CD++

A tool for and DEVS and Cell-DEVS Modelling and Simulation

User's Guide

DRAFT – AUGUST 2004

Gabriel A. Wainer

Wenhong Chen, Juan Ignacio Cidre, Ezequiel Glinsky, Steve Leon, Ali Monadi, Alejandro Troccoli

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Dr. Ottawa, ON. Canada

http://www.sce.carleton.ca/faculty/wainer

gwainer@sce.carleton.ca

Table of Contents

CD++

CD++ is a tool for Discrete-Event modeling and simulation, based on the DEVS formalism. It runs either in standalone (single CPU) or parallel mode (over a network of machines). This document is a user's guide to CD++, and we will only focus on tool-related aspects. If needed, the reader can refer to the following references for better understanding of DEVS and Cell-DEVS related topics (available at http://www.sce.carleton.ca/faculty/wainer):

"A framework for remote execution and visualization of Cell-DEVS models". G. Wainer, W. Chen. In Simulation: Transactions of the Society for Modeling and Simulation International. November 2003. pp. 626-647.

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

"N-Dimensional Cell-DEVS". G. Wainer, N. Giambiasi. In Discrete Events Systems: Theory and Applications, Kluwer. Vol. 12, No. 1. January 2002. pp. 135-157.

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

DEVS is a discrete event paradigm that allows a hierarchical and modular description of the models. Each DEVS model can be behavioral (atomic) or structural (coupled), consisting of inputs, outputs, state variables, and functions to compute the next states and outputs. Cell-DEVS modeling systems that can be represented as executable cell spaces. The DEVS formalism is used to provide enhanced execution speed.. For more information about DEVS and Cell-DEVS models please refer to: http://www.sce.carleton.ca/faculty/wainer/celldevs/introduction.html. From now on, a complete understanding of DEVS and Cell-DEVS models is assumed. Details about the DEVS formalism can be found in:

"Theory of Modeling and Simulation". B. Zeigler, H. Praehofer, T. G. Kim. 2nd Edition. Academic Press. 2000.

1.1 CD++Builder – Quick Reference Guide

This section is a quick reference for CD++Builder. It will explain how to run a number of simple examples without providing detailed information about the tools. The goal is to allow users to develop a basic familiarity with the tool and it's functionality. The details of the tool be presented in the coming sections.

Most of the functions of CD++ can also be accessed through line commands but in this section we assume you will use CD++Builder, an Eclipse-based GUI for CD++. Eclipse is a workbench that can be used for any form of software development. It is an open-source integrated development environment (IDE) that can be extended through the definition of plugins. The desired functionality can be programmed as plugins by combining them with existing features of the platform. Task menus, notes, error logs, multiple project operations and even drag and drop tools/functionalities are included. For a more detailed description of Eclipse, visit http://www.eclipse.org/eclipse/faq/eclipse-faq.html.

CD++Builder is a plugin for Eclipse that provides the users with a CD++ development environment to create, edit or view CD++ simulation projects. The tool, implemented with the Eclipse workbench, provide an IDE where one can create, open and save projects. It enables editing CD++ related files and support for developing multiple projects.

CD++Builder is essentially a front-end to CD++, a toolkit for DEVS and Cell DEVS modeling and simulation

(http://www.sce.carleton.ca/faculty/wainer/celldevs/). CD++Builder uses Eclipse and its platform plugin development to provide an easy and simple environment to use CD++. Features such as a coupling syntax editor, C++ editing support, importing and exporting data, and a graphical user interface for the CD++ tools are featured in the plugin. However, Eclipse has many of it's own tools to make any user comfortable to developing anything on it's workbench.

A key aspect in CD++Builder is the Coupling Editor, which provides a user means to manipulate DEVS coupled models files. The editor provides a syntax coloring scheme which colors keywords, variables and comments. Basic editor functions are provided as well such as CUA compliant keyboard commands or copy and pasting functionality. This Editor also supports XML files for coupled model definition. XML coupled model files can be converted into CD++ coupled model files for use in a project.

C++ support is featured to allow users to edit C++ related files involved with a CD++ project. Creating, viewing and editing any sort of C++ files is done within a syntax editor with a coloring scheme. (C++ editing support is featured from the CDT project, http://www.eclipse.org/cdt/)

The CD++Builder packages all the potential tools that can be used in a CD++ project from the existing CD++ toolbox and the Eclipse workbench and its own tool set. A perspective in eclipse defines a set of editors and views arranged in an initial layout for a particular role or task. A default perspective for CD++Builder is given to support the CD++ environment. The CD++Builder must be installed to view its features (please refer to Appendix A – Installation and Technical notes).

If you start Eclipse, you will have access to the tools provided by CD++Builder. You will see a window like the following:

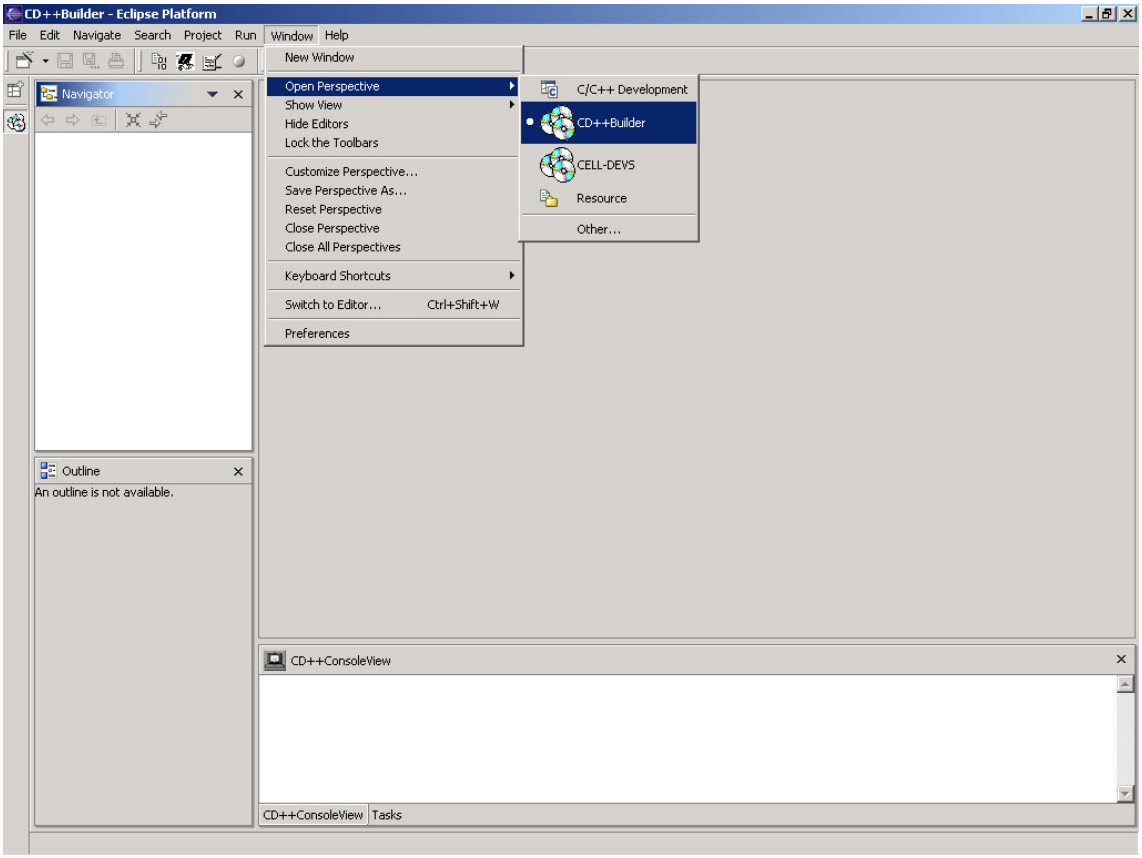


Figure 1. CD++Builder main window.

To open the CD++Builder perspective, select *windows -> Open perspective -> Other....* The following panel will appear.

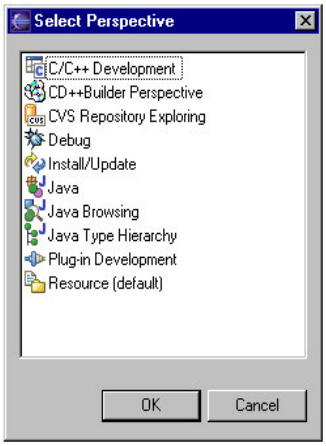


Figure 2. Perspective panel

Figure 2 displays all perspectives available within Eclipse. To activate the plug-in select *CD++Builder Perspective* and click *ok.* When this option is selected, CD++Builder is activated, as in the following figure.

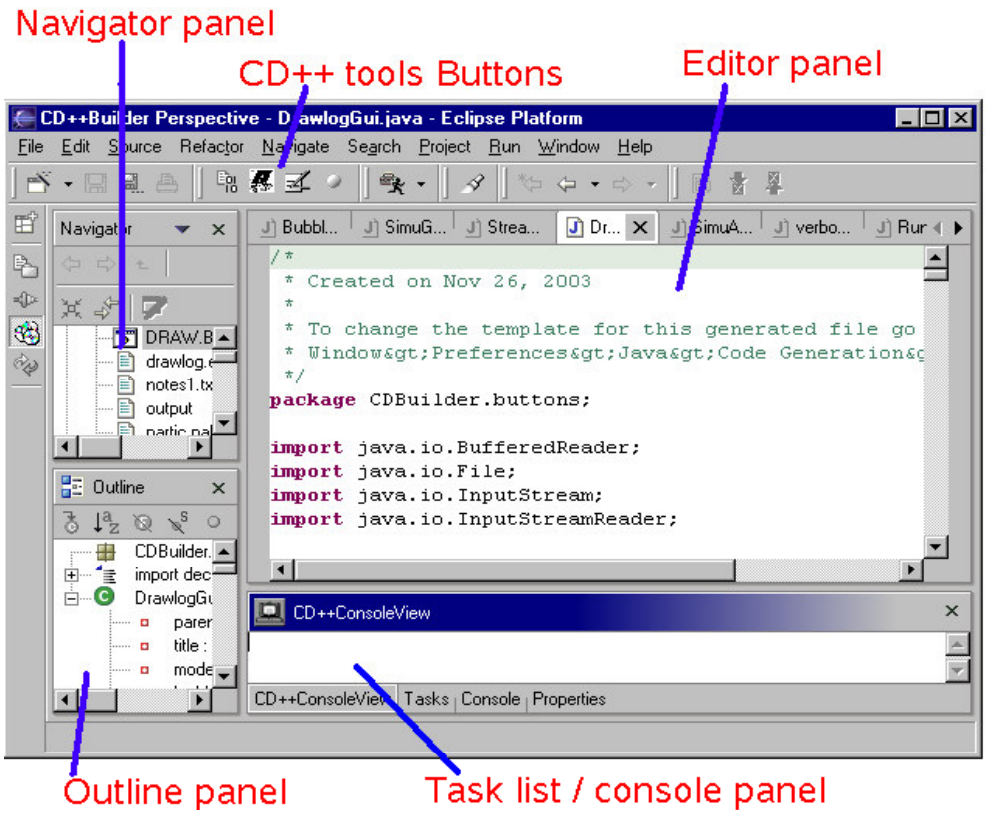


Figure 3. CD++Builder.

- **Navigator Panel:** allows user to view the current projects and their contents.
- **Editor panel:** allows user to view the contents of a selected file. It is programmed to open the default editor for that particular file.
- **Task list/Console panel:** a section to write down planned tasks for particular project. The task view also shows the errors encountered when compiling the project. The console will display any errors encountered

while running a CD++ function and output from CD++ tools.

- **Outline panel:** outlines the functions and objects in a selected class file. This portion is only implemented for C++ files.

The CD++ tool set has a variety of components to execute DEVS models and to analyze simulation results. Our main perspective, called CD++Builder, integrates the major as buttons located in the top toolbar. There are currently four buttons available:

- **Build:** This button automatically creates a makefile for a specific project and runs the *make* command to compile the source code for the models. The result is an executable to run a simulation.
- **Simulation:** This button activates the CD++ simulator. This executable represents a project-specific simulation program that will simulate what you are modeling.
- **Drawlog:** This button generates a (.)drw file for easier visualization of the execution of a Cell-DEVS model in a text file.
- **CD++Modeler:** This button loads the CD++ Modeler program, a graphical tool for designing and executing DEVS and Cell-DEVS models. In this application you can design atomic and coupled models as well as animate the simulation results.

In this section, we will illustrate how to use the toolkit by executing two previously existing models. The first one, called *life*, is a Cell-DEVS version of the "Life" Game. The second example is called *ATM*, which is a DEVS model representing an ATM machine. These example (and other existing models) can be downloaded from http://www.sce.carleton.ca/faculty/wainer/wbgraf. The model examples are located at the left of the web frame. To download an example, click on the link indicated by: "*Download model and sample*". All the examples are compressed in zip files. Download the two samples into a local directory.

The first step to start a simulation is to create a new project. Open Eclipse and make sure that the CD++Builder perspective is open as explained earlier. Click on the "*File*" tab, select "*New*", and click on "*Project...*". This will bring up a new project wizard panel, which is shown in the following figure. Select the "*Other*", and on the left side of the project wizard panel, select "*CD++Builder Project Wizard*". Click "*Next*" to start the creation of a CD++Builder project.

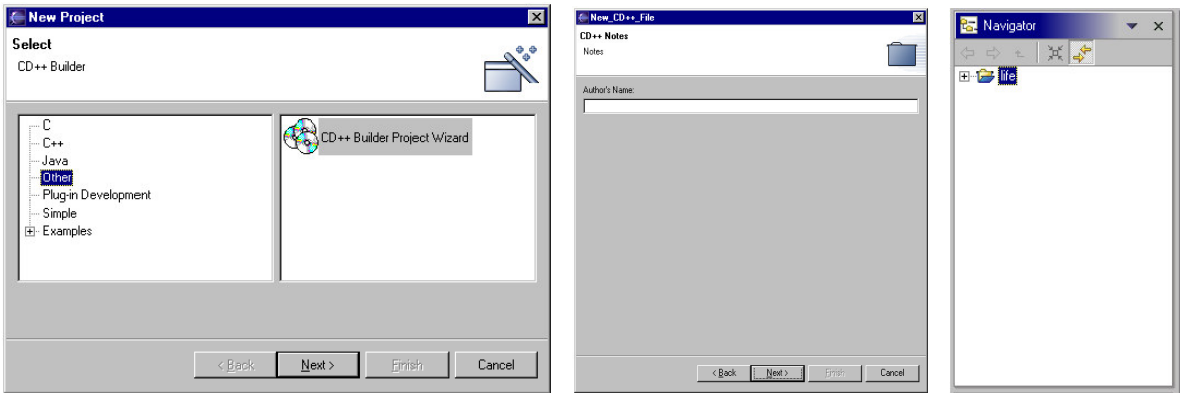


Figure 4. (a) New project wizard panel; (b) CD++Builder Project Wizard; (c) Navigator.

At this stage, the CD++Builder Project Wizard will be opened and the author name will be asked, as showed in Figure 4 (a). Enter the project author's name and click *Next.* The wizard will ask for the project name. Enter `life` and click on *Finish.* The newly created project can be seen in navigator view, as illustrated in Figure 4 (c).

After having created a new project, the next step is to add the `life` example model to the project. Select the project on the navigator view and right click on it. On the menu, select "*Import...*". This will open the panel shown in figure 6.

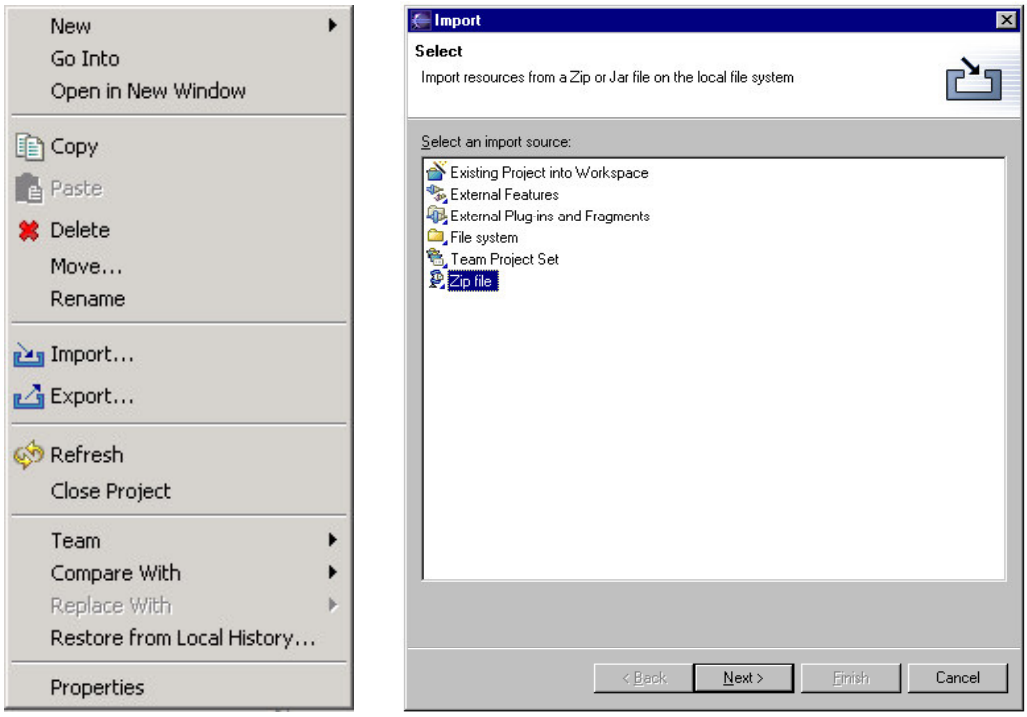


Figure 5. Import panel windows.

Since the `Life` example model is compressed in a Zip file, select *Zip file* and click *Next*. Another panel will open asking to enter the name of the zip file to be imported. To locate the zip file click on the *Browse...* button located on the top right of the panel, open the `life.zip` model and click *Finish*.

Figure 6. Import panel windows.

Figure 8 shows the navigator view, where you will see the new life example model that was added to the project. We will now simulate this model. The first step is to select any file from the folder by clicking on the `life` file (THIS STEP IS REQUIRED; as there are different folder for different projects, we must pick the one we want to run). Then click on the *simulation* button  and the following panel will open.

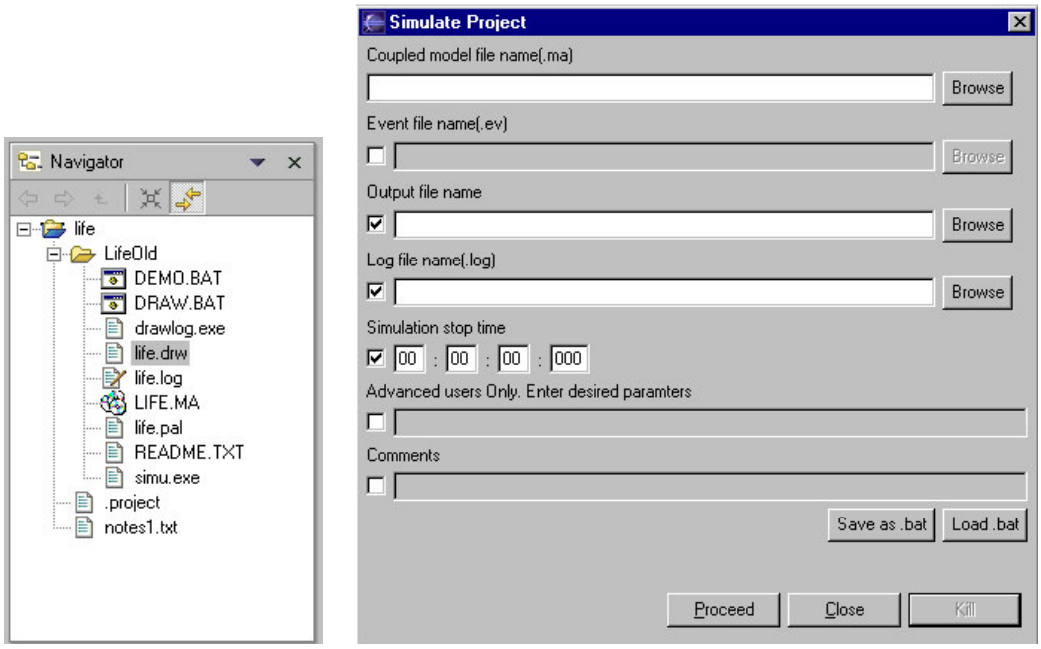


Figure 7. (a) Navigator view; (b) Simulation panel.

There are different ways to run a model, which will be discussed later. Here, we will use a previously defined script. To do so, we must click on the "`Load .bat`" button, and open the file `demo.bat` (included in the `life.zip` you originally downloaded). This will fill the panel with the necessary parameters to run the simulation. Click on *Proceed*. At this stage, the console view will show all the details of the simulation (Figure 8). To see the log file created from this simulation, double click on `life.log` from the navigator view. This will open a editor view showing the content of the log file.

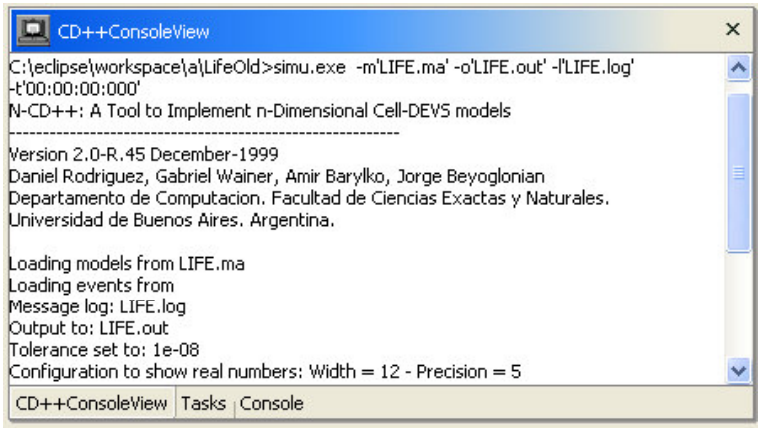


Figure 8. CD++ Console View

If you open the *.log file created by the simulator, you are able to view the simulation results, as in the following figure.

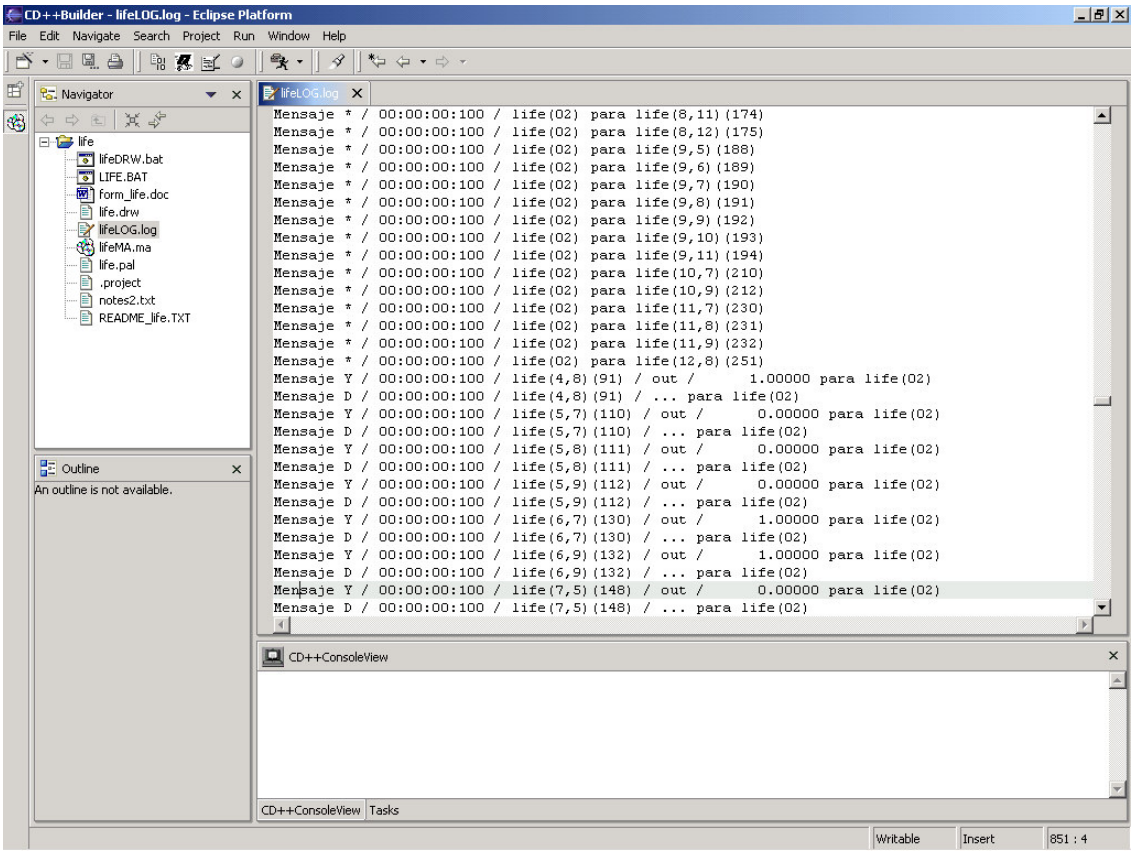


Figure 9. CD++ log outputs.

To view the state of a Cell-DEVS model, we can use the *Drawlog* tool, that permits viewing the outputs in a simpler way. To use this feature, click on the *Drawlog* button  . The following panel will open:

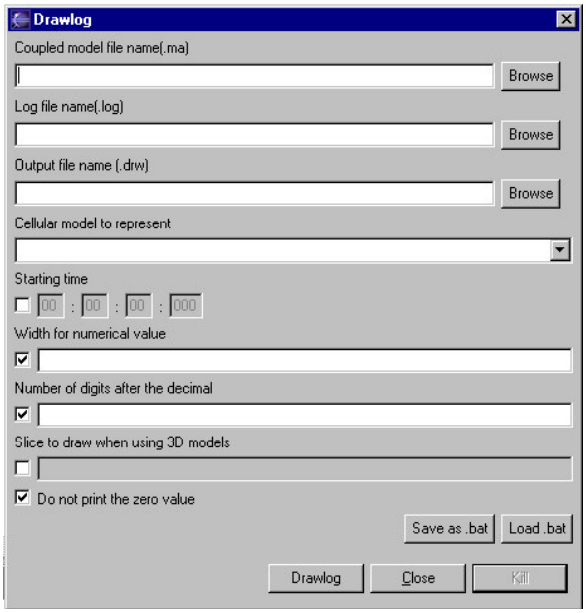


Figure 10. Drawlog panel

Here, we will also use a previously defined script. Click on the "`Load .bat`" button, and open the file `draw.bat` (included in the `life.zip` you originally downloaded). This will fill the panel with the necessary parameters to run the drawlog. Click on *drawlog*. The CD++ console view will once again show the state of the drawlog creation. Once the conversion finishes, a text file will be created and opened illustrating the state of the simulation, as in the following figure.

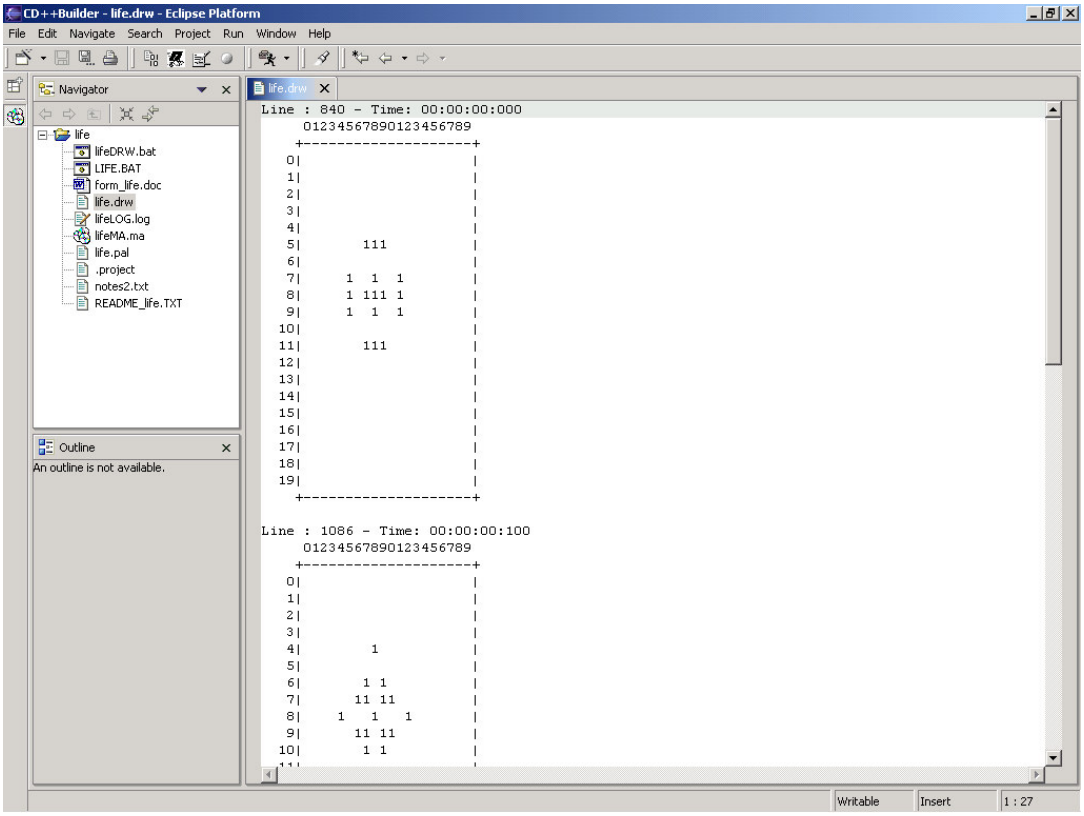


Figure 11. *.drw file

To simulate our second example create a new project following the previous steps and name it *ATM*. Once the project is created, import the example to the project. Since the ATM model is a DEVS model, it needs to be compiled before simulating it. To compile the project, click on one of the files in the project within navigator view (in order to pick the project we want to compile) and press the *Build* button. A new panel, shown in figure 12, will appear asking the user if they would like to run in verbose mode (a mode that provides detailed info of the compiling project during compilation).

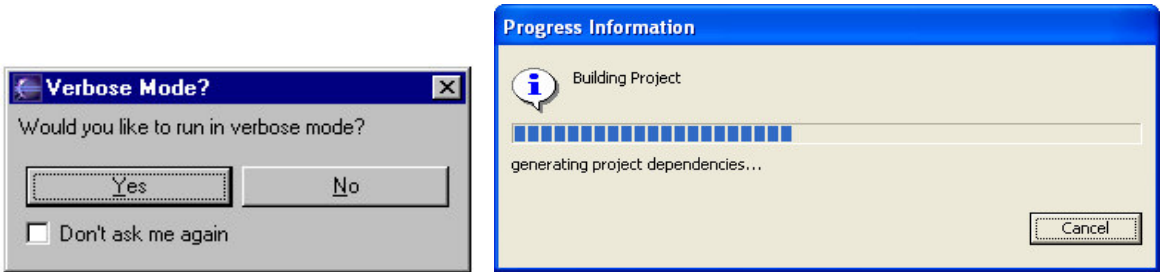


Figure 12. (a) Verbose mode panel (b) Progress dialog

Once you have selected the mode, compilation will start. A progress dialog will appear showing the progress.

Also, the console view will show detail of the compilation and will inform you if the compilation was successful. If there are errors, you can see them on the console, as showed in the following figure. Once the compilation is successful, we can start simulating the project.

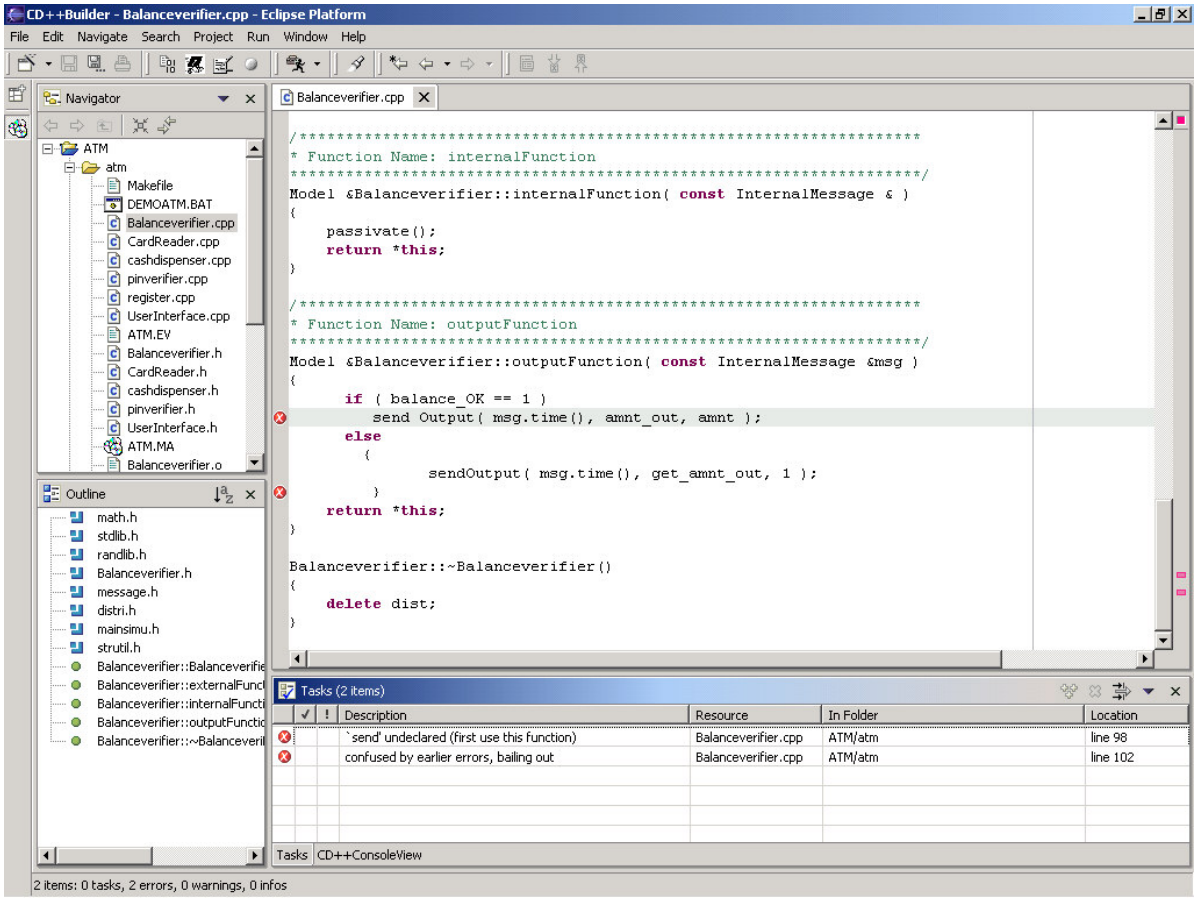


Figure 13. Error display.

To simulate the project, we can follow the same steps previously mentioned (except for the use of the drawlog, which only applies to Cell-DEVS models).

Further instruction and information about simulating a model are explained in section <>.

2 Model definition: Atomic models

This section describes the mechanism to defines and incorporate new atomic models into CD++. These models can be used to interact directly with other models or to be part of a DEVS coupled model. Atomic models are added to the tool at compile time, and if a new atomic models need to be defined; they must be coded in C++ and incorporated into CD++ model hierarchy.

2.1 Creating new atomic models using CD++Builder

Eclipse provides a set of tools and plug-ins which facilitate the creation of a model. This section will provides useful features that Eclipse delivers to create of a new atomic model.

2.1.1 Opening an existing project

To open an existing project into your workspace, one will have to import it by using the import wizard, which is illustrated in Figure 14. This wizard is opened by selecting *File* on the top window bar in Eclipse and then clicking on *import*. Then select the *Existing Project into Workspace* option and click on the *next* button. The next screen should prompt you to select the folder in which the existing project is located. Once you have selected where the project is, you may click on the *finish* button to import the project. This project will now be in your workspace/navigator for you to edit or view.

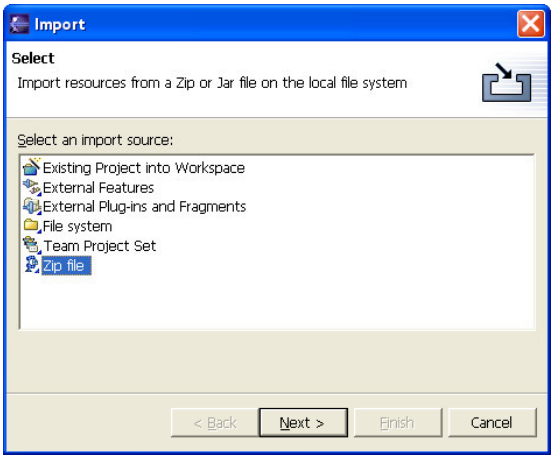


Figure 14. Import wizard panel & Navigator view respectively

2.1.2 Creating a new project

To create a new project, click on the *File* tab, select *new*, and then click on *project*. This will bring up a new project wizard panel, where you can create different types of projects, as shown in the following figure.

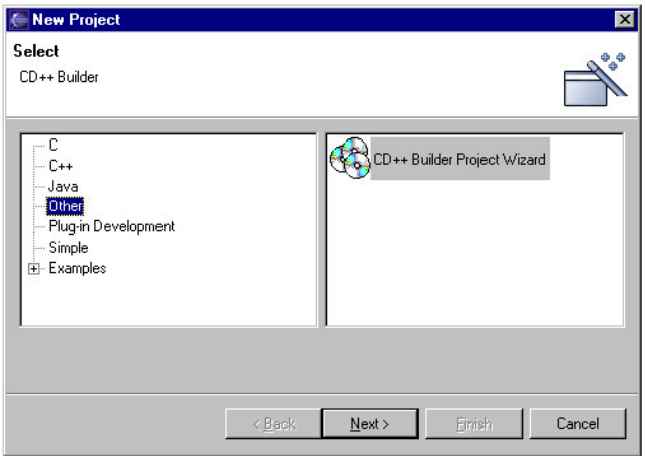


Figure 15. New project wizard panel

The left side of the panel shows the different types of project that can be created, while the right side shows the different project wizards corresponding to each type of project selected on the left side.

The CD++Builder Project Wizard is located on the “Other” section. After selecting the project, a new panel will ask for the author's and the project's names. Once these are filled, select *finish* to finalize the creation of the project or click *next* to make references to existing projects inside your workspace. In this way, you can include models already defined for other projects in the newly created one.

When the project has been created, we are able to access a *notes* file and a *project* file. The *notes* file is a simple text file that you can store short notes about your project. The *project* file is a file which acts as an identification file for the project.

The new project and its files are now available in the navigator view. The following figure illustrated the creation of a new project named “Clock”.

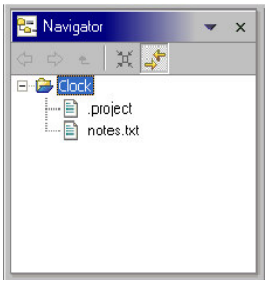


Figure 16. New project shown in the navigator view

2.1.3 Creating and editing files

To create a new atomic or coupled model, we need to created and edit a new file. To create a file, go to *File* -> *New* -> *File*. This will bring us to a new file wizard, which will request information about the type and location of the file as illustrated in the figure below:

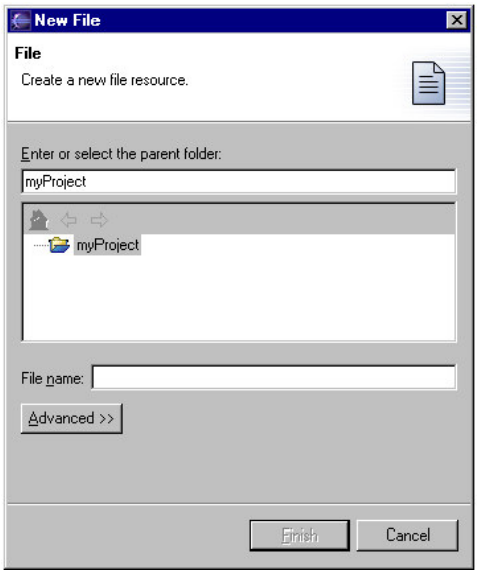


Figure 17. New file wizard panel

Highlight the project of where you want the file created and then enter the name in the bottom panel. Before you click on finish you must enter the extension of the file you are creating. For example if you are creating a C++ file that does DEVS modeling, you must end the file in “.cpp”. As soon as you select finish, the file is created and already opened in the editor panel. Figure 18 illustrates the creation of a new file called *newFile*.

Eclipse has a “smart editing” functionality that will open the appropriate editor in the editor panel. When you select any file in your project for viewing or editing, it opens it in the editor panel to the affiliated editor. C++ files will be opened in CDT (Eclipse' C++ Editor), text files will be opened in a basic text editor etc. If you edit a file you can simply hit the save button in the top left corner, if you feel you made a mistake, you can close the file and select “no” to not save it at all.

In Figure 18, the file that we just created, *newFile,* is shown in the navigator view and it is also opened for editing.

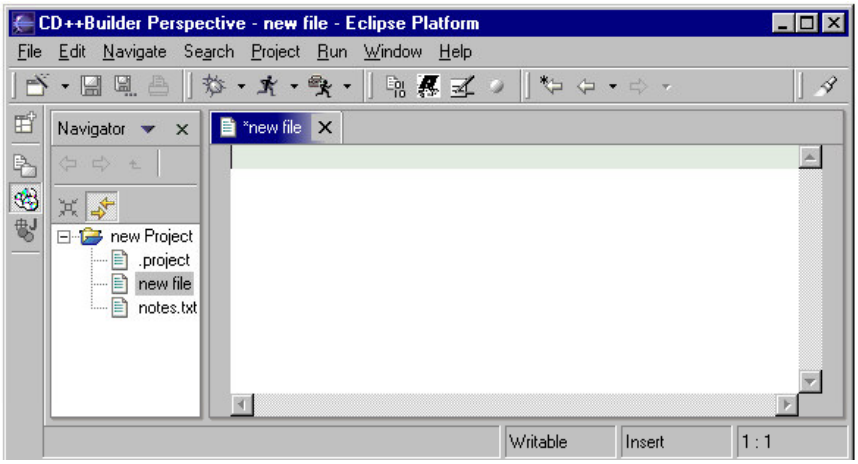


Figure 18. CD++Builder with text editor opened.

2.1.4 Adding Files to projects

Files can be added to a project in multiple ways. One option is to drag and drop files from any window outside of eclipse into your project (located in the navigator), which automatically copies that file into the project.

Another method is to use the importing function. Select *File* and then click *import.* This will bring up the importing panel shown in Figure 19.

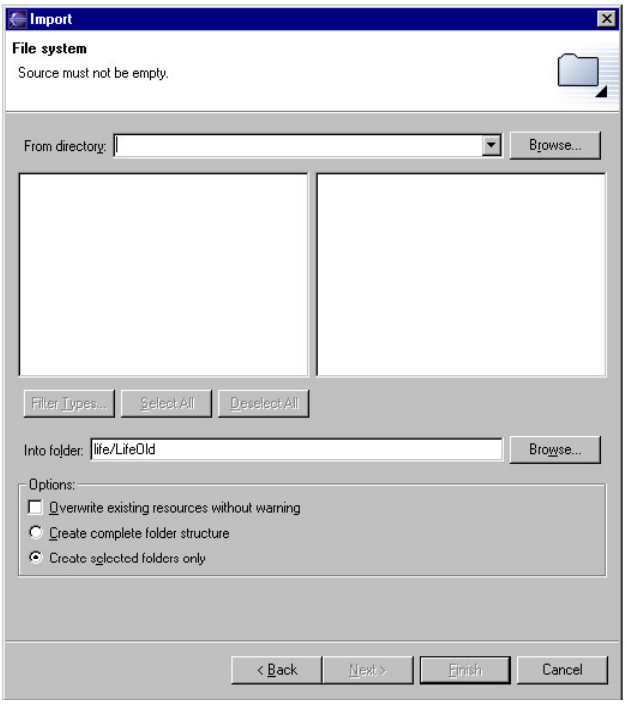


Figure 19. File System Import Panel

Here, select *file system* and then click *next*. The panel shown in Figure 20 will appear. On this screen you must specify the directory of the files you want to add by pressing the *Browse* button that is located at the top of the panel. Then by checking off the files you want to import, Eclipse will copy them into your project folder.

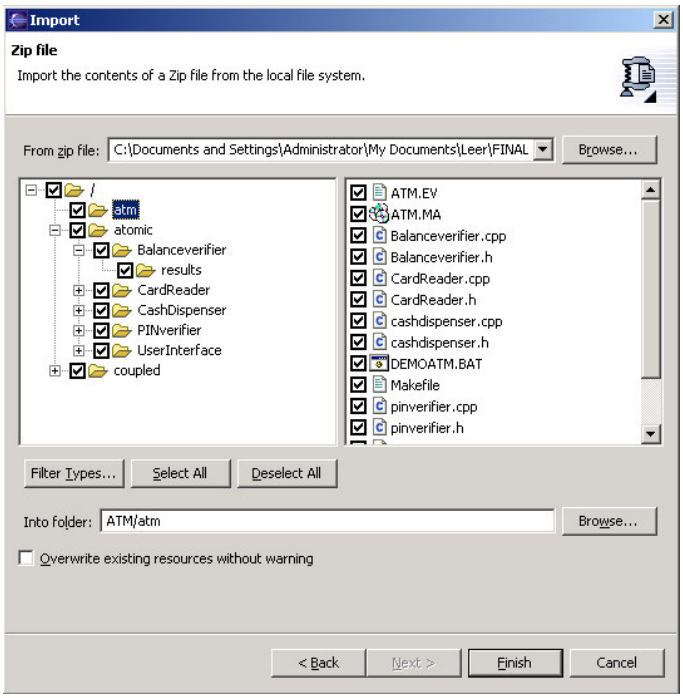


Figure 20. File System Import Panel

One can also import files in the form of a zip file. On the Import Wizard panel, select "*Zip File*" and specify the zip file.

2.1.5 Using the Navigator

The Navigator (the panel on the left side), is a component that enables you to view multiple projects and their indexes of all the files in each project. The format for viewing these files are in tree-fashion as shown in figure 25. You can open the index of each project and view the files or folders in them. Opening a folder inside the project will expand that section to show its contents. You can move or copy files from project to project by using the drag and drop option or the import wizard option.

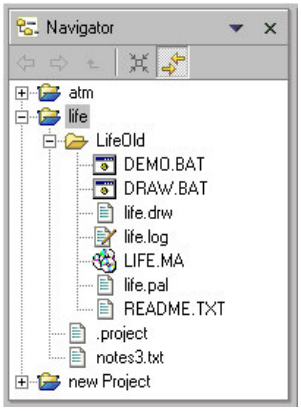


Figure 21. Navigator view

The navigator can also close projects down that you may not want to view or use, but keeps a closed folder in case you may want to reopen it again. To close a folder, right click on the project you want to close, and select *close project.*

The navigator can also filter specific file types. To do this click on the arrow pointing downwards button in the navigator and select filters. You can select which file types you want to filter out by checking off its' box. After selecting *Ok*, you will notice the navigator will show files that you have not checked.

2.1.6 Using the task list

The task list (on the bottom section of the CD++Builder perspective) is used to help organize any notes, reminders or directions you have on your project. You can use it to leave notes on any files' line of code or even a reminder for a task you want to do later. It is weaved together with any of the editors supported in Eclipse to place task markers on any line of code. This can be done by opening any file and right clicking on the far left bar of the editor panel and selecting *add task*. Here you can enter the description of the task and its priority level. When you click on *Ok* , you can see on the task list that your task has been added. Double clicking on this task will automatically open up the file and bring you to the line of code.

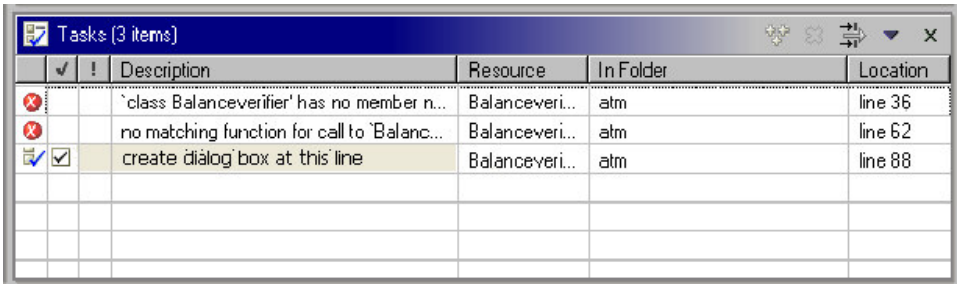


Figure 22. Task list view

To add a generic task that is may not be code based, right click on the task list and select *new task.* This will prompt you with the similar panel as above asking you for the description and priority.

Another useful feature of the task list is to show errors that have occurred from the compilation of the C++ files using the *Build* button. The errors are shown in the task lists and by double clicking on one of them will automatically open up the file and take you to the line of code where the error is.

2.1.7 Using the Outliner

The outline is used for C++ files. When you have opened a C++ file and it is the active part in the editor. The outliner outlines all the variables, functions and dependencies of that particular class. This can be used for quick referencing by double clicking on a function or variable it will bring you immediately to its location in the source code.

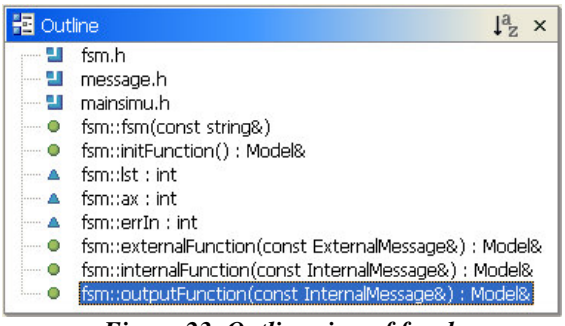


Figure 23. Outline view of fsm.h

2.2 Defining Atomic models in CD++

We will now show how to make use of the features mentioned previously by building an atomic model from scratch. These files can be created in CD++Builder as explained in section <> or a standard text editor. The

example model to be created is a queue which is a device of temporary storage that uses a FIFO (First in First out) mechanism. The source code of this model comes included with the CD++ toolkit. When you download the original zip files, you will find the queue.cpp and queue.h files containing the models discussed in this section. In CD++Builder, you will find it in the `...\eclipse\plugins\CD++Builder_1.1.0\internal` folder.

2.2.1 Adding new models

These are the steps to add a new atomic model:

a) Write a class derived from *Atomic* **overloading** the following methods:

- ***initFunction***: Before calling this method, the sigma value is infinite and the state is passive.
- ***externalFunction***: Called when an external event arrives in one of the model's output ports.
- ***internalFunction***: Before calling this method, the sigma value is zero because the interval to the internal transition has expired.
- ***outputFunction***: Called when an internal event arrives, before calling the internal transition function.
- ***className***: the class name.

b) Modify **register.cpp**, adding to the method Simulator::registerNewAtomics() the new atomic model, in the Queue example, we can see:

```
SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Queue>(), "Queue" )
```

2.2.2 Interacting with the Simulator

In each of the former methods you have to use some primitives to interact with the simulator in order to accomplish the common atomic model operations.

These primitives are:

- **holdIn(state, time):** The model changes its state into *state* by time *time* and when this interval expires a change of state has to occur. The state could be: *active passive*.
- **passivate():** The model change its state to *passive* and it will only be activated when an external event arrives.
- **sendOutput(time, port, value):** It sends an output message.
- **nextChange():** It informs the time remaining before the next change of state (sigma).
- **lastChange():** It informs the time of the last change of state.
- **state():** It informs the model's state.
- **getParameter(modelName, parameterName):** It gets the parameter *parameterName* value.

Since *Model* is an abstract base class, it defines the interface for message exchange, *Atomic* and *Coupled* classes are the only ones which can receive and send messages. The derived classes are responsible for overloading the initialization, internal transition, external transition and output methods.

The *Atomic* derived classes should not send any kind of message, except for the output values informed through the *sendOuput* method. The *Atomic* class is responsible for sending the Y and D messages to their parents using the *sigma* and *state* values.

2.2.3 An example: a model of a queue

Our model of a queue will hold any type of user defined value. The queue will have three input ports and one output port. Values to be stored will be received through the input port *In* and will later be sent through the port *Out.* The input ports *start-stop* and *next* will serve to regulate the flow of values through the output port. Figure 24 shows the structure of our model of a queue.

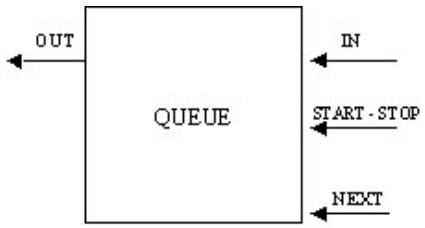


Figure 24. Structure of a Queue

Initially, the queue is empty. When the first value is received through the input port *In*, it will be stored in the queue and forwarded through the output port *Out* after a time as defined by the user parameter *preparationTime*. If a value is received and the queue is not empty, then it will be stored, but it will not be forwarded immediately. Instead, it will be sent through the output port *Out* only after a message is received through the port *next*.

A message received through the input port *start-stop* will temporarily disable the queue. If the queue is disabled, it will only respond to new events received through the input port *In*. Any value received will be stored, but no output will be ever sent until the queue is enabled again by sending an event to the *start-stop* port.

After this brief description, we are ready to begin writing our model. Create a new project. Once the project is created, create a new .cpp file. For more information refer the previous topic. First, we need to define a class to store the state of the queue. The queue will have two state variables: a list of elements and a *boolean* to store the enabled/disabled status. Figure 3.10, lists the Queue state class declaration and definition.

The first step when implementing a new atomic model is to define a class derived from *Atomic* overloading the methods for transition handling. These methods are not public since only the class *Atomic* can invoke them. Atomic is an abstract class that declares a model's API and defines some service functions the user can use to write her model. The class atomic provides a set of services and requires a set of functions to be redefined. The services are functions that allow the model to tell the simulator the current state and duration.

The following example shows the variable's definitions referring to the queue's ports and the time it takes to prepare the value before sending it through the out port. It also shows the definition of the list of values to hold the input data (*ElementList*) and the time remaining when the model is interrupted by a flow control signal (*timeLeft)*.

```
class Queue : public Atomic {
public:
   Queue(); // Default constructor
   virtual string className() const ;

protected:
   Model &initFunction();
   Model &externalFunction( const ExternalMessage & );
```

```
   Model &internalFunction( const InternalMessage & );
   Model &outputFunction( const InternalMessage & );

private:
   const Port &in;
   const Port &stop;
   const Port &done;
   Port &out;
   Time preparationTime;

   typedef list<Value> ElementList ;
   ElementList elements ;

   Time timeLeft;

}; // class Queue
```

Figure 25. Queue.h: model definition.

The constructor creates the input and output ports and sets the variable *preparationTime*. The parameter *preparation* must be specified in the configuration file.

```
Queue::Queue()
   : preparationTime( 0, 0, 10, 0 )
   , in( this->addInputPort( "in" ) )
   , stop( this->addInputPort( "stop" ) )
   , done( this->addInputPort( "done" ) )
   , out( this->addOutputPort( "out" ) )     {
   this->description( "Queue" ) ;

   string time( Simulator::Instance().getParameter(
                          this->description(), "preparation" ) ) ;

   if( time != "" )
      preparationTime = time ;
}
```

Figure 26. Queue.cpp: model constructor.

The initialization function erases the queued data list. The following state change will take place when an external event arrives, which is why *sigma* remains constant. If you wish to modify the next state change use the *holdIn* method. The *getParameter* method queries the coupled model file (*.ma file, described in the following section) and identifies the "preparation" parameter. The value of the parameter (a string), is converted into the initial preparation time.

```
Model &Queue::initFunction(){
   elements.remove( elements.begin(), elements.end() ) ;
   return *this;
}
```

Figure 27. Queue.cpp: model initialization function.

The Queue class has three input ports through which it can receive external events. An event that arrives in the ***in*** port represents a new input value, which has to be queued. If it is the only one in the queue it has to be prepared to be sent.An event that arrives in the port ***done*** indicates that the last element sent has been received and therefore it has to be erased from the queue. If there were more elements to be sent the first value in the queue should be prepared to be released. An event that arrives in the port ***stop*** indicates that the flow should be stopped or restarted. If the queue was in *active* state and the message value is not zero the queue will execute a working pause, here the time remaining to process the next state change is calculated (end of preparation time) and then the queue changes its state to *passive* calling the *passsivate* method. If the queue was in *passive* state

and the message value is zero then the queue restarts the work setting the next state change to the remaining processing time.

```
Model &Queue::externalFunction( const ExternalMessage &msg ){
   if( msg.port() == in )  {
      elements.push_back( msg.value() ) ;
      if( elements.size() == 1 )
         this->holdIn( active, preparationTime );
   }

   if( msg.port() == done ) {
      elements.pop_front() ;
      if( !elements.empty() )
         this->holdIn( active, preparationTime );
   }

   if( msg.port() == stop )
      if( this->state() == active && msg.value() )
      {
         timeLeft = msg.time() - this->lastChange();
         this->passivate();
      }
      else
         if( this->state() == passive && !msg.value() )
            this->holdIn( active, timeLeft );

   return *this;
}
```

Figure 28. Queue.cpp: external transition function.

When the preparation time interval expires this method is invoked and the first value in the queue has to be sent through the output port.

```
Model &Queue::outputFunction( const InternalMessage &msg ){
   this->sendOutput( msg.time(), out, elements.front() ) ;
   return *this ;
}
```

Figure 29. Queue.cpp: output function.

When the preparation time interval expires and after calling the output function this method is invoked. Here there is nothing to do, except wait for the acknowledge in the *done* port.

```
Model &Queue::internalFunction( const InternalMessage & ){
   this->passivate();
   return *this ;
}
```

Figure 30. Queue.cpp: internal transition function.

A new atomic model is created as a new class that inherits from the class *Atomic*. To tell CD++ that a new atomic definition has been added, the model must be registered in the method *MainSimulator.registerNewAtomics*(). This method is located in the register.cpp, which is shown in the next figure:

```
void MainSimulator::registerNewAtomics() {
  SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Queue>() ,
"Queue" ) ;
  SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Generator>() ,
"Generator" ) ;
  SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<CPU>() , "CPU"
```

```
) ;
    SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Transducer>()
, "Transducer" ) ;
    SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Trafico>() ,
"Trafico" ) ;
}
```

Figure 31. Content of register.cpp

2.2.4 Creating new atomic models for parallel simulation

A new atomic model is created as a new class that inherits from *Atomic*. To tell CD++ that a new atomic definition has been added, the model must be registered in the ParallelMainSimulator.registerNewAtomics() function. In addition, for an atomic model to support the TimeWarp protocol, a model's state has to be defined as a separate class that is derived from *AtomicState*. The current state is available through the function *getCurrentState()* which returns a pointer to the model state. States are managed by the Warped kernel, and are only valid through a simulation cycle. There is no guarantee a pointer returned during a simulation cycle will still be valid during the next one. In addition, the states are not created until the initFunction is called, so no state initialization code should be placed in the class constructor.

- **virtual Model &initFunction():**

This method is invoked by the simulator at the beginning the simulation and after the model state has been initialized. All initialization should take place when this method is call. An active model should usually set the time for the next transition using the **holdIn** function. The holdIn function will be further explained later in this section.

- **virtual Model &externalFunction (const MessageBag &)**

virtual Model &externalFunction(const ExternalMessage &): These methods are invoked when one or more external events arrive from a port of the model. It corresponds to the δ_{ext} function of the DEVS formalism. The simulator calls the first function, the one that receives a message bag. By default, this function will iterate through all the messages in the bag and call the second one. This is provided for backward compatibility. If the modeler would like to have more control on the model's behavior when multiple simultaneous events are received, it is recommended the first function is overridden. If the model's behavior is simple enough for simultaneous events to be handled sequentially, then it will be enough to redefine the second function.

The interface for the MessageBag class is shown below.

```
class MessageBag {

public:

       MessageBag();     //Default Constructor
       ~MessageBag();

       MessageBag &add( const BasicPortMessage* );

       bool portHasMsgs( const string& portName ) const;

       const MessageList& msgsOnPort( const string& portName )
const;

       int size() const
```

```
        MessageBag& eraseAll();

        const VTime& time() const;

};
```

Figure 32. MessageBag class

- **virtual Model &internalFunction(const InternalMessage &):** This method corresponds to the δ_{int} function of the DEVS formalism.
- **virtual Model &outputFunction(const CollectMessage &):** This function is called before δ_{int}. It should send all the output event. Each output event is sent using the function sendOutput defined below.
- **virtual Model &confluentFunction (const InternalMessage &, const MessageBag &):** It corresponds to the δ_{conf} function of the DEVS formalism. By default, it is set to:

```
Model &Atomic::confluentFunction ( const InternalMessage &intMsg,
const MessageBag &extMsgs )
{
   //Default behavior for confluent function:
   //Proceed with the internal transition and the with the external
   internalFunction( intMsg );

   //Set the elapsed time to 0
   lastChange( intMsg.time() );

   //Call the external function
   externalFunction( extMsgs );

   return *this;

}
```

Figure 33. confluentFunction method

- **virtual string className():** Returns the name of the atomic class.

  The following methods can invoke certain predefined primitives allow to interactuar with the abstract simulator:

- **holdIn(state, time):** indicates to the simulator that the model should stay in the same state during a time, and after that it will generate an internal transition.
- **passivate():**indicates to the simulator that the model enters in passive mode and that it will only be reactivated when an external event arrives.
- **sendOutput(time, port, value):** sends an output message through the port.
- **nextChange()**: this method allows to obtain the remaining time for its next state change (sigma).
- **lastChange():** this method allows to obtain the time in that the last state change took place.
- **state():** this method obtain the actual phase of the model.
- **getParameter(modelName, parameterName):** this method allows to access to the parameters that configure the class. In figure 3.1, shows the model name between the clauses and the different parameter names which range from "value*1*" to "value*n*".
- **virtual Port &addInputPort(const string &)**

virtual Port &addOutputPort(const string &): These methods add the input and output port of the model respectively.

- **ModelState* getCurrentState():** This method get the current state of the model.
- **virtual int valueSize() const:** Returns the size of the class. It should be set to:
  `return sizeof( className);`
- **virtual string asString():** Returns a string that is used in the log file to log the value sent or received.
- **virtual BasicMsgValue * clone():** Returns a pointer to a new copy of the message value. The function that receives the pointer will own it and afterwards delete it.
- **BasicMsgValue(const BasicMsgValue&):** Copy constructor.

The state of a model is made of all those variables that can change during a simulation cycle. The basic state variables required by an atomic model are defined in the *AtomicState* class. A user can create a new class to define the state variables required by his model.

The AtomicState class declaration is shown below.

```
class AtomicState : public ModelState {
public:

        enum State
        {
                active,
                passive
        } ;


        State st;

        AtomicState(){};
        virtual ~AtomicState(){};

        AtomicState& operator=(AtomicState& thisState); //Assignment
        void copyState(BasicState *);
        int  getSize() const;

};
```

Figure 34. The AtomicState class.

To access the current state this function ;

```
ModelState* getCurrentState()
```

should be used. The pointer that is returned can be casted to the proper type.

An assignment operator and a copy constructor need to be provided for Warped to work properly. In addition, the method getSize should be overridden to return the size of the class.The set of services provided by the class atomic as well as the methods required to be redefined can be seen in section 3.1.1.

The user can define a new class for the output values. To define a new structure for output values, a new class that derives from BasicMsgValue has to be created. A class for sending and receiving real values is already provided.

There is only restriction that applies: no pointers can be defined as part of the class. This is because message values are sent across a network when parallel simulation is used and pointers will be just copied as pointers. The data they are pointing to will not be copied.

```
class BasicMsgValue  {
public:
    BasicMsgValue();
    virtual ~BasicMsgValue();
    virtual int valueSize() const;
    virtual string asString() const;
    virtual BasicMsgValue* clone() const;

    BasicMsgValue(const BasicMsgValue& );

};

class RealMsgValue : public BasicMsgValue {
public:
    RealMsgValue();
    RealMsgValue( const Value& val);

    Value v;
    int valueSize() const;
    string asString() const ;
    BasicMsgValue* clone() const;
    RealMsgValue(const RealMsgValue& );
};
```

Figure 35. The BasicMsgValue and RealMsgValue classes

The user needs to define the following functions:

- **virtual int valueSize() const;**
  Returns the size of the class. It should be set to:
  `return sizeof( className);`

- **virtual string asString**();
  Returns a string that is used in the log file to log the value sent or received.

- **virtual BasicMsgValue * clone**();
  Returns a pointer to a new copy of the message value. The function that receives the pointer will own it and afterwards delete it.

- **BasicMsgValue(const BasicMsgValue&)**
  A copy constructor is required.

Once the state class has been defined, we are ready to implement the model itself. The Queue class declaration is shown in Figure 3.11.

```
class QueueState : public AtomicState {

public:

     typedef list<BasicMsgValue *> ElementList ;
     ElementList elements ;
     bool enabled;

     QueueState(){};
     virtual ~QueueState(){};

     QueueState& operator=(QueueState& thisState){
         (AtomicState &)*this  = (AtomicState &) thisState;

         ElementList::const_iterator cursor;
```

```
        for(cursor = thisState.elements.begin();
               cursor != thisState.elements.end(); cursor++)

               elements.push_back( cursor->clone() );

        return *this;
    }

    void copyState(QueueState *){ *this = *((QueueState *) rhs);}

    int  getSize() const    { return sizeof(QueueState);}
};
```

Figure 36. QueueState class

The *Queue* model overloads the initialization methods, internal function, external transition and output function. In addition, its shortcut functions to access the elements of the current state.

```
class Queue : public Atomic{
public:
 Queue( const string &name = "Queue" );
 virtual string className() const {  return "Queue" ;}
protected:
 Model &initFunction();
 Model &externalFunction( const MsgBag & );
 Model &internalFunction( const InternalMessage & );
 Model &outputFunction( const CollectMessage & );

 ModelState* allocateState()
 {      return new QueueState;}

 private:
        Port &in, &done, &out;

        VTime preparationTime;

QueueState::ElementList& elements(){ return
((QueueState*)getCurrentState())->elements; }

 bool enabled() const{   return ((QueueState*)getCurrentState())->enabled;
}

 void enabled (bool val){  ((QueueState*)getCurrentState())->enabled = val;
}

};      // class Queue
```

Figure 37. The Queue class declaration

The initFunction has to set the initial state for the queue, as shown in Figure 3.12. The elements of the list will be erased and the enabled will be set to true.

```
Model &Queue::initFunction()
{
  enabled( true );
   return *this;
}
```

Figure 38. initFunction for the Queue model

The externaFunction will be activated every time one or more events are received. For the queue model, this

function will have to insert into the queue all values received through port *In*, schedule an output if a value is received through the port *next* and enabled or disable the queue if an event is received through port *start-stop*, as detailed in Figure 3.13. It is important to notice that it is the modeler's responsibility to set which message will have the highest priority when more than one is received. For our queue model, it can be seen from Figure 3.13 that the *start-stop* messages will have higher precedence than the *done* and *in* messages.

```
Model &Queue::externalFunction( const MsgBag & bag )
{

        if ( portHasMsgs( "start-stop") )
        {
              enabled ( !enabled() );
              if ( !enabled() )
                    passivate();
        }

        if ( enabled() && portHasMsgs( "done" ))
        {
              elements().pop_front();
              holdIn( AtomicState::active, preparationTime );
        }

        if ( portHasMsgs( "in" )
        {

              MessageList::const_iterator cursor;
              cursor = bag.msgOnPort( "in" ).begin();

              for (; cursor != bag.msgsOnPort( "in").end() ; cursor++)
                    elements().push_back( cursor.value());

              //If the queue was empty, schedule the next transition
              if ( enabled() && elements.size()==msgsOnPort("in").size() )
                    holdIn( AtomicState::active, preparationTime );


        }

}
```

Figure 39. External transition function for the queue model

The output function is called before an internal transition. In our queue model, the output function should send the first value in the list through the output port. The internal transition function will passivate the model which will wait for an external event to take place.

```
Model &Queue::outputFunction( const CollectMessage &msg )
{
   sendOutput( msg.time(), out, elements.front() );
   return *this;
}

Model &Queue::internalFunction( const InternalMessage & )
{
   passivate();
   return *this;
}
```

Figure 40. Methods for the Output Function and the Internal Transition of the Queue

The sendOutput function will delete the pointer it receives, so all memory previously allocated to store the queue values will be reclaimed. If we wanted to use the queue for a network model, the queue would store IP packets. Then an IP packet class derived from BasicMsgValue should be defined.

3 Coupled models

After defining the atomic models for a given application, they can be combined into a multicomponent model. Coupled models are defined using a specification language specially defined with this purpose. The language was built following the formal definitions for DEVS coupled models. Therefore, each of the components defined formally for DEVS coupled models can be included. Optionally, configuration values for the atomic models can be included.

The **[top]** model always defines the coupled model at the top level. As showed in the formal specifications presented earlier, four properties must be configured: components, output ports, input ports and links between models. The following syntax is used:

- **Components**:
  components : model_name1[@atomicclass1] model_name [@atomicclass2] ...
  Lists the component models that make the coupled model. If this clause is not specified, an error will occur. A coupled model might have atomic models or other coupled model as 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 an atomic class. For coupled models, only the model name must be given. This model name must be defined as another group in the same file.
- **Out**:
  out : portname1 portname2 ...
  Enumerates the model's output ports. This clause is optional because a model may not have output ports.
- **In:**
  in : portname1 portname2 ...
  Enumerates the input ports. This clause is also optional because a couple model is not required to have input ports.
- **Link :**
  link : *source_port*[@*model*] *destination_port*[@*model*]
  Defines the links between the components and between the components and the coupled model itself. If name of the model is omitted it is assumed that the port belongs to the coupled model being defined.

A model definition is shown below.

```
[top]
components : transducer@Transducer generator@Generator Consumer
Out : out
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out

[Consumer]
components : queue@Queue processor@Processor
in : in
out : out
Link : in in@queue
Link : out@queue in@processor
Link : out@processor done@queue
Link : out@processor out
```

Figure 41. Example for the definition of a DEVS coupled model

CD++Builder allows the user to easily import and export maml files. .maml files are a representation of coupling files in XML format. Exporting a ".ma" file into maml format is done automatically every time you save your ".ma" file. The saving feature writes a maml file with the same name as the ma file in maml format. This is automatically added to your project.

Importing maml files can be simplistic as well. When you have a maml file in your project, right clicking it and selecting convert to ma, will create a coupling file implementation of the maml file. This coupling file is written and created in your project and should appear in the projects' contents

As it was mentioned above, atomic models must be coded. In addition, an atomic model might have user defined parameters that must be specified within the .ma file. If this is the case, the parameters are specified in a group with the model's name (the model's name as defined in the components clause, not the atomic class name).

```
[model_name]
var_name1 : value1
.
.
.
var_namen : valuen
```

Figure 42. User defined values for atomic models

The parameter names are defined by the model's author and must be documented. Each instance of an atomic model can be configured independently of other instances of the same kind.

The next example shows two instances of the atomic class *Processor* with different values for the user defined parameters.

```
[top]
components : Queue@queue Processor1@processor Processor2@processor
.
.
.

[processor]
distribution : exponential
mean : 10

[processor2]
distribution : poisson
mean : 50

[queue]
preparation : 0:0:0:0
```

Figure 43. Example of setting parameters to DEVS atomic models

4 Cell-DEVS models

The tool includes a specification language allowing the description of Cell-DEVS models. These definitions are based on the formal specifications defined earlier, and can be completed by considering a few parameters: size, influencees, neighborhood and borders. These are used to generate the complete cell space. The behavior of the local computing function is defined using a set of rules with the form: VALUE DELAY { CONDITION }. These indicate that when the *CONDITION* is satisfied, the state of the cell changes to the designated *VALUE*, and it is *DELAY*ed for the specified time. If the condition is *false*, the next rule in the list is evaluated until a rule is satisfied or there are no more rules. In the latter case, an error is raised, indicating that the model

specification is incomplete. The existence of two or more rules with same condition but with different state value or delay is also detected, avoiding the creation of ambiguous models. In these situations, the simulation is aborted.

In CD++, Cell DEVS models are a special case of coupled models. Then, when defining a cellular model, all the coupled model parameters are available. In addition there exist some parameters that are of cellular models. These parameters define the dimensions of the cell space, the type delay, the default initial values and the local transition rules.

These parameters are:

- **type :** [**CELL | FLAT**]
  Defines the abstract simulator to be used. If **cell** is specified, there will be one DEVS processor for each cell. Instead, if **flat** is specified, one flat coordinator will be used. CD++ currently supports the **cell** option only.

- **width :** integer

  Defines the width of the cellular space. As it is the case with height, the **width** parameter is provided for backward compatibility and implies that a 2-dimensional cellular space will be used. For an n-dimensional cell space the **dim** parameter should be used. **width** and **height** can not be used together with **dim**. If such a situation exists, an error will be reported.

- **height :** integer

  Defines the height of the cellular space model. The same restrictions that were given for **width** apply. For 1 dimension models, **height** should be set to 1.

- **dim :** $(x_0, x_1, ..., x_n)$

  Defines the dimensions of the cellular space. All the x_i values must be integers.
  Dim can not be used together with any of the **width** and **height** parameters.
  The vector that defines the dimension of the cellular model must have two or more elements.
  For a one-dimensional cellular model, the following form should be used; $(x_0, 1)$.
  When referencing a cell, all references must satisfy:
  $(y_0, y_1, ..., y_n)$ $0 \leq y_i < x_i$ $\forall$ i = 0, .., n with y_i an integer value

- **In :** Defines the input ports for a cellular model.

- **Out :** Defines the output ports the cellular model.

- **Link :** Defines the components coupling. For a coupled cell model, the components are cells. To define the couplings, cell references must be used for the model name. A cell reference is of the form:

  CoupleCellName(x_1,x_2,...,x_n)

  Valid link definitions are of the form:

  Link : outputPort inputPort@cellName (x_1,x_2,...,x_n)
  Link : outputPort@cellName (x_1,x_2,...,x_n) inputPort
  Link : outputPort@cellName (x_1,x_2,...,x_n)inputPort@cellName(x_1,x_2,...,x_n)

- **Border :** [**WRAPPED | NOWRAPPED**]
  Defines the type of border for the cellular space. By default, NOWRAPPED is used. If a nonwrapped border is used, a reference to a cell outside the cellular space will return the undefined value (**?**).

- **Delay :** [**TRANSPORT | INERTIAL**]
  Specifies the delay type used for all cells of the model. By default the value TRANSPORT is assumed.

- **DefaultDelayTime :** integer

  Defines the default delay (in milliseconds) for inputs received from external DEVS models. If a **portInTransition** is specified, then this parameter will be ignored for that cell.

- **Neighbors :** cellName ($x_{1,1}$, $x_{2,1}$,...,$x_{n,1}$)... cellName ($x_{1,m}$, $x_{2,m}$,...,$x_{n.m}$)

  Defines the neighborhood for all the cells of the model. Each cell ($x_{1,i}$, $x_{2,i}$,...,$x_{n.i}$) represents a displacement from the centre cell (0,0,...., 0)

  A neighborhood can be defined with any valid list of cells and is not restricted to adjacent cells.

  It is possible to use more than one **neighbors** sentence to define the neighborhood.

- **Initialvalue :** [*Real* | **?**]

  Defines the default initial value for each cell. The symbol **?** represents the undefined value. There are several ways of defining the initial values for each cell. The parameter **initialvalue** has the least precedence. If another parameter defines a new value for the cell, then that value will be used.

- **InitialRowValue :** row_i $value_1$...$value_{width}$

  Defines the initial value for all the cells in row i.

  Precondition:

  $0 \leq row_i <$ *Height* (where *Height* is the second element of the dimension defined with **Dim**, or the value defined with **Height**).
  Can only be used for bidimensional models. For n-dimensional models the **initialCellsValue** or **initialMapValue** parameters are preferred.

  This clause is used for backward compatibility. All values are single digit values in the set {**?**, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. The first digit will define the value for the first cell in the row, the second for second cell and so on. No spaces are allowed between digits.

- **InitialRow :** row_i $value_1$ ... $value_{width}$

  Same as **initialrowvalue**, but values can now be any member of the set $\Re$ U {**?**}. Each value in the list must be separated by a blank space from the next one.

- **InitialCellsValue** : *fileName*

  Defines the filename for the file that contains a list of initial value for cells in the model. Section 6.1 defines the format for these files. **initialcellsvalue** can be used with any size of cellular models and will have more precedence that **initialrow** and **initialrowvalue.**

- **InitialMapValue** : *fileName*

  Defines the filename for the file that contains a map of values that will be used as the initial state for a cellular model. Section 6.2 defines the format for these files.

- **LocalTransition :** *transitionFunctionName*

  Defines the name of the group that contains the rules for the default local computing function.

- **PortInTransition :** *portName@ cellName* (x_1, x_2,...,x_n)
  TransitionFunctionName

  It allows to define an alternative local transition for external events. By default, if this parameter is not used, when an external event is received by a cell its value will be the future value of the cell with a delay as set by the **defaultDelayTime** clause.

  Section 12.3 illustrates the use of the **portInTransition** clause.

- **Zone :** *transitionFunctionName* { $range_1$[..$range_n$] }

  A zone defines a region of the cellular space that will use a different local computing function. A zone is defined giving as a set of single cells or cell ranges. A single cell is defined as (x_1,x_2,...,x_n), and a range as (x_1,x_2,...,x_n)..(y_1,y_2,y_n). All cells and cell ranges must be separated by a blank space. As an example,

  zone : pothole { (10,10).. (13, 13) (1,3) }

  tells CD++ that the local transition rule pothole will be used for the cells in the range (10,10)..(13,13) and the single cell (1,3). The zone clause will override the transition defined by the **localtransition** clause.

The following figure illustrated the .ma file of the life game:

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
defauultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1-1) life(1,0) life(1,1)
initialvalue : 0
initialvalue : 5     00000001110000000000
initialvalue : 7     00000100100100000000
initialvalue : 8     00000101110100000000
initialvalue : 9     00000100100100000000
initialvalue : 11    00000001110000000000
localtransition : life-rule

[life-rule]
rule : 1 100 { (0,0) = 1 and trueCount = 5 }
rule : 1 100 { (0,0) = 0 and trueCount = 3 }
rule : 0 100 { t }
```

Figure 44. Example for the definition of a Cell-DEVS life model

5 Supporting files

5.1 Defining initial cell values using a .val file

Within the definition of a cellular model, the *InitialCellValue* parameter defines a file name with the initial values for the cells. This is a plain text file. Each line of the file defines a value for a different cell. The format of this file is shown in Figure 6.1.

```
(x0,x1,...,xn) = value_1
...       ...      ...
(y0,y1,...,yn) = value_m
```

Figure 45. Format of the file used to define the initial values of a cellular model

The extension **.VAL** is normally used for this kind of files. The file is processed in sequential order, so if there are two values defined for the same cell, the latest one will be used.

The dimension of the tuple should match the dimensions of the cellular space.

For the definition of the initial values of a cellular model, a single file should be used, which can not contain initial values for other cellular models.

It is not necessary to define an initial value for each cell. If no value is defined in this file, then the value defined by the parameter *InitialValue* will be used. Figure 6.2 shows a short fragment of a .val file for a cellular space of 4 dimensions.

```
(0,0,0,0) = ?
(1,0,0,0) = 25
(0,0,1,0) = -21
(0,1,2,2) = 28
(1, 4, 1,2) = 17
(1, 3, 2,1) = 15.44
(0,2,1,1) = -11.5
(1,1,1,1) = 12.33
(1,4,1,0) = 33
(1,4,0,1) = 0.14
```

Figure 46. Example of a file for the definition of the initial values for a Cellular Model

5.2 Defining initial cell values using a .map file

If the *InitialMapValue* parameter is used, then the initial values for a cellular model are specified in a .map file. This file contains a map of cell values, as shown in Figure 6.3:

```
value_1
...  ...
value_m
```

Figure 47. .map file format

Each value of the .map file will be assigned to a cell starting with the origin cell (0,0...,0). For a three-dimensional cellular model of size (2, 3, 2), the values will be assigned in the following order:

```
(0,0,0) (0,0,1) (0,1,0), (0,1,1) (0,2,0) (0,2,1) ... (1,2,0) (1,2,1)
```

If there are not enough values in the file for all the cells in the model, the simulation will be aborted. If instead there are more values than cells, the remaining values will be ignored.

The *toMap* tool creates a .map file from a .val file.

5.3 External events file

External events are defined in a plain text file with one event per line. Each line will be of the format:

HH:MM:SS:MS PORT VALUE

Where:

HH:MM:SS:MS	is the time when the event will occur.
Port	is the name of the port from which the event will arrive.

Value is the numerical value for the event. Can be a real number or the undefined value (**?**).

Example:

```
00:00:10:00 in 1
00:00:15:00 done 1.5
00:00:30:00 in .271
00:00:31:00 in -4.5
00:00:33:10 inPort ?
```

Figure 48. File with external events

5.4 Partition file

A partition file is required for parallel simulation. For each atomic model, the partition file defines the machine that will host its associated simulator. For coupled models, CD++ will decide where the coordinators will be running.

A partition file, usually referred as a .par file, has lines with the following format:

```
MachineNumber : modelName1 modelName2  cell(x,y) cell(x,y)..(x2, y2)
```

A line starts with a machine number (machine numbers start at 0) followed by a space, a colon and a list of names separated by spaces. Different lines may start with the same machine number.The list of names following a machine number is the list of atomic instances that will be hosted by that machine. For cellular models, a single cell may be specified or a range of cells may be given. A cell range is described with name of the coupled cell model followed by the first cell in the range, two dots, and the last cell in the range.

As an example, consider the following partial definition of a model:

```
[top]
components : superficie generadorCalor@Generator generadorFrio@Generator
link : out@generadorCalor inputCalor@superficie
link : out@generadorFrio  inputFrio@superficie

[superficie]
type : cell
width : 100
height : 100
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : superficie(-1,-1) superficie(-1,0) superficie(-1,1)
neighbors : superficie(0,-1)  superficie(0,0)  superficie(0,1)
neighbors : superficie(1,-1)  superficie(1,0)  superficie(1,1)
initialvalue : 24
in : inputCalor inputFrio
```

Figure 49. Partial definition of the heat diffusion model

If we wanted to run this model in a cluster of nine machines, then the following is a valid partition:

```
0 : generadorCalor generadorFrio
0 : superficie(0,0)..(32,32)
1 : superficie(0,33)..(32,65)
2 : superficie(0,66)..(32,99)
3 : superficie(33,0)..(65,32)
```

```
4 : superficie(33,33)..(65,65)
5 : superficie(33,66)..(65,99)
6 : superficie(66,0)..(99,32)
7 : superficie(66,33)..(99,65)
8 : superficie(66,66)..(99,99)
```

Figure 50. Valid partition for the heat diffusion model over 9 machines

A valid partition must specify one and only one location for each atomic and each cell. If more than one machine or no machine is specified for a model, then an error will be raised and the simulation will be aborted.

6 Output Files

6.1 Output events

If the command line option **–o** is given, all the output events generated by the simulator are written to the specified file. There will be one event per line, and lines will have the following format:

```
HH:MM:SS:MS PORT VALUE
```

Following is a small example of an output file.

```
00:00:01:00 out 0.000
00:00:02:00 out 1.000
00:00:03:50 outPort ?
00:00:07:31 outPort 5.143
```

Figure 51. Example of an Output file

6.2 Format of the Log File

A log file keeps a record of all the messages sent between DEVS processors. A log is created when the **–l** command line argument is used. If no log modifiers are specified, all received messages are logged. Otherwise, only those messages set by the log modifiers will be logged.

When a filename for the log is given, there will be one file per DEVS processor and one file with the list of all the names of the files that have been created. This latter file will be named with the name given after the **–l** parameter. All other files will be named with the name after the **–l** parameter followed by the DEVS processor id.

Each line of the file shows the number of the LP that received the message, the message type, the time of the event, the sender and the receiver. In addition, messages of type ***X*** or ***Y*** will include the port through which the message was received and the value received. For messages of type ***D*****,** the remaining type for the next transition will be shown. A '…' for this field will indicate infinity.

The numbers between brackets show the ID of the DEVS processor and are provided for debugging purposes only.

As an example, the log files for the following model will be shown.

```
[top]
components : superficie generadorCalor@Generator generadorFrio@Generator
link : out@generadorCalor inputCalor@superficie
link : out@generadorFrio  inputFrio@superficie

[superficie]
```

```
type : cell
width : 5
height : 5
...
```

Figure 52. Partial definition of the heat diffusion model

When running this model with the –**lcalor.log** parameter, the following are the contents of calor.log.

```
[logfiles]
ParallelRoot : calor.log00
top : calor.log29
superficie : calor.log01
superficie(0,0) : calor.log02
superficie(0,1) : calor.log03
superficie(0,2) : calor.log04
superficie(0,3) : calor.log05
superficie(0,4) : calor.log06
superficie(1,0) : calor.log07
superficie(1,1) : calor.log08
superficie(1,2) : calor.log09
superficie(1,3) : calor.log10
superficie(1,4) : calor.log11
superficie(2,0) : calor.log12
superficie(2,1) : calor.log13
superficie(2,2) : calor.log14
superficie(2,3) : calor.log15
superficie(2,4) : calor.log16
superficie(3,0) : calor.log17
superficie(3,1) : calor.log18
superficie(3,2) : calor.log19
superficie(3,3) : calor.log20
superficie(3,4) : calor.log21
superficie(4,0) : calor.log22
superficie(4,1) : calor.log23
superficie(4,2) : calor.log24
superficie(4,3) : calor.log25
superficie(4,4) : calor.log26
generadorcalor : calor.log27
generadorfrio : calor.log28
```

Figure 53. Calor.log

This is a list of the models and their corresponding files. If more than one file is created (as is the case of coupled models with more than one coordinator), all of them are listed. The log messages received by the coordinator superficie will be logged into the file calor.log01, which is shown next.

```
0 I / 00:00:00:000 / top(29) para superficie(01)
0 D / 00:00:00:000 / superficie(0,0)(02) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,1)(03) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,2)(04) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,3)(05) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,4)(06) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,0)(07) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,1)(08) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,2)(09) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,3)(10) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,4)(11) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,0)(12) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,1)(13) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,2)(14) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,3)(15) / 00:00:00:000 para superficie(01)
```

```
0 D / 00:00:00:000 / superficie(2,4)(16) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,0)(17) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,1)(18) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,2)(19) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,3)(20) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,4)(21) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(4,0)(22) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(4,1)(23) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(4,2)(24) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(4,3)(25) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(4,4)(26) / 00:00:00:000 para superficie(01)
0 @ / 00:00:00:000 / top(29) para superficie(01)
0 Y / 00:00:00:000 / superficie(0,0)(02) / out /  24.00 para superficie(01)
0 D / 00:00:00:000 / superficie(0,0)(02) / 00:00:00:000 para superficie(01)
0 Y / 00:00:00:000 / superficie(0,1)(03) / out /  24.00 para superficie(01)
0 D / 00:00:00:000 / superficie(0,1)(03) / 00:00:00:000 para superficie(01)
0 Y / 00:00:00:000 / superficie(0,2)(04) / out /  24.00 para superficie(01)
0 D / 00:00:00:000 / superficie(0,2)(04) / 00:00:00:000 para superficie(01)
0 Y / 00:00:00:000 / superficie(0,3)(05) / out /  24.00 para superficie(01)
0 D / 00:00:00:000 / superficie(0,3)(05) / 00:00:00:000 para superficie(01)
0 Y / 00:00:00:000 / superficie(0,4)(06) / out /  24.00 para superficie(01)
0 D / 00:00:00:000 / superficie(0,4)(06) / 00:00:00:000 para superficie(01)
...
...
0 X / 00:00:00:000 / top(29) / inputcalor /   1.00 para superficie(01)
0 X / 00:00:00:000 / top(29) / inputfrio /    1.00 para superficie(01)
0 * / 00:00:00:000 / top(29) para superficie(01)
```

Figure 54. Fragment of calor.log01

6.3 Partition Debug Info

The partition debug info file lists all the DEVS processors that are taking part of the simulation, their IDs and they machine they are running in. This file is useful to were the coordinators for coupled models are placed. One partition debug info file is created by each LP. The files will be named with the text after the command line –**D** argument followed by the LP number.

Figure 7.6 shows a fragