file *calor.ma* is selected, the simulation end time is specified as *00:01:20*, and the result format to be used by *drawlog* is specified as 5-digit long with one decimal digit.

To run a model using these facilities, the user should follow the following steps [7]:

a) Set a Configuration File: this file stores the default server address, a port number to be

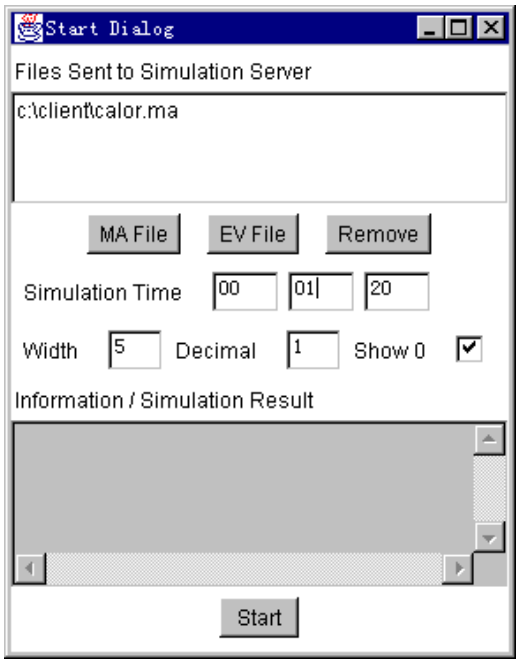


Figure 4.4 StartDialog (select model file, simulation time, and result format)

used, and a file to save this default information. When the client starts, this file will be read and a default server, port and directory will be set up.

b) Select the model stream(s) that will be sent to the server for simulation.

c) Change the Server and input socket: a user can choose server and port addresses different from those defined as default in the configuration file.

d) Input the stop time and result format.

e) Connect to the server.

Using this interface, a client can communicate with any computer with an IP address on the Internet/Intranet. The users even can execute the same model or several different models on different servers at the same time. In addition, as illustrated in Figure 4.5, if one of the clients is used as a hub client to communicate with the server, a remote multi-observer CD++ simulation environment can be easily set up.

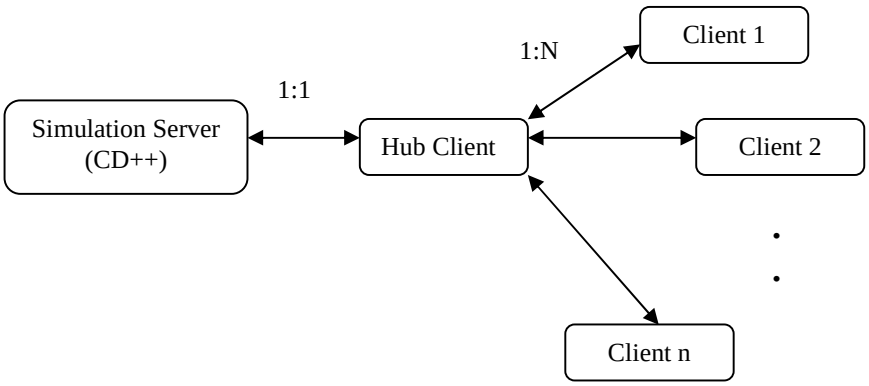


Figure 4.5 Remote multi-observer simulation environment digraph

The hub client can send model files to the server, receive the results, and distribute the results among the other clients as needed. Moreover, the comments of the users can be

distributed among other users through this hub client. Therefore, many users in different locations can observe in the same simulation result.

4.3 Visualization GUIs

As we indicated earlier, the users can use the CD++ simulator as a local application or as a remote server. After the simulation finishes, the user can analyze the simulation results using different visualization tools. A set of visualization tools was introduced and now it is an integral part of the CD++ modeling and simulation toolkit.

For the main part of this thesis work, sophisticated 2D and 3D visualization tools have been developed for the CD++ simulator. With the 2D visualization tools, the users can check the results with 2D s, navigating in the visualization and selecting parameters to specify the items to visualize. With the 3D visualization tools, the users can check the results in a 3D scene. The users can navigate in the visualization with many ways, select the color palette and shape for the nodes, and edit the node matrix or individual nodes. The users also can filter the nodes with specific value ranges to check the results more effectively.

4.3.1 2D Visualization GUI

The 2D visualization GUIs [39] are used to visualize the results of atomic models, Coupled DEVS models, and Cell-DEVS models. In each of these GUIs, navigation

methods are provided for the users to understand the results better.

One of the visualization facilities introduced here enables the users to analyze the input/output values transmitted from/into each of the input/output ports of an atomic model by displaying these values on a graphical display. The information transmitted through each of them is collected in a result file during the simulation. Therefore, the result file stores all the messages sent between the DEVS components.

The visualization routines extract all the messages related to the atomic models and their results, so the user can select any of the atomic models for visualization. As illustrated in Figure 4.6, all the atomic models are listed in the Choice component and the name of the

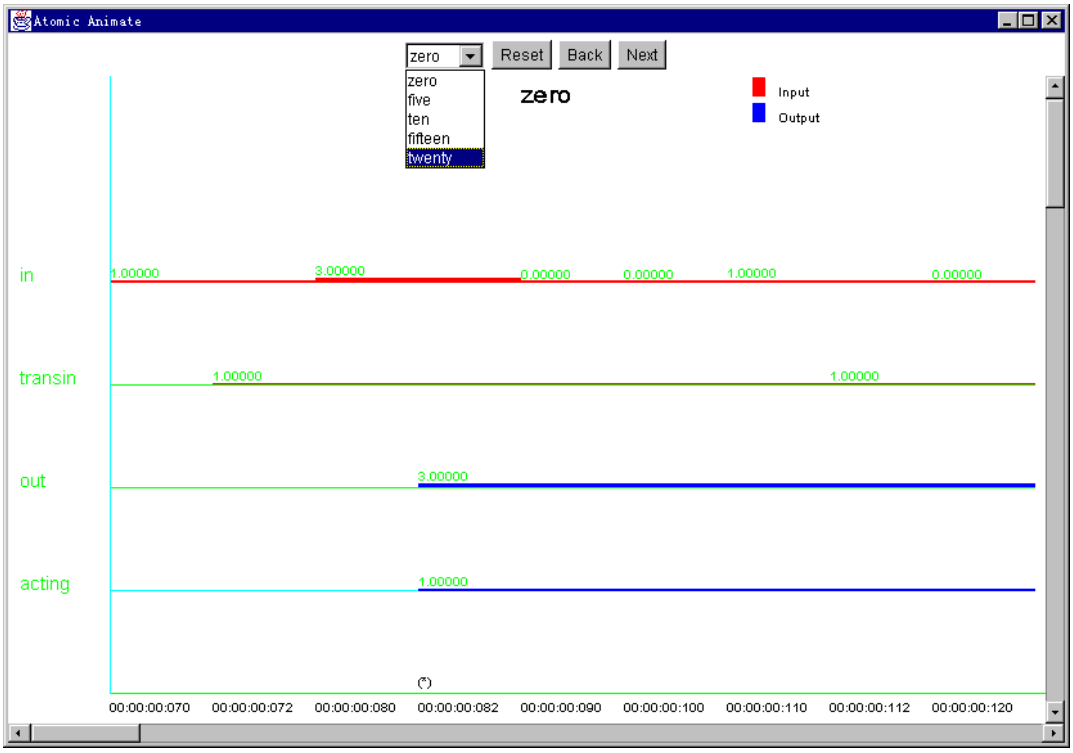


Figure 4.6 An example for atomic model visualization

currently visualized model is displayed below the buttons. The user can select any of models for visualization. The timeline lists all the port on the left and the times on the bottom. The value is shown as a piecewise constant signal, whose height is related to the value displayed.

Each signal starts when the port receives (input port) this value, or sends out this value (output value) and ends when the model generates a new output. The character (*) just above the time means that there was an internal transition at this time. With this graphical display, the user can check all of the input/output values of an atomic model through the whole simulation process.

The execution of coupled models also can be visualized by associating the graphs representing coupled models with the result stream generated during the execution of the coupled model. The user should specify a model file created with the CD++ Modeler and the result file resulting from the execution of the model. The graphical specification for a coupled model defined using the CD++ Modeler, is combined with the result file that contains the information needed for displaying. Figure 4.7 shows an example of execution of this facility. We can see that the model file is displayed on screen, and the values received or sent out by the ports are extracted from the result file and displayed near the corresponding ports. In addition, the timing of the events is included. Therefore, we are able to see the input/output values transmitted during the simulation within a coupled model.

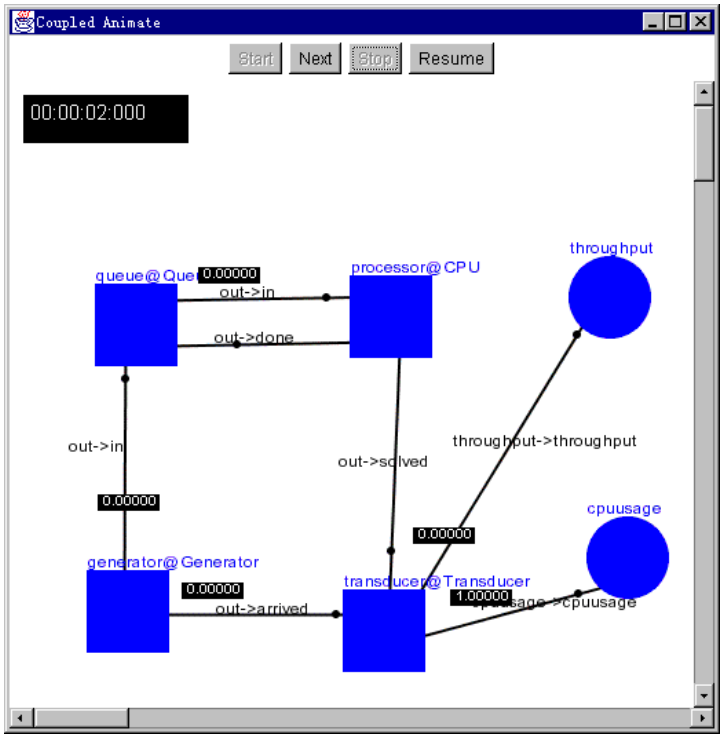


Figure 4.7 CD++ coupled model definition and execution

Cell-DEVS spaces are defined as DEVS coupled models. To better understand the results of Cell-DEVS models, we also added a new facility to visualize the outputs generated by the *drawlog* tool with a graphical interface. Figure 4.8 is an example of 2D Cell-DEVS model visualization. Simulation results for 2D Cell-DEVS models are shown in one plane by giving different color classifications to the different cell values. Simulation results for 3D models are shown by displaying the values of all the planes comprising the model simultaneously. In addition, different color classifications are given to different cell values in order to improve model visualization.

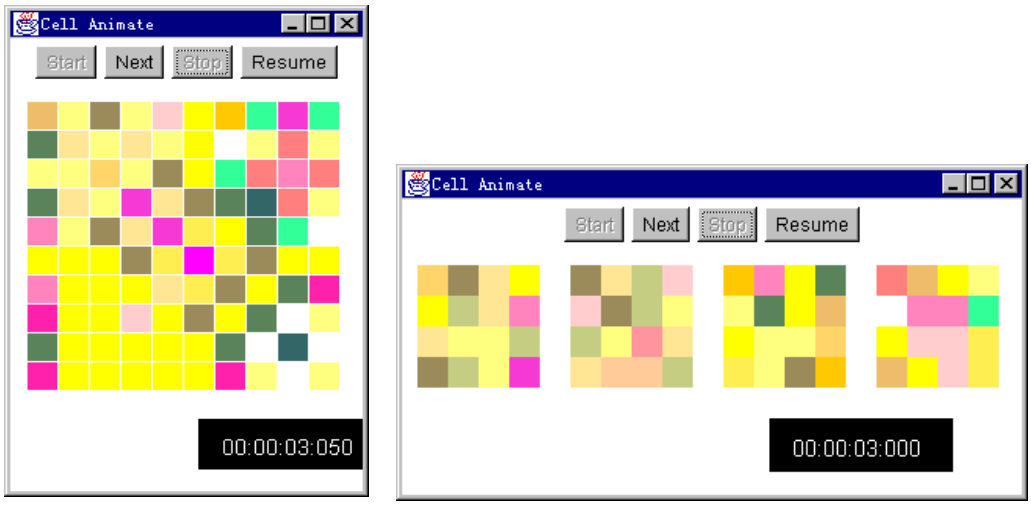


Figure 4.8 Examples for 2D Cell-DEVS model visualization

4.3.2 3D Visualization of Cell-DEVS Model

The 2D Cell-DEVS model visualization tool can be used to visualize 3D Cell-DEVS models, but it has many limitations, which can be analyzed in the output example in above figure 4.8. The result is displayed in several raster images, and the users should keep track of each of them. The users must understand intellectually the relationships between the raster images and compare several 2D raster images to figure out how the real system looks like. In this section, we introduce a 3D GUI for the model result visualization. In this GUI, the users can see the result in a 3D environment. In addition, the user can check the same result in several different viewpoints at the same time.

The 3D visualization GUI is a sophisticated visualization GUI for Cell-DEVS model

result visualization. The results are displayed as a node matrix with the same size as the model. The functions introduced in this GUI are classified into four categories:

1. Navigating the visualization.
2. 3D node matrix (scene) and individual node edition.
3. Shape selection, color palette selection, scale selection for all the nodes in the scene.
4. Filtering the nodes with specific value ranges.

The 3D visualization interface looks like following Figure 4.9 when it starts.

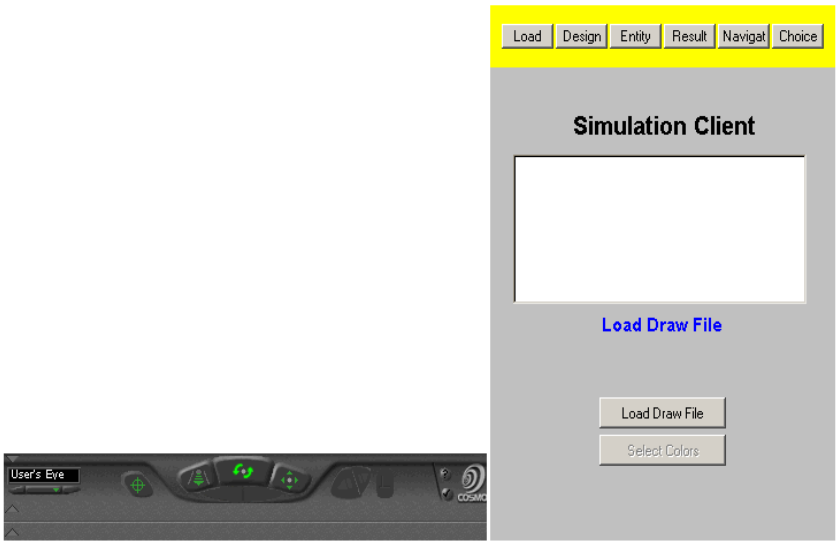


Figure 4.9 3D visualization GUI execution

The left is the VRML scene where the result will be displayed, and the right is the panels to control the scene. There are four panels, *InfoPanel*, *EntityPanel*, *ResultPanel* and *NavigatePanel***.** Each of them is explained in the following sections.

4.3.2.1 InfoPanel

The *InfoPanel* is shown when the application starts. The text field can be used to display debugging information and a label is used to display status information, which will be updated accordingly. It includes methods to select the result file to be visualized. After a result file is loaded, an associated color palette also will be loaded (if it exists). The color palette specifies the color classifications chosen for different value ranges. If the corresponding color palette does not exist, default color classifications and default value ranges will be used. In this panel, a color palette dialog also can be loaded to specify the color classifications for different value ranges.

The color palette is selected with a color palette selection dialog. The user can select default color classifications. The user also can specify the value ranges by providing any two of the following three parameters:

1. Maximum value
2. Minimum value
3. Interval

With these two parameters, the minimum value and interval can be obtained, and then the values for all the value ranges can be calculated. In addition, the users can assign specific color classifications to different value ranges.

4.3.2.2 EntityPanel

This panel allows editing individual nodes in the scene. A list of currently displayed nodes is populated by the *NavigatePanel* every time it updates the nodes in the scene. Before a node can be edited, it should become the editable node (i.e., the only node that can be edited in the scene). A node can become the editable node when the user clicks it in the scene, or selects its corresponding item in the list. After being the editable node, it can be edited with the related methods of the node and the methods in this class. The methods included in the node permit to change the shape, color palette, and size of the selected node. The methods in this class permit to add or remove the individual node in the scene. All the editions on the nodes will be kept during the later visualization process, so the edited nodes can be used to highlight some special nodes. The users also can remove some nodes for better investigation on some interested nodes inside the model.

4.3.2.3 ResultPanel

The *ResultPanel* is used to navigate in the VRML world. Figure 4.10 shows this panel when it is executed. Different methods are defined to control the navigation: 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 continuous display method (this iteration will end only at the end of file); g) a method to go to any selected timestamp; h) a method to remove layer(s); i) a method to display the removed layer(s). These methods call the corresponding methods in the *ResultPanel* Class, which activates the

corresponding methods in the *NavigatePanel* Class.

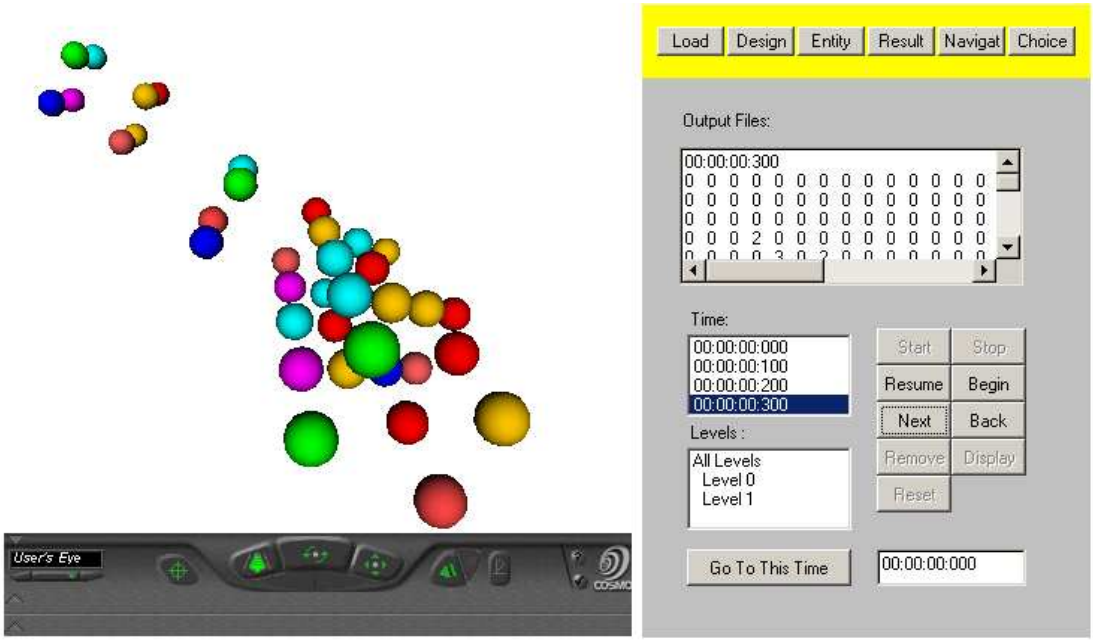


Figure 4.10 ResultPanel execution

4.3.2.4 NavigatePanel

NavigatePanel is the main panel in the application. It stores the currently displayed result, the currently displayed nodes, and the names of every displayed node. Their contents change whenever a scene is updated with a new result or a new color palette. It first initiates the scene as a matrix of transparent nodes with the same size as the model. Therefore, the matrix can map with the model, and any node in the matrix can be associated with a value in the result stream. The class includes methods to add or remove nodes in the scene, to change the shape, color palette, and size of the nodes, and to check the results from a favorite viewpoint.

Chapter 5: Implementation

From previous sections, we know that the client mainly includes three components: CD++ Modeler, result visualization facilities and the client Interface for the remote simulation server. In this chapter, we will give a general overview of the implementation of the 3D VRML visualization GUI and the client Interface. In the end of this chapter, we will introduce the algorithms to transform 2D model to 3D VRML model.

5.1 3D VRML Visualization GUI

The 3D VRML visualization GUI is used to visualize the result in a 3D environment. As illustrated in Chapter 4, we use visible nodes in the VRML scene to represent the results. A node is the basic standalone element in VRML. There are many kinds of nodes in the VRML, some of them are used to define the environment of the scene, and some of them are used to describe a visible object in the scene. The nodes in a VRML scene have a hierarchical relationship (please refer to the on-line report [6] for more detailed explanation of VRML nodes). To display different results in the VRML scene, the visible nodes should be added to the scene and removed from the scene dynamically according to the results being visualized. In addition, the user should be able to navigate in the scene, and edit the nodes for convenient investigation of the result.

To implement this GUI, we need an empty VRML scene, which will be used to hold the

nodes. The VRML scene is the whole VRML displaying area including its environment and all the visible nodes in it. The nodes will be added to or removed from it dynamically to represent the current results. This empty VRML scene is embedded in an HTML file as a root file, and the HTML file should include an applet to control the scene. The applet must include all the functions to update the scene, navigate in the scene, and edit the nodes in the scene. The applet we implemented includes the following functions:

1. Load the result file and its corresponding color palette.
2. Add nodes into the scene or remove nodes from the scene.
3. Change the shape and the size of the nodes, and the interval between them.
4. Select the color classifications for the value ranges, so nodes with different values can be displayed with different color classifications, or nodes with special values are hidden.
5. Navigate in the visualization.
6. Edit the scene and the individual node.

These functions are organized into four groups, each of which is implemented in a different class. These classes are *InfoPanel*, *EntityPanel*, *ResultPanel* and *NavigatePanel*. They extend the *Panel* class in Java to be used as a panel in the 3D visualization GUI. The class diagram and their inheritance relationships are:

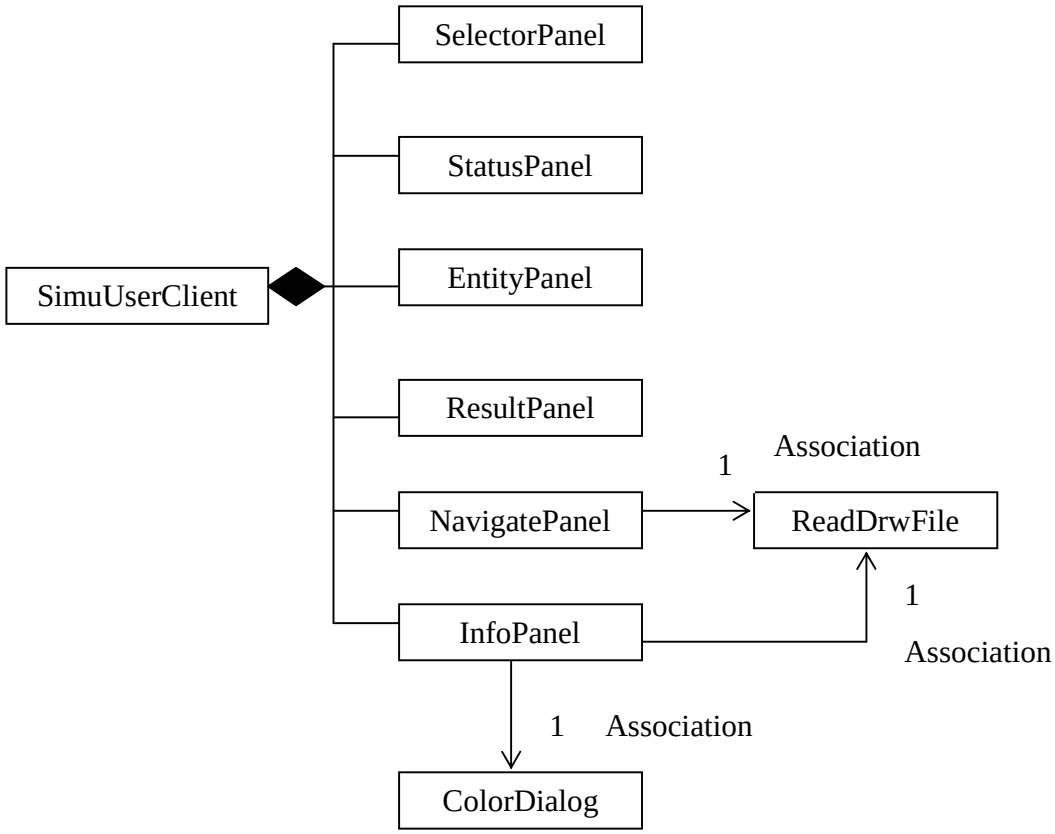


Figure 5.1 Class diagram of the entire VRML visualization applet

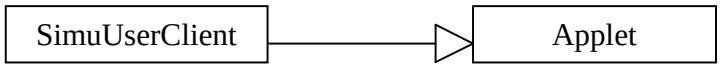


Figure 5.2 Class diagram of the class extending Applet

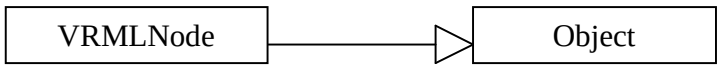


Figure 5.3 Class diagram of the class extending Object

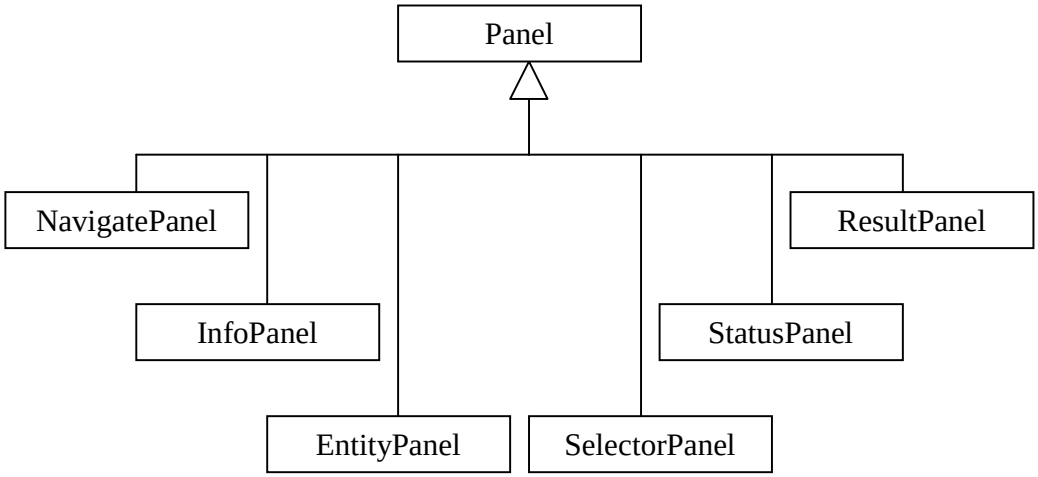


Figure 5.4 Class diagram of the classes extending Panel

From figure 5.1, we can see that the GUI consists of a VRML scene and an applet called *SimuUserClient*. The applet includes six panels, *NavigatePanel*, *InfoPanel*, *EntityPanel*, *SelectorPanel*, *StatusPanel* and *ResultPanel*. The *StatusPanel* is used to display status information of the program. *SelectorPanel* is used for the buttons to select the panels. *ColorDialog* is used to specify color classifications for the values, and can be brought up in *InfoPanel*. *ReadDrwFile* class is used to get the result at a time for display; the *WarnDialog* class is used to display various information to the users. We will now introduce the other four classes (*NavigatePanel*, *InfoPanel*, *EntityPanel*, and *ResultPanel*) later.

The inheritance relationships of most of the classes are straightforward (such as, *InfoPanel* class extends *Panel* class in Java) except by two classes: *SimuUserClient* and *VRMLNode*. The *SimuUserClient* class extends the *Applet* class in Java. *VRMLNode* is

used to represent a visible object in the VRML scene, it extends the *Object* class in Java, and implements the *EventOutObserver* interface for the callback method. It will be called when the node in the VRML scene is clicked.

We will only introduce six important classes, *VRMLNode*, *ReadDrwFile*, *NavigatePanel*, *InfoPanel*, *EntityPanel*, and *ResultPanel*. The main relationship of these six classes is illustrated as follows.

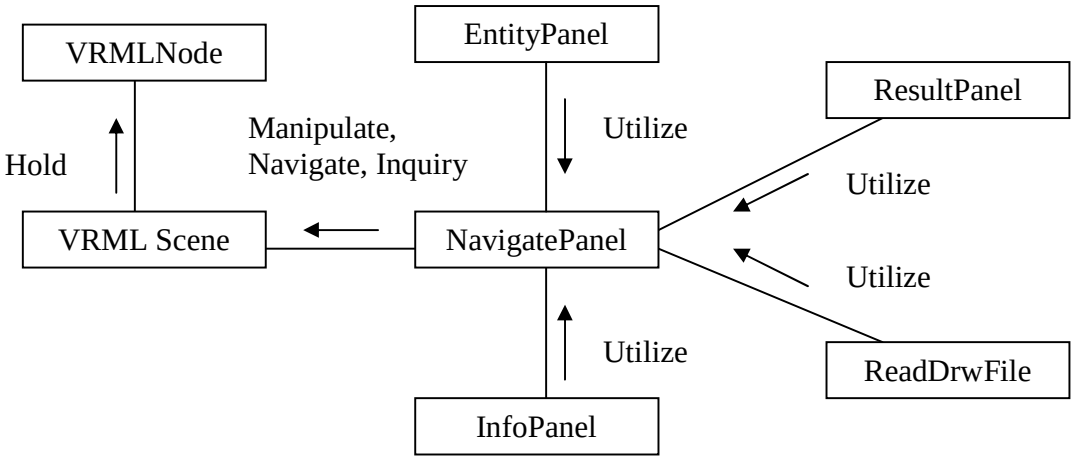


Figure 5.5 Class relationship diagram

The *NavigatePanel* is the only class to control the VRML scene and all the other classes control the VRML scene through this class. Therefore, it should have all the functions to control the VRML scene. These functions include adding or removing a node, updating the scene, navigating in the scene, and editing the nodes in the scene.

5.1.1 The VRML Root File

The VRML root file is embedded in the HTML file, and loaded as an empty scene to hold the nodes representing the result. The root file should have a Group node to hold the nodes in the scene. Since the scene is empty at first, different node matrix should be built dynamically for different models. Any node in the matrix should be authored as a child to this Group node. This is because the nodes defined in the VRML file cannot be removed and we cannot add any node in the scene through the VRML file with the program (applet). Adding nodes to a child field in a group node is the only way to build the 3D node matrix from an empty scene dynamically. In addition, only the nodes in the child field can be removed dynamically from the scene. The root VRML file can be defined simply as follows:

```
#VRML V2.0 utf8
DEF Root Group {}
```

Here a simple Group node, the *Group*, is used, and it is defined as "*Root*" for the access to this node.

This file can contain an actual VRML world, but the only requirement is that a group node, such as, *Group*, named "*Root*" must be present. To facilitate identifying the nodes in the scene and visualizing the results, the following two nodes are also included in this root file.

Background node: The *background* node allows defining the background of the VRML

world. To identify the nodes in the scene easily, we used white as the background color palette for the result visualization.

Viewpoint node: A viewpoint describes a predefined viewing position and orientation in the VRML world. It just acts as a camera in the real world. A VRML world can have any number of viewpoints (or cameras), that is, the interesting positions 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 named *Viewpoints*. We will use these viewpoints to switch to different viewing areas of the scene.

5.1.2 VRMLNode Class

The *VRMLNode* class is used to create visible nodes in the scene to represent the simulation result. Since the nodes should be arranged as a 3D matrix (such as in Figure 5.3) in the scene to represent the results of Cell-DEVS models, the node should have a translation function to be located in the scene. As illustrated in on-line report [6], we can use a *Transform* node because it not only includes a translation function, it also has all the other necessary attributes to represent a node in the scene as well, such as, containing a visible node displayed in the scene. In addition, it also allows manipulating a node's size and orientation.

To construct a *Transform* node to represent a node in the scene, the hierarchy structure should be built in this *VRMLNode* Class according to its definition as in the example file in on-line report [6]. For *Transform* node, we can use the *set-value* function and *get-value*

functions of its translation, rotation and scale fields. The *get-value* function can be used to get the current value of the field. The *set-value* function can be used to change the value of the field. If the value of a field changes, the corresponding attribute of the node also will change. Such as, if the translation field of a node changes, it will move to a new location. Therefore, the position, orientation and size of the nodes can be modified as needed with these *set-value* functions, and their recent values can be obtained with these *get-value* functions. In addition, to shown as a visible node, a *Geometry* node or an *Inline* node should be added to its children field because these two nodes are the only nodes to represent visible objects in the VRML scene. We designed two types of VRML Node classes. One is for primitive shape nodes and the other is for Inline nodes. Therefore, we can use both primitive shape nodes and Inline nodes to represent visible objects in the VRML scene. In addition, a *callback* method is implemented to respond to the clicking on these two kinds of nodes.

5.1.2.1 Primitive VRML Node Class

The Primitive VRML Node can be used to display a primitive shape in the VRML scene. For this kind of node, we can get the *set-value* and *get-value* functions for color palette, texture and transparency fields. Therefore, we can change the color palette, texture and transparency of the node with the *set-value* functions. In addition, we can get the current values of the color palette, texture and transparency of the node with the *get-value* functions.

5.1.2.2 Inline VRML Node Class

To translate and rotate the *Inline* node easily, the *Inline* node can be used as a child node in a *Transform* node. Then if the translation and rotation of its parent *Transform* node change, the child *Inline* node will be relocated and rotated accordingly. The *Inline* node includes all the nodes defined in another VRML file, and uses them as a single node. Since the *Inline* node refers to the nodes that exist in another VRML file, the size and appearance of these nodes cannot be changed. Only translation and rotation operations can be applied to these nodes to locate them in the VRML scene. The VRML file for the *Inline* node is specified with an *url* address, and this class has the function to change the *url* address of the VRML file to be used as the *Inline* node. Therefore, the user can select a different VRML file to describe this child node. It also includes functions to change the position and orientation of the node.

5.1.3 ReadDrwFile class

This class is in charge of reading the values to be displayed from the result stream. It is called by the *NavigatePanel* when the user decides to view the new result. The new result can be one of the following cases:

1. The result at the next timestamp in the result stream.
2. The result at the previous timestamp.
3. The result at any user-selected timestamp.

For case 1, the read pointer is in the right place, so we just need to read the result. For case 2 and 3, we should re-locate the read pointer to the right place first, then we can read the result. The general idea is that we reset the read pointer to the beginning of the result file, and then check the timestamps from the beginning. When we find the timestamp we need, we read the result. In case 2, we have two read pointers, one pointing to the recent timestamp, and the other pointing to the previous timestamp.

Therefore, this class includes the following functions:

1. Reset the read pointer to point to the beginning of the file.
2. Get the number of rows, columns and layers for the initialization of the VRML scene.
3. Read the result and return as a string.

Some separators are inserted in the returned string to facilitate the separation of the whole string into individual values. In this string, all the not displayed zero values are added. This is very important because sometimes the value of zero is not displayed in the result file.

5.1.4 InfoPanel Class

InfoPanel is a subclass of *Panel*. When the users specify a result file for visualization, it checks the result file and its corresponding color palette. If their formats are correct and the corresponding color palette exists, the color palette will be loaded. Then it will call

the reset function to clear the VRML scene and all the information about current display. Finally, it will call a function in *NavigatePanel* class to initialize the VRML scene and begin the visualization.

5.1.5 NavigatePanel Class

NavigatePanel is a subclass of *Panel*. It stores the recently displayed result, recently displayed nodes in the VRML scene, and the names of all the recently displayed nodes. This information changes whenever the scene is updated with the new result or new color palette selection. This class includes the following functions:

1. Initiate the scene.
2. Add or remove the node in the scene.
3. Change the shape, color palette, and size of the nodes.
4. Move to the next result, the previous result, any selected time, or any input time.
5. Begin the visualization from the beginning again.
6. Delete and re-display layers.

At first, it gets the number of rows, columns and layers of the model. It then initiates the scene as a 3D matrix of transparent nodes with the same size as the model. Therefore, each node in the matrix can be associated with a value in the result stream, that is, each value in the result can be represented by a node in the matrix. There are two main reasons to define all the nodes as transparent nodes. a). after a user selects a favorite view area,

the view area is retained when the scene is updated because VRML browsers can remember the viewing point automatically. b). it is easier to add or remove nodes in the scene. With this design, we just need to set the transparency attribute of the nodes to add or remove them. If a node is set as transparent, it is removed from the scene. If a node is set as non-transparent, it is added to the scene. The navigation is implemented with the viewpoint in two methods. First, the viewpoint can be a child of *Transform* node, its position and orientation are changed with those in this *Transform* node. Second, a user can select different viewpoints. A viewpoint is defined as a type of bindable node. For each type of bindable nodes the VRML browser encounters, the first one is bound (used) at first. Moreover, among all the bindable nodes of the same type only one can be bound at any time, so if another one is bound, the recently bound (used) one will be unbound automatically. Therefore, though several viewpoints can exist in a VRML world, only one viewpoint can be used at any time. As in the root file mentioned before, *UserEye*, and a group of viewpoints named *Viewpoints* are defined in the root file. Among these viewpoints, *UserEye* will be bound first, that is, the first used viewpoint. To bind another viewpoint, we just need to get the interested viewpoint in the Group node *Viewpoints*, and set its bind attribute. If another viewpoint is bound, the recent active viewpoint will be unbound automatically, and the scene will be updated according to the newly bound viewpoint.

To change the shape, color palette, and size of the nodes, we call the *set-value* methods of the node with new values. In this panel, the shape, color palette, and size of all the nodes in the scene will be changed. To move to the next time, the previous time, any selected

time, or any input time, we search the time first, and then read the result at that time and display it. To delete a layer, we just need to set the nodes in this layer as transparent, and do not update them at following display. If we set the nodes as non-transparent, then the nodes in this layer will be re-displayed.

5.1.6 EntityPanel Class

EntityPanel is a subclass of *Panel*. It is used to edit individual node in the scene. There is a list on this panel to display the names of all the visible nodes in the scene. Whenever *NavigatePanel* class updates the scene, it will call a method in *EntityPanel* Class to update the entity list. There are many slide components for the edition of the nodes. The current node can be removed or re-displayed.

The functions included in this class are:

1. Change the shape, color palette, and size of the selected node
2. Add or remove the individual node in the scene.
3. Set the slides according to the values of the selected node.

The set-value methods of the node are used to change its color palette, size and translation. When the users want to remove the current node or re-display it, the program will call the corresponding methods in *NavigatePanel* Class.

When a node becomes the current node, the values of its color palette, size and translation can be obtained with the *get-value* method, and then all the slides on this panel can be set according to these values. Therefore, when the users edit the current node, its color palette, size and translation will change continuously, not abruptly. After a node is edited, the VRML can remember the changes automatically. Therefore, the node can remain all the changes later, and the users can specify some interested nodes and check them carefully.

5.1.7 ResultPanel Class

ResultPanel is a subclass of *Panel*. This class controls the result display. There is a text field on this panel used to display the current result. This text field is updated whenever *NavigatePanel* class reads new result for display. There are two lists, one is level list and the other is time list. The level list is updated at first when the program gets the number of levels of the model. A new time item will be added to the time list whenever the result at this time has been displayed in the scene. Different methods are defined to control the navigation, including:

1. Start, stop and resume the visualization.
2. Go to the next time, or go back to the previous time step by step.
3. Continuously display method.
4. Go to any time selected in the time list, or the time input in the text field.
5. Remove layers in the scene or re-display the removed layers.

6. Rest the scene, display all the layers including the removed layers.

To facilitate controlling the display, the display process is implemented as a thread. We can control the display through this thread, which includes start, stop, suspend and resume functions. The continuous display is implemented by the run function of the thread, this function is called when the thread starts. The run function is designed as an endless iteration until the end of the file. The sleep function in this run function can be used to control the display speed. In this thread, there are also other functions to start, stop and resume the display, move to next time, go back to the previous time. These functions will call the related ones in the *ResultPanel* class, which will then call the related functions in the *NavigatePanel* class.

5.2 Client Interface

As indicated in Chapter 3, running a model on a remote server involves five steps: (1) the user builds the model and sends it to the remote server, (2) the server executes the model, (3) sends back the simulation results, (4) the client receives the result stream and (5) changes it to another format, which can be visualized with the 2D and 3D GUIs. All the necessary functions of this interface are listed in section 4.2, and this interface is implemented according to these functions. This interface is designed as an application because it uses a socket to communicate with the CD++ server and saves the result file locally. As an application, this interface can run in various environments without any violation to the security requirements of the computer systems. It also should be

remembered that the design of this interface is based on the recent CD++ simulator design. To implement these functions, we design many classes. The entire class diagram and their inheritance relationships are shown in Figure 5.7:

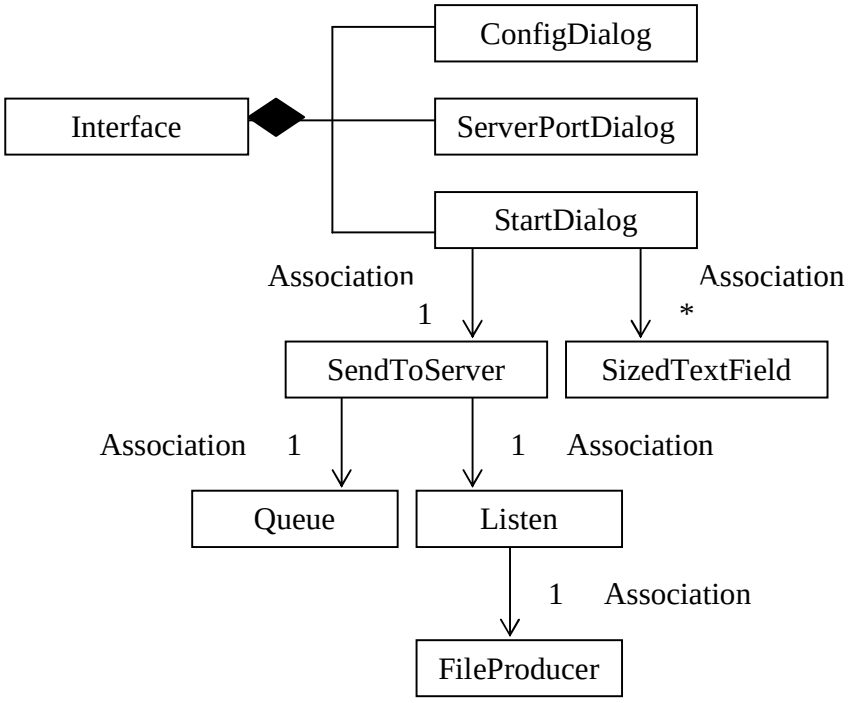


Figure 5.6 Class diagram of the client Interface

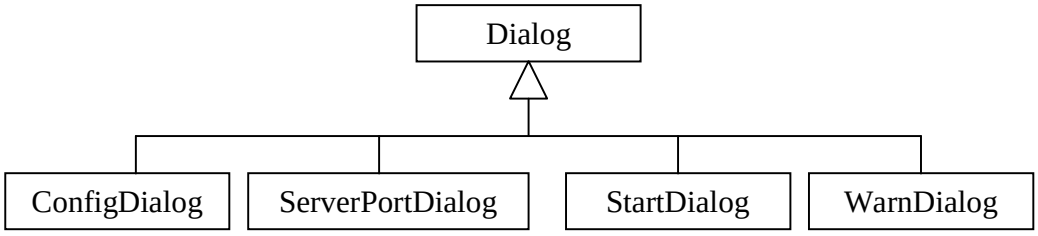


Figure 5.7 Class inheritance diagram of the classes extending Dialog

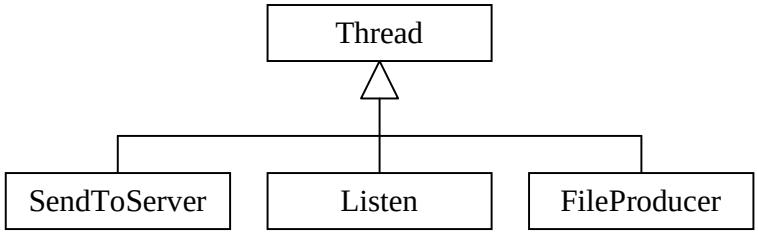


Figure 5.8 Class inheritance diagram of the classes extending Thread

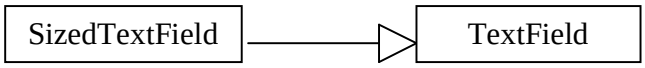


Figure 5.9 Class inheritance diagram of the classes extending TextField

This interface includes three main dialogs: *ConfigDialog*, *ServerPortDialog* and *StartDialog*. The warning dialogs appear accordingly when needed. It is implemented as a menu bar with three menus, each of which corresponds to one of the three main dialogs. The user only needs to follow the sequence of menus to use this interface. He can bring up the dialogs by selecting the menu item under the menu. Under the first menu, there is another useful menu item, it is used to save the choices of the users, which will be used as the default choices next time.

The inheritance relationships of most of the classes are straightforward, except three classes, *SendToServer*, *Listen* and *FileProducer* (which extend the *Thread* class). The *Queue* is actually a vector, and we add many synchronized functions to manipulate it. The *SizedTextField* class is sized text field. It is used to ensure the correct format of the simulation time by controlling the number of the characters in this text field.

The client sends model file(s), an event file (optional) and a stop time (optional) with a specific stream to the specified TCP port on the server as follows:

1. Send the model text file, line by line.
2. Send a delimiter line using only a dot character (“.”).
3. Send the event list file (send a blank line if the top model does not need external events).
4. Send a delimiter line using only a dot character (“.”).
5. Send a line specifying the stop time (format: 00:00:00:00).

The CD++ Server returns the result thought the same TCP port with the following format:

1. The result (log) file (X, Y, * and done messages among components).
2. A delimiter line using only a dot character.
3. The output file.

The main function of this interface is to provide the service to connect with the server. The application gets ready to receive the results before sending the model file(s) to the server. The results will be saved to a queue. After every certain period, the results in the *queue* will be taken out and saved to a file, and the queue will be reset. After all the results have been saved in the result file, the *drawlog* facility will be launched.

Several threads are started for listening on the port and saving the result.

Listening thread: Always listening on the port, once a result arrives, it starts a read thread.

Read thread: Read the result, and save the result in a vector. Start a saving thread if there are a certain number of results in the vector, and reset the vector.

Saving thread: Save the result in the behind of the result file

5.3 Algorithms to Transform 2D Model to 3D VRML Model

As mentioned in Section 4, a DEVS model can be created using the CD++ Modeler tool (as in Figure 4.2). The model created can be saved into a graphical file, which is the graphical representation of the model as in Figure 4.3. The graphical file stores the coordinates and all the other related information for all the nodes, and the links between these nodes of the model. We can use this graphical file, and transform the 2D model into a corresponding 3D VRML model. The algorithms introduced here can be used to transform a DEVS atomic or coupled model built with the CD++ Modeler into a 3D definition of the same model in VRML. Therefore, the user can transform the 2D model graphical file to a 3D VRML model or file, which can be used as an Inline component, or as the start point in new model definition. This is very useful in the model definition with a 3D VRML environment. The reason is that DEVS coupled models usually include many other atomic models and coupled models as its components. These component atomic models and coupled models may exist, but they may have been designed in 2D with 2D Modeler.

The algorithms include three main parts:

(1) Coordinate Value Transformation: locate the transformed 3D VRML model in the center of the VRML scene.

(2) Node Transformation: how to transform the nodes in 2D model.

(3) Link Transformation: how to transform the links in 2D model.

Coordinate Value Transformation:

In the corresponding 3D VRML model file, we can use the same coordinate x and y values as in 2D graphical file, and leave the z value to be decided by the users when needed. For our algorithms, we suppose the z value is zero. However, to allocate the center of the transformed VRML model to the center of the 3D VRML scene, we transform the 2D model graph first, as indicated in Figure 5.11.

Where

x_2: the x coordinate value in 2D graphical file;

y_2: the y coordinate value in 2D graphical file;

x_3: the transformed x coordinate value in 2D graphical file;

y_3: the transformed y coordinate value in 2D graphical file;

min: the minimum value;

max: the maximum value;

From above figure 5.11, we can see that to locate the 2D model graph to the origin (that is, the transformed VRML model to the center of the 3D VRML scene), the following

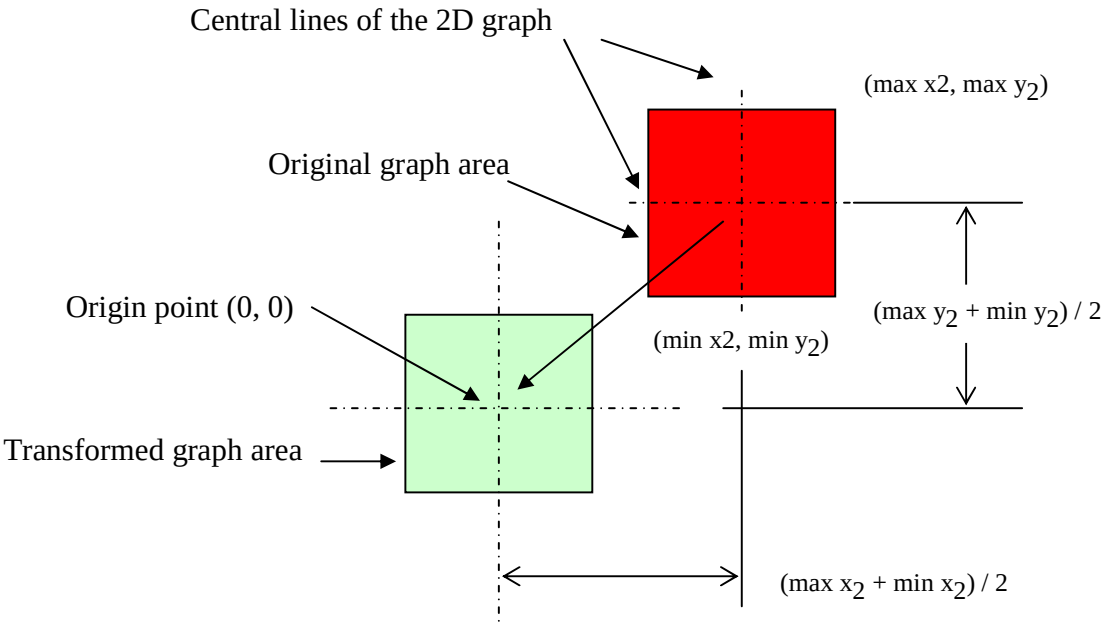


Figure 5.10 Coordinate transformation digraph

transformation equations should be applied:

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

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

After this transformation, the central point of the 2D model graph will be relocated to the origin, and all the nodes and links in the model graph will be relocated with the same transition accordingly. Therefore, we can use the new coordinate values for our 3D VRML model, and calculate the geometric parameters with the new coordinate values.

Node Transformation:

For a node in a 2D graphical file, as we indicated before, we can use a primitive node or an Inline node to represent it in 3D VRML scene. To locate it in the VRML scene, we can use it as a child node of a Transform node, and we just need to set the translation field of this Transform node with the results of above transformation equations.

Link Transformation:

The links in the model graph are used to represent the message transformation relations between the nodes. For each link in 2D model, we will use a cylinder and a cone on the cylinder to represent it in 3D VRML scene. The cone is used to indicate the direction of message transformation. Since we use a cylinder and a cone to represent a link, we need to know the default cylinder and cone in VRML because all the cylinders and cones in VRML scene are transformed from the default cylinder and cone. As illustrated in Figure 5.12, the default cylinder in VRML has one unit in radius, 2 units in length, and its center at the origin and the default cone has one unit in bottom radius, 2 units in length, and its center at the origin.

Figure 5.11 Default Cylinder node and Cone node in VRML

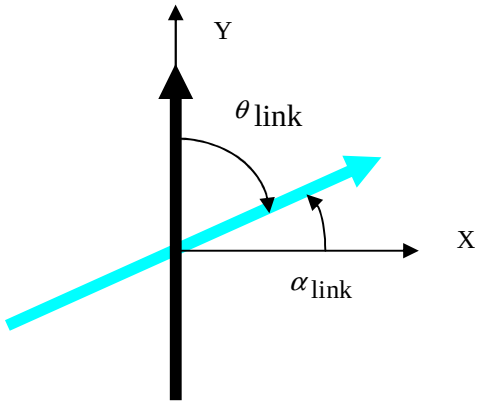
In order to use them to represent links in VRML scene, first we need to transform the default cylinder to let it have the same length as the corresponding link, and then arrange

it to the same position and orientation as the corresponding link. To implement these transformations, we should calculate the length l_{link} and the orientation α_{link} of the corresponding link. First, we should obtain the transformed coordinates (x_{3_start}, y_{3_start}), (x_{3_end}, y_{3_end}) with above equations for the two ends of the link. Then, calculate the length l_{link} and the orientation (the angle α_{link} between the link and the x coordinate) with the following equations:

$$L_{\text{link}} = \sqrt{(x_{3\_end} - x_{3\_start})^2 + (y_{3\_end} - y_{3\_start})^2}$$

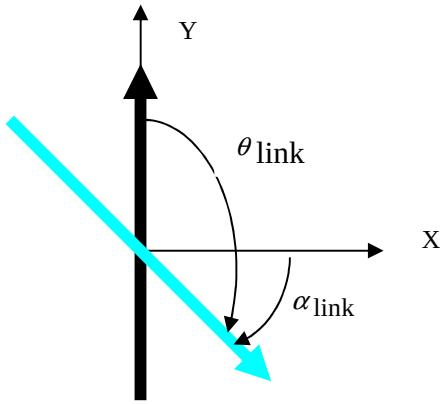
$$\alpha_{\text{link}} = \tan^{-1}((y_{3\_end} - y_{3\_start}) / (x_{3\_end} - x_{3\_start}))$$

With the length L_{link} of the link, we can transform the default cylinder to let it have the same length as the corresponding link with the height field in the Cylinder node. Its radius also can be specified with the radius field in the Cylinder node as needed. With the orientation of the link, we can calculate the necessary rotation angle θ_{link} of the transformed cylinder, as illustrated in following figure 5.13. In VRML, counterclockwise rotation angle is positive, and the clockwise rotation angle is negative. The calculation has four conditions as in figure 5.13. We suppose that the black cylinder is the transformed cylinder with the same length as the corresponding link. We rotate this transformed cylinder with θ_{link} to let it have the same orientation as the corresponding link, and now the transformed cylinder, represented as the gray cylinder, has the same length and orientation as the corresponding link. The arrow end (x_{3_end}, y_{3_end}) just indicates the input port of the link, which is one of the ends of the link.



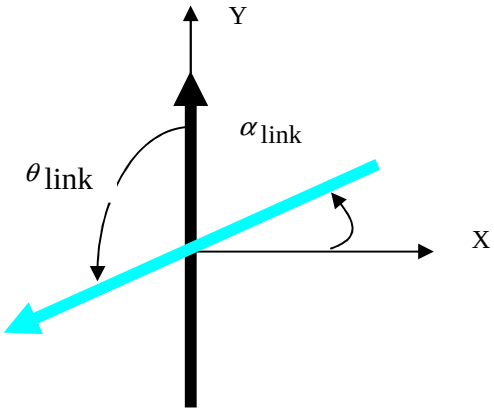
(1)

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



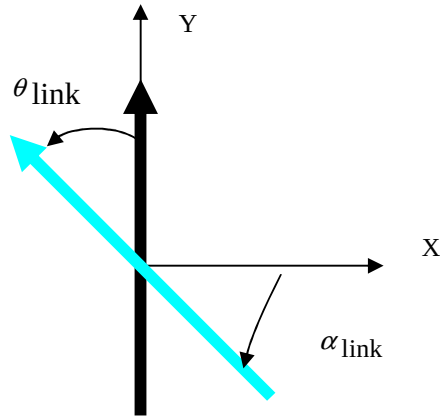
(2)

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



(3)

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



(4)

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

Figure 5.12 Calculate the rotation angle

After these transformations, the cylinder has the same length and orientation as the corresponding link, and we just need to relocate the cylinder to the same location (x_{center}, y_{center}) as the corresponding link, as indicated in figure 5.14. The central point (x_{center}, y_{center}) of the link can be calculated with following equations:

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

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

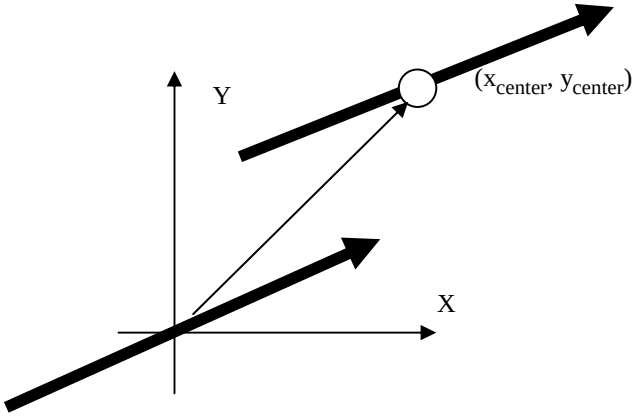


Figure 5.13 Link transformation

After the link transformation, we can add the cone to indicate the direction of the message transformation. The cone will have a radius several times of the radius of the cylinder, and a length comparable with its bottom diameter. It is at the central line of the link and has the same orientation as the cylinder. To get all these transformations, we just need to set the corresponding translation, rotation and scale attributes of the Transform node for the link.

Chapter 6: Execution Examples

In this chapter, we will show different functions of the client with some examples. All components in the client have been tested thoroughly with various models. In this chapter, we will focus on examples of our 3D result visualization facilities, presenting different viewpoints, geometries, scales and color classifications. Finally, we will give an example about access to a remote server. For detailed steps about how to use these GUIs, you can refer to [7].

6.1. 3D Result Visualization

As illustrated in Chapter 2, the result stream obtained when executing a Cell-DEVS model can be changed to another output format using the *drawlog* facility. This new output format is a series of two, three or more dimensional matrices depending on the number of the dimensions of the cellular model. In 3D result visualization facility, it is shown as a three-dimensional matrix of colored nodes with the same size. Each node corresponds to a value in the result matrix at a time, and the color palette of the node is specified by its value and can be set with a color palette selection facility. If the model has more than three dimensions, the *drawlog* facility can choose the 3D model to show by ignoring one, two, three or more dimensions until the results becomes 3D matrices. Therefore, it also can be visualized with these GUIs.

In the following examples, we will use a fragment of the result file of a 3D version of the heat diffusion model, as in following Figure 6.1.

```
Line : 166 - Time: 00:00:00:000
        0    1    2    3         0    1    2    3         0    1    2    3         0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 24.0 24.0 24.0 24.0|   0| 24.0 24.0 24.0 24.0|   0| 24.0 24.0 24.0 24.0|   0| 24.0 24.0 24.0 24.0|
   1| 24.0 24.0 24.0 24.0|   1| 24.0 24.0 24.0 24.0|   1| 24.0 24.0 24.0 24.0|   1| 24.0 24.0 24.0 24.0|
   2| 24.0 24.0 24.0 24.0|   2| 24.0 24.0 24.0 24.0|   2| 24.0 24.0 24.0 24.0|   2| 24.0 24.0 24.0 24.0|
   3| 24.0 24.0 24.0 24.0|   3| 24.0 24.0 24.0 24.0|   3| 24.0 24.0 24.0 24.0|   3| 24.0 24.0 24.0 24.0|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+

Line : 340 - Time: 00:00:01:000
        0    1    2    3         0    1    2    3         0    1    2    3         0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 24.0 24.0 24.0 24.0|   0| 24.0 24.0 24.0 24.0|   0| 24.0 24.0 24.0 24.0|   0| 24.0 24.0 24.0 24.0|
   1| 24.0 24.0 24.0 24.0|   1| 24.0 24.0 24.0 24.0|   1| 24.0 24.0 24.0 24.0|   1| 24.0 24.0 24.0-12.5|
   2| 24.0 24.0 24.0 24.0|   2| 24.0 24.0 71.4 24.0|   2| 24.0 24.0 24.0 24.0|   2| 24.0 24.0 24.0 24.0|
   3| 24.0 24.0 24.0 53.5|   3| 24.0 24.0 24.0 24.0|   3| 24.0 24.0 24.0 -1.6|   3| 24.0 24.0 24.0 24.0|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+

Line : 644 - Time: 00:00:02:000
        0    1    2    3         0    1    2    3         0    1    2    3         0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 25.2 24.0 25.2 23.7|   0| 24.0 24.0 27.8 24.2|   0| 23.0 24.0 23.0 21.5|   0| 22.5 24.0 22.5 22.7|
   1| 22.5 24.0 24.4 24.9|   1| 24.0 25.9 25.9 23.0|   1| 22.5 24.0 24.4 20.5|   1| 22.5 21.1 22.5 22.5|
   2| 25.2 25.9 27.1 25.6|   2| 27.8 25.9 25.9 26.1|   2| 23.0 25.9 24.9 23.4|   2| 22.5 24.0 26.3 22.7|
   3| 25.2 26.4 27.1 23.1|   3| 24.2 25.9 26.1 26.1|   3| 23.0 22.0 24.9 25.3|   3| 24.2 24.0 24.2 21.2|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+

Line : 968 - Time: 00:00:03:000
        0    1    2    3         0    1    2    3         0    1    2    3         0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 24.1 24.5 24.7 24.0|   0| 24.5 24.7 24.6 24.2|   0| 23.4 23.8 24.0 23.5|   0| 23.1 23.6 24.0 23.3|
   1| 24.0 24.6 24.7 23.8|   1| 24.2 24.5 24.6 24.4|   1| 23.3 23.5 24.0 23.6|   1| 23.2 23.8 23.8 22.8|
   2| 24.7 25.1 25.1 24.6|   2| 24.6 25.1 25.9 24.7|   2| 24.0 24.4 24.4 24.1|   2| 24.0 24.2 24.2 23.9|
   3| 24.5 24.6 25.1 24.9|   3| 24.7 25.3 25.2 24.6|   3| 23.9 24.4 24.5 23.4|   3| 23.6 24.0 24.2 23.9|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+


Line : 1292 - Time: 00:00:04:000
        0    1    2    3         0    1    2    3         0    1    2    3         0    1    2    3
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
   0| 24.1 24.3 24.4 24.2|   0| 24.2 24.4 24.5 24.2|   0| 23.8 24.1 24.1 23.9|   0| 23.8 24.0 24.0 23.8|
   1| 24.1 24.3 24.4 24.2|   1| 24.2 24.5 24.5 24.2|   1| 23.9 24.1 24.1 23.8|   1| 23.8 23.9 24.0 23.8|
   2| 24.4 24.6 24.6 24.4|   2| 24.5 24.6 24.7 24.5|   2| 24.1 24.3 24.3 24.1|   2| 24.0 24.2 24.3 24.0|
   3| 24.4 24.6 24.6 24.3|   3| 24.4 24.6 24.7 24.5|   3| 24.1 24.2 24.3 24.1|   3| 24.0 24.2 24.2 24.0|
    +--------------------+    +--------------------+    +--------------------+    +--------------------+
```

Figure 6.1 A fragment of an example result file

We also need to specify the color classifications for the values in the result file. The color palette selection can be brought up in the *InfoPanel*. Whenever the new color classifications have been selected, the nodes in the result space will be updated and displayed with the new color classifications corresponding to their values. In our examples, we will use the color palette selection illustrated in Figure 6.2.

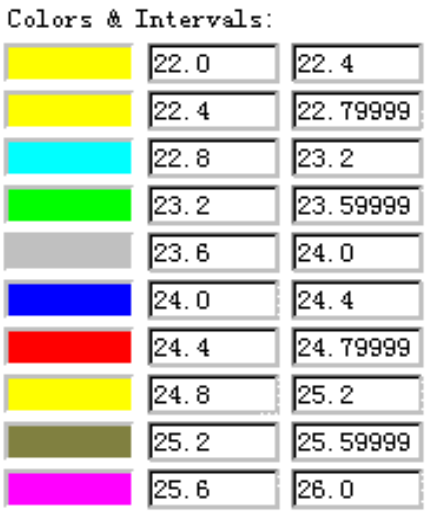
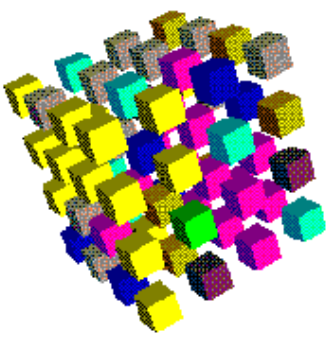


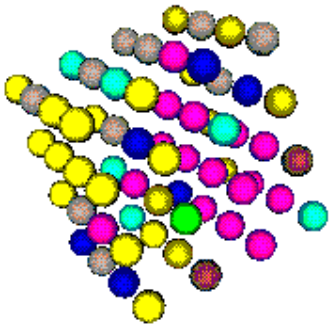
Figure 6.2 Color palette selection

6.1.1 Geometry Selection

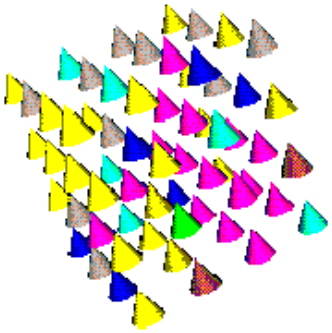
We can use different geometries to represent the nodes in the result space. The user can select box, sphere, cone or cylinder as the geometry of the nodes in the result space as in following Figure 6.3. It uses the result in Time: 00:00:03:000 in above example file.



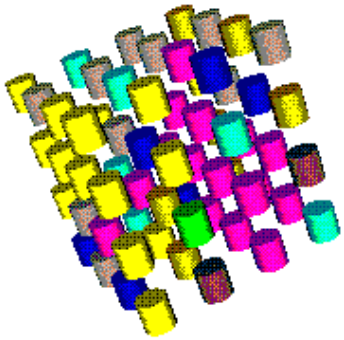
(1) Box



(2) Sphere



(3) Cone



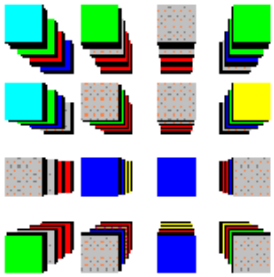
(4) Cylinder

Figure 6.3 Different geometries

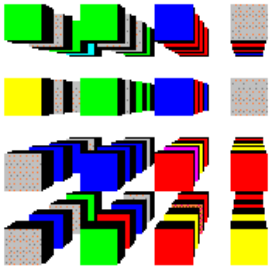
As we can see, the original result matrix is now shown as a 3D VRML model consisting of colored nodes with the same size. Each node corresponds to a value in the result matrix, and the color palette of the node is specified by its value and set with a palette selection facility.

6.1.2 Different Viewpoints

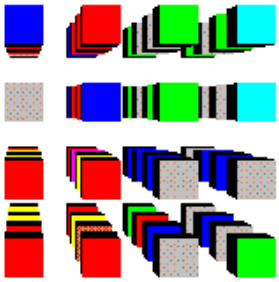
Another facility available enables the users to select different viewpoints to visualize the results. This can be seen in figure 6.4.



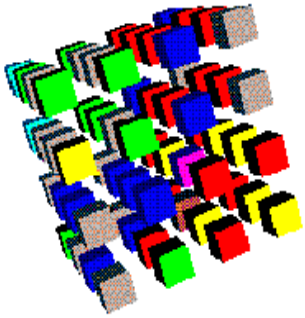
(1) User's Eye



(2) Side view 1



(3) Side view 2



(4) Random viewpoint

Figure 6.4 Different viewpoints

The user can select any viewpoint defined in the VRML file to visualize the result. Here, we select viewpoints *User's Eye* and the *Side view 1* and *Side view 2*. In addition, the user can select any viewing area, that is, any viewpoint, as in (4).

6.1.3 Continuous display

The navigation facilities enable displaying the results following the sequence of the original simulation. The user can see the results continuously, advance step by step, move backwards, or jump to any certain time. Figure 6.5 shows the execution results obtained using these functions.

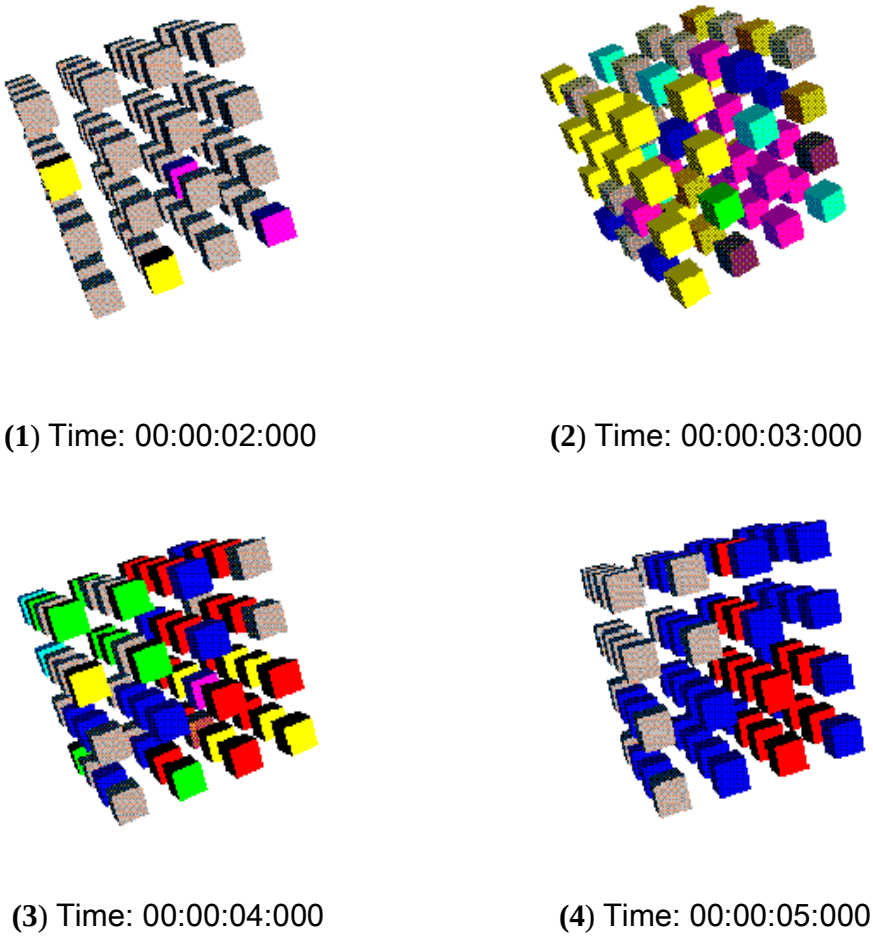


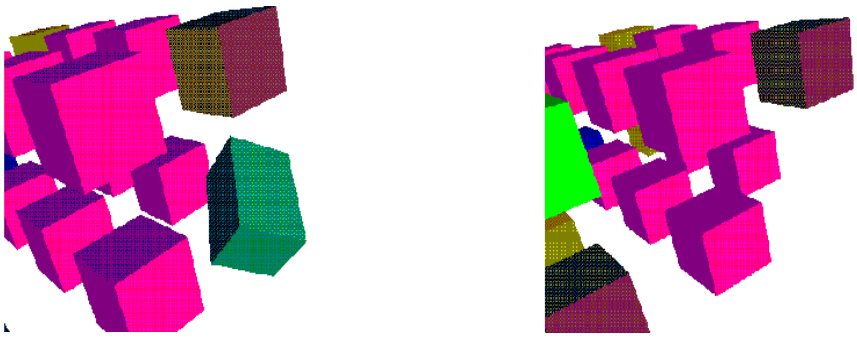
Figure 6.5 Continuous advance

If the result is displayed continuously with the same sequence as the simulation, the user can check the simulation progress. (1), (2), (3) and (4) respectively display the results in

time 00:00:02:000, 00:00:03:000, 00:00:04:000 and 00:00:05:000.

6.1.4 Edit a Node in the Scene

The user also can edit a single node in the scene, changing its shape, color palette or position, deleting it or re-displaying it, as shown in the following figure.



(1) (2)

Figure 6.6 Edit single node

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 palette, size, translation and rotation, or delete the node. In addition, the user can redisplay a previously deleted node.

6.1.5 Delete a Layer

The user can remove any layer in the display to check the result of certain phenomena easily. In following figure 6.7, we show the previous examples, but level 1 was removed, which can be redisplayed later if needed.

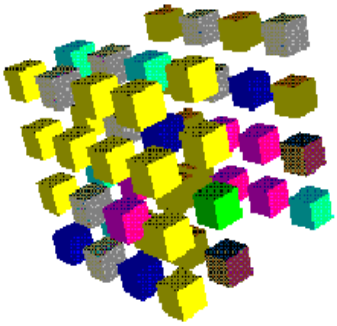


Figure 6.7 Delete layers

6.1.6 Scale Nodes

The nodes in the scene can be scaled up or down, as shown in Figure 6.8, where the nodes have been scaled to the minimum distance and cannot be scaled further. The nodes also can be scaled to smaller size.

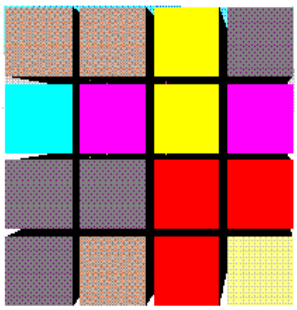


Figure 6.8 Scale nodes

6.1.7 Transparent Display

Sometimes, the users do not want to display the nodes with special values. They can do this by setting the color palette as white in the color palette dialog as in figure 6.9. Then the nodes with the values within those special value ranges will not be displayed as in figure

6.10.

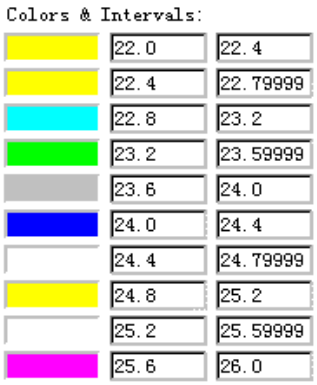


Figure 6.9 Color selection

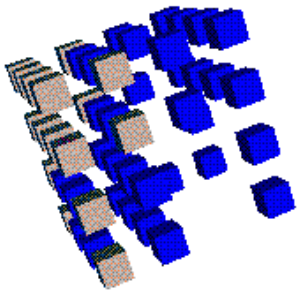
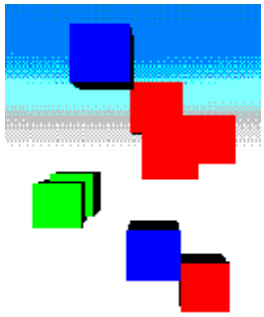


Figure 6.10 Transparent display

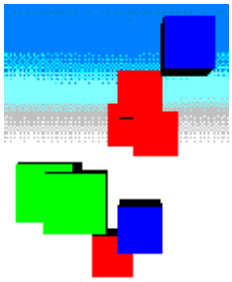
We can see the nodes whose values fall in the ranges 24.4 to 24.8 and 25.2 to 25.6 are not displayed. This is very useful if the user just wants to monitor the nodes with some special value ranges.

6.2 Multi-View

Multiple instances of the GUI can be activated to visualize the same result, using different viewing areas, as shown in the following figure 6.11. Different geometry or Inline nodes can be used if needed, as shown in the following figure 6.12.



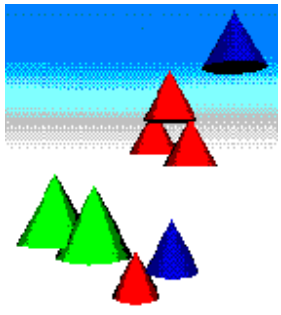
(1) Side View 1



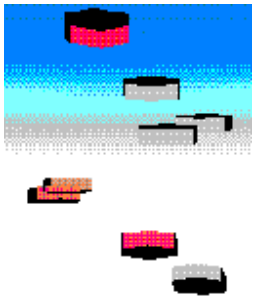
(2) Entry View

Figure 6.11 Use different viewpoints

In Figure 6.11, two GUIs are used to visualize the same result with different viewpoints. In addition, different GUIs can use different geometry or Inline nodes, as in following Figure 6.12



(1) Use Cone in Side View 1



(2) Use Inline node in Entry View

Figure 6.12 Use different geometry

6.3 Remote access

When the CD++ simulator works as a server, the Client Interface should send model files to the server and the server will send back the result. All the results will be saved in a local file, as the one in Figure 2.3. Then the client will launch the *drawlog* facility to change its format to be visualized by the 2D and 3D GUIs.

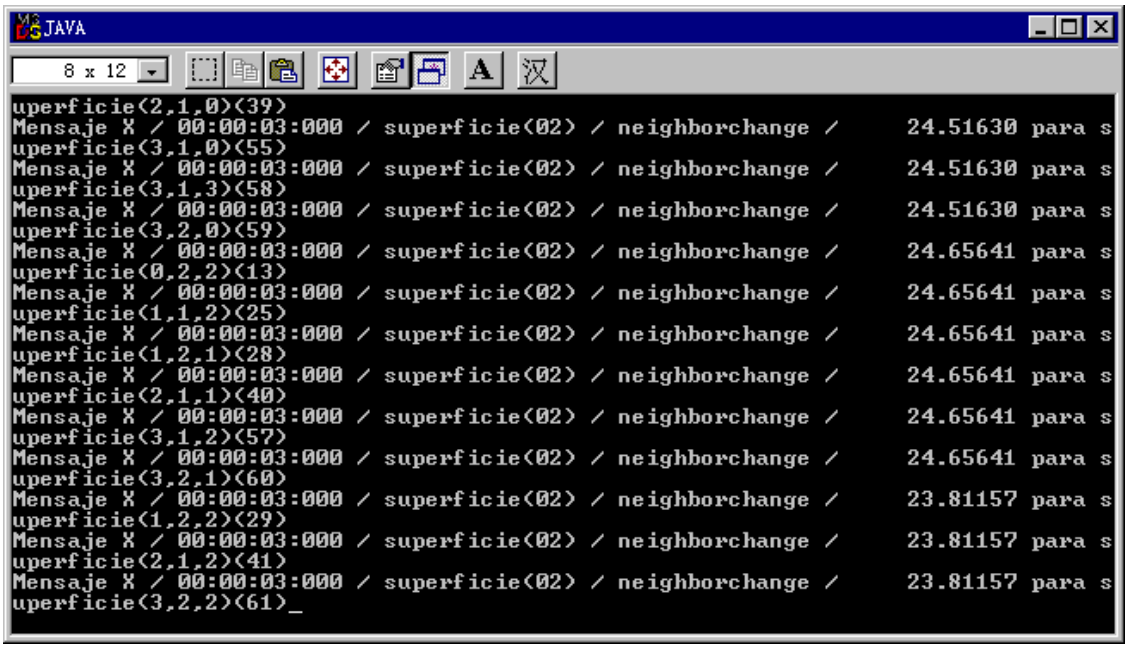


Figure 6.13 Remote access example

The above figure 6.13 shows the client is receiving the results from the server. Each line is one of the messages sent between components in the model with a timestamp. At the same time, the results will also be saved in a buffer. When the buffer is full, the results are attached to the end of a local file and the buffer is reset.

6.4 Transform 2D model to VRML 3D Model

A 2D DEVS model (saved as a graphical file) can be transformed to a VRML 3D model with a tool which implements the algorithms illustrated in section 5.3. This tool is a utility included in the VRML visualization GUI. For instance, the 2D model in Figure 4.3 is transformed into the 3D VRML model in following Figure 6.14.

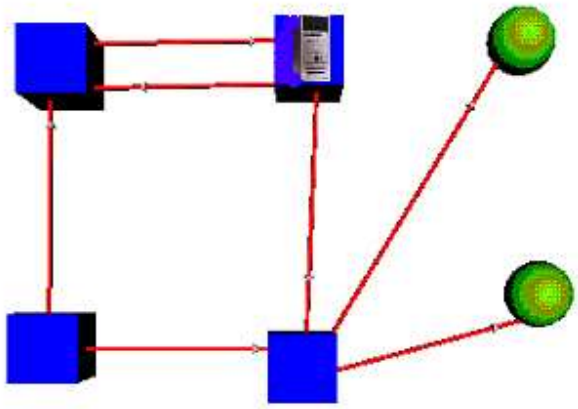


Figure 6.14 Transformed VRML 3D model example with texture

The coupled models are represented with box nodes. These box nodes can be textured with images to represent the components in the real system, such as, one of them is textured with a processor image to represent a processor in the real system. In addition, the coupled models also can be represented by Inline nodes. The input/output ports are represented with nodes with sphere shape, they also can be represented as other suitable shapes. There is a cone on each link to indicate the direction of the message transformation between the components. This utility can be used to transform a DEVS atomic or coupled model built with 2D Modeler into a 3D definition of the same model in

VRML. Therefore, the users can transform the old 2D models directly, and need not input them again, and use the transformed VRML 3D model as a start to build new models.

The other utility is to transform 2D model (saved as a graphical file) to a VRML file. The VRML file in Figure 6.15 is translated from the 2D model in Figure 4.3. Any object in the scene is represented as a Transform node in the file.

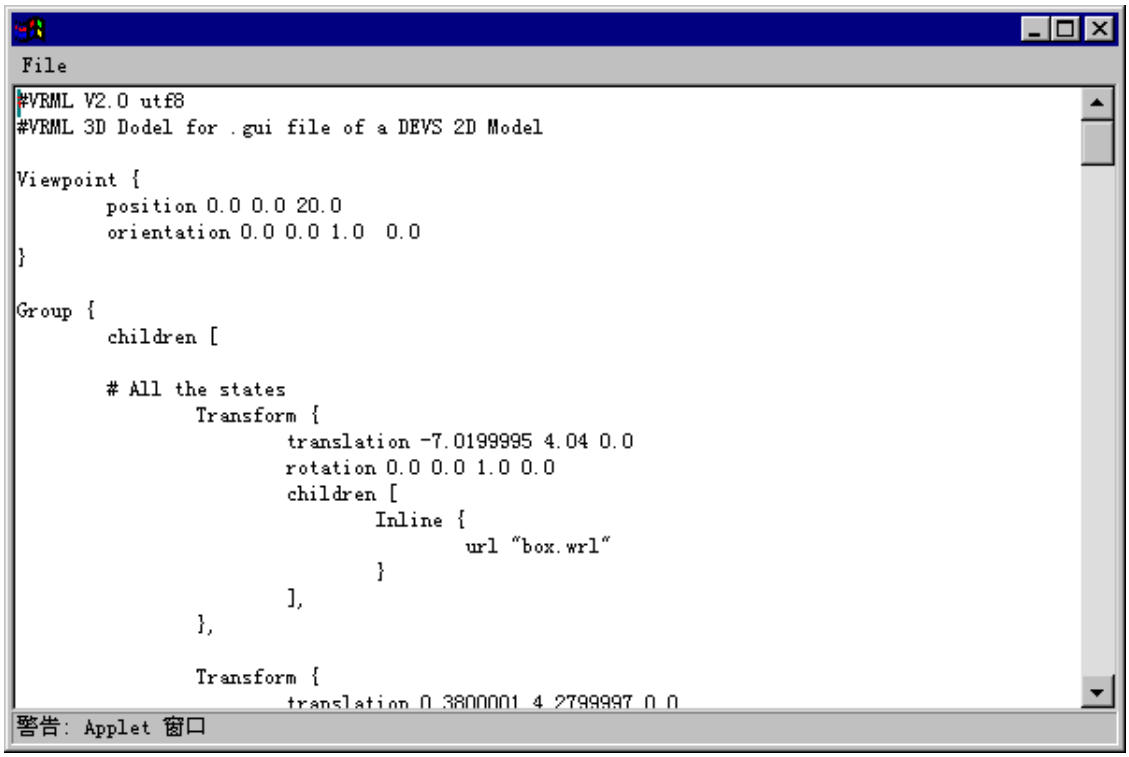


Figure 6.15 3D VRML model file (states/nodes)

In Figure 6.15, the VRML file head and a viewpoint node are added. The nodes in the scene are shown as Inline nodes, and their positions are calculated with the algorithms mentioned in section 5.3. The link transformation is shown in Figure 6.16. It is

transformed as a long cylinder calculated with the algorithms mentioned before, and for each cylinder, a cone is added to indicate the direction of the message transformation between the components. Their positions and orientations are also calculated with the algorithms mentioned in section 5.3. This VRML file can be saved and used as an Inline node in the new model input later.

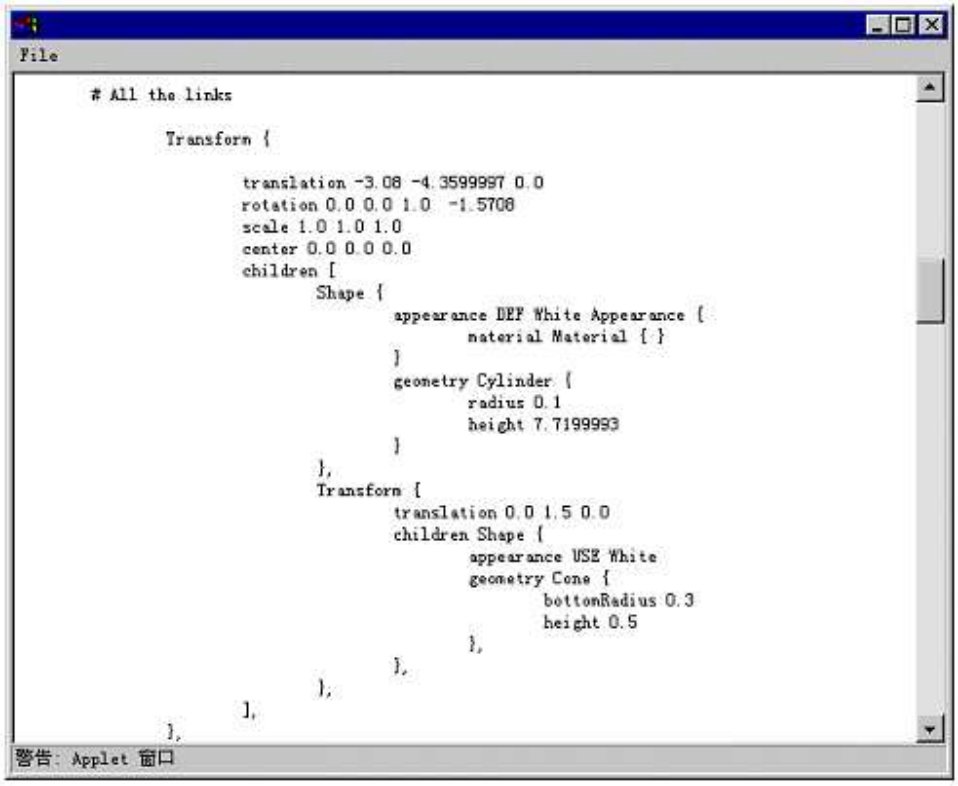


Figure 6.16 3D VRML model File (links)

Chapter 7 Conclusion and Future Work

7.1 Conclusion

Simulation is becoming increasingly important in the analysis and design of complex systems. DEVS is a formalism for modeling and simulation gaining popularity in recent years and has found a variety of applications. CD++ implements the DEVS formalism and can be used to simulate DEVS and Cell-DEVS models. CD++ is a tool that can simulate complex physical systems, and can be used to simulate a variety of models.

Visualization is also very important in modeling and simulation, it uses simulation results to construct useful 2D or 3D images. Visualization tools are crucial in helping to better understand the behavior of complex systems. Now visualization has become an integral part in modeling and simulation. Effective simulation tools must include good corresponding visualization tools.

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 the CD++ Modeler, 2D and 3D result visualization tools, and an Interface for remote simulation model execution. All these tools provide enough functions to be organized as a remote simulation environment in the CD++ simulation.

The client is implemented with Java and VRML. Java is the most popular object-oriented

programming language, and VRML is a Web-based graphics language for building 3D models. Therefore, the client can run on a variety of operating systems and environments.

The CD++ Modeler can be used to build atomic and coupled models, which can be executed with the CD++ simulator. It uses graph-based notation to represent the entities in the real system, and the relations between these entities. This facility has highly enhanced previously existing means for model definition in CD++ by defining the models with graph units, not using text edition tools as the previous ones.

A 2D visualization tool can be used to visualize the results of executing atomic models, Cell-DEVS models and Coupled DEVS models. The 2D visualization provides a very simple method for understanding model execution, however the user must understand intellectually the relationships between several 2D raster images and compare them to figure out how the real system looks like. Therefore, a 3D visualization tool also has been developed, it can display the simulation results in a 3D format.

The 3D visualization GUI enables sophisticated visualization of Cell-DEVS models. To better understand the results, the user can select different shapes to represent a node in the 3D space, select different color classifications, or hide some nodes with specific values, edit individual node and remove layers. It also allows the users to navigate in the visualization with many ways he likes.

The Interface in this client can send simulation models to a remote CD++ server, then

receive, and visualize the results locally. The user can access any remote CD++ simulation server in the Internet/Intranet around the world for simulation services.

As indicated in Chapter 2, a number of recent efforts have been devoted to build DEVS models and cellular models. For CA tools, some are only for 1D or 2D models. They were designed as an applet or application, in which the users can change the size of the model and the color palette map, specify some characters of the model, select many existing rules or try their own rules, then initiate and run the model. Some enable 3D visualization of the executing cells, but they do not have the 3D visualization abilities we need. When we consider DEVS tools, we see that some of them have not visualization facilities, and the users have to develop themselves. Some provide some basic visualization tools, such as, SimBeams and JDEVS. SimBeams provides a set of visible components, and the users should select these components to build their own visualization tools for their specific models. JDEVS provides sophisticated visualization tool, but it is mainly for GIS, and only displays the data with the outside entities. DEVS/Java provides some interface to visualize the state of the components in the models although no powerful visualization methods are included, and can execute the model in a web browser, but does not provide client/server facilities. DEVS/HLA discusses the architecture about how to build large-scale distributed modeling and simulation environments, and it should be implemented with a programming language.

As illustrated in this thesis, we attacked and solved these problems, and we developed four main components for a remote simulation and visualization system. The CD++

Modeler provides an easy way to input model with graph units. The 2D visualization GUI provides a very simple method for understanding DEVS model execution with various navigation methods, and the users can control many important visualization parameters for better understanding the results. The 3D visualization GUI enables sophisticated visualization of 3D Cell-DEVS models with various navigation methods. The users can get into the inside of the model, edit the 3D VRML scene, even individual node for better understanding the results. The Interface provides an easy way to connect remote server, and can connect different users in many places together.

Users can easily build the models and visualize the results locally on basic workstations with 3D visualization GUIs, while executing the models remotely in a high performance platform anywhere in the world. With this client, a remote simulation system can be easily established in various environments (such as, Windows, Unix, etc). In addition, this client also can run several different models simultaneously and easy to be extended to support multi-observer simulation. A series of examples were executed to demonstrate the feasibility of this approach. All of the research goals stated in section 2.5 have been reached.

7.2 Future Work

The facilities introduced in this thesis have significantly improved the user interaction in comparison to the previously existing tools, thus facilitating the users to build model and analyze the simulation results. There are still many ways to improve this client. The

following is a proposed list of improvement and extensions.

- Model Consistency Checking

  It is very important to ensure that the model design conforms to the DEVS formalism. Although the CD++ Modeler can check the model consistency with the rules in section 4.1 when the design graph is exported as a model file, it is more convenient to check the model consistency during the model input. There are two methods to do this: a) check model consistency once a new link has been added to the new model, b) provide port choices for the users when a new link is added to ensure that it starts from an output port and ends at an input port.

- Model Execution Exception Handling

  When CD++ works as a server, the model is executed on the remote server. However, when an error occurs and the simulation execution stops, the exception messages are only displayed on the server and no any reply are sent back to the user. It would be more convenient that the server checks the model file for consistency first before execution, and if there is any error in the model file, the server can send back a message and indicate the error. The server also should send back the exception messages to the clients and indicate the errors in model execution.

- Visualizing Model Identification

  After the viewing area has been changed, or some nodes have been deleted in the 3D VRML scene. The user may be confused by the new display and finds it difficult to identify the nodes, that is, the node-value mapping again. Therefore, new methods should be developed, and the user can choose how to visualize the result to facilitate

the model identification.

- Expanding the VRML Library

  More Inline files and images for texture feature should be found or designed to represent the objects in the real system. With these files and images, two libraries should be built and organized as databases to facilitate the management and selection of the Inline nodes and images.

- Networking improvement

  Recent networking is point-to-point communication between the user and the server, that is, just the user who sends model to server can receive the result. In the multi-observer simulation situation, many users should receive the same results at the same time. Instead of being implemented in client side as in this these, it is more convenient to be implemented in server side and the model sender just specifies all the receivers of the results when he sends the model.

- Optimization Method for the Modeler

  If the model is complex, there will be many nodes and links in the design area. It would be very hard to identify the nodes and links in the model graph because these nodes, links and their names overlap each other. Therefore, it is necessary to use optimization algorithms, such as, Simulated Annealing (SA) [21], to optimize the arrangement of all these noses and links. The user can select several goals for the optimization, such as, minimizing the number of the link-link crossings.

- Use new technologies, such as, XML (eXtensible Markup Language) [8] and X3D (Extensible 3D) [37]

XML: over the last few years the Web has evolved from HTML quite dramatically with revolutionary techniques for content and structural modeling, such as, XML. Compared to HTML, the content of XML documents is enriched with semantic and structural features, and is completely separated from its visual appearance. This allows a Web document to be displayed in any desired form. With such an unrestricted choice, many companies and end users prefer a graphically rich document appearance with effective visual access to semantic and structural information.

X3D: a next-generation, extensible, 3D graphics specification that extends the capabilities of VRML97. X3D is building upon the success of the VRML 97 ISO standard with clearly defined backward compatibility with existing VRML content. Now X3D is a next-generation 3D standard that includes integration with XML, and defined as an interoperable set of lightweight, composable 3D standards that flexibly address the needs of a wide range of markets, including Internet and road cast applications.

References

[1] “3D Cellular Automata”, *Cellular Automata Simulation Artificial Life, Inc*. 2001. http://www.artificial-life.com/v5/demos/geneticode/

[2] 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.

[3] AMEGHINO, J.; WAINER, G. "Application of the Cell-DEVS paradigm using N-CD++". In *Proceedings of the 32nd SCS Summer Computer Simulation Conference*. Vancouver, Canada. 2000.

[4] AMES, A.; NADEAU, D.; MORELAND, J. “VRML 2.0 Sourcebook" (Second Edition). *John Wiley & Sons, Inc*. 1997.

[5] BAY, C. “The Game of Three Dimensional Life”, Department of Computer Science and Engineering, University of South Carolina. http://www.cse.sc.edu/~bays/d4d4d4/

[6] CHEN, W.H.; WAINER, G. “An introduction to VRML”, on-line report, Department of Systems and Computer Engineering, Carleton University, 2003

[7] CHEN, W.H.; WAINER, G. “A Tool for Remote Execution and 3D Visualization of Cell-DEVS Models (User Manuals)”, Technical Report no. SCE 03-10, Department of Systems and Computer Engineering, Carleton University, 2003

[8] CHRIS R, M.; ERIK T, R. “Learning XML”, *O'Reilly & Associates*, 2001.

[9] 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.

[10] DRAGICA, W. "Project Description - Task E: Visualization", Department of Electrical Engineering, Arizona State University, 1999
http://www.geekspiff.com/academics/nsf_research/TaskE.php

[11] 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.

[12] DAVILA, J.; UZCAGEGUI, M. "GALATEA: A multi-agent, simulation platform", Proceedings of MSNN'2000: *International Conference on Modeling, Simulation and Neural Networks*, 2000.

[13] ELLIOTT, J. M. G. "Cellsprings" On-line document:
http://jmge.net/java/csprings/doc/

[14] 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.

[15] FILIPPI, J.; BISGAMBIGLIA, P.; DELHOM, M. "Neuro-DEVS, an hybrid environment to describe complex system", *Proceedings of ESS'2001 13TH European Simulation Symposium and Exhibition*. Marseille, France. 2001.

[16] FILIPPI, J.B.; BERNARDI, F. DELHOM, M. "The JDEVS environmental modeling and simulation environment" *IEMSS 2002 conference on Integrated Assessment and Decision Support*. Lugano, Switzerland. 2002.

[17] GUARRACI, B.; POTTER, J.; RITZ, B.; WINTER, D. “Modeling Grid Cellular Automata in 3-D”, Department of Mathematics/Computer Science, Salisbury State University, Salisbury, MD 21801 http://faculty.ssu.edu/~mjbardze/dan_vrml/index.html

[18] GARDNER, M. "The fantastic combinations of John Conway's New Solitaire Game 'Life'". *Scientific American*. 23 (4). pp. 120-123. April 1970.

[19] HARTMAN, J and WERNECKE, J. "The VRML 2.0 Handbook", *Addison Wesley*, 1997.

[20] KRAAK, M.-J.; MACEACHREN, A.M. "Visualization for exploration of spatial data". *International Journal of Geographical Information Science*, 13(4), 285-287. 1999.

[21] LAARHOVEN, P.J.M.; ARTS, E.H.L. “Simulated Annealing: Theory and Applications”, *Reidel, Dordrecht*, 1987.

[22] MACKINLAY, J.; ROBERTSON, G.; CARD, S. "Rapid Controlled Movement Through a Virtual 3D Workspace", *Computer Graphics* 24(4), 1990.

[23] MARRIN, C. “External Authoring Interface Reference”, *Silicon Graphics, Inc.*, 1997. http://www.web3d.org/WorkingGroups/vrml-eai/history/eai_draft.html

[24] 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.

[25] NUTARO, J. "*Adevs*: User manual and API documentation". On-line document, http://www.ece.arizona.edu/~nutaro/adevs-docs/index.html.

[26] 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 Modelling & Simulation, San Francisco, California, January 17 -20, 1999.

[27] RUCKER, R.; OSTROV, D. “Continuous-Valued Cellular Automata for Non-Linear Wave Equations”, *Complex Systems* 10 (1196) 91-119, 1997.

[28] SARJOUGHIAN, H.; ZEIGLER, B. "DEVSJava: Basis for a DEVS-based Collaborative M&S Environment". *Proceedings of 1998 international conference on web-based modeling & simulation*. San Diego, California. 1998.

[29] 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*. 1994.

[30] 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.

[31] TOLLERTON, R.; PETROVA, M.; SANGAPPA, S. "3D-Life visualization". Department of Computer Engineering and Computer Science, University of Missouri-Columbia
http://meru.rnet.missouri.edu/courses/cecs361/projects/ws99/3d_life/web_pages/catalog.html

[32] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modeling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", *Springer-Verlag*. 2001.

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

[34] WAINER, G. “CD++ User’s Guide (Parallel Version)”, Department of Systems and Computer Engineering, Carleton University, 2001

[35] WAINER, G.; GIAMBIASI, N. "Application of the Cell-DEVS paradigm for cell spaces modeling and simulation.". In *Simulation*; Volume 76, Number 1. January 2001.

[36] WATSON S.-H. “3D Visualization of Software Architectures”. M.Sc. Thesis, Department of Systems and Computer Engineering, Carleton University, 1996

[37] WEB3D CONSORTIUM. "Extensible 3D (X3D) International Draft Standards". http://www.web3d.org/fs_specifications.htm

[38] WU, X.; WANG, Y. “The CD++ Modeler”, M.Sc. project report, Department of Computer Science, Carleton University, 2001.

[39] WU, X.; WANG, Y; CHEN, W.H.; WAINER, G. “User Manual for the CD++ Modeler”, Technical Report no. SCE 03-11, Department of Systems and Computer Engineering, Carleton University, 2003

[40] WOJTOWICZ, J. “1D and 2D Cellular Automata explorer”. On-line documentation: http://www.mirwoj.opus.chelm.pl/ca/mjcell/mjcell.html

[41] ZEIGLER, B. "Theory of modeling and simulation". *Wiley*, 1976.

[42] ZEIGLER, B.; KIM, T.; PRAEHOFER, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". *Academic Press*. 2000.

[43] 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.

[44] ZEIGLER, B.; MOON, Y.; KIM, D.; KIM, J. "DEVS-C++: A High Performance Modeling and Simulation Environment: . *Proceedings of HICSS*, 1996.

[45] 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.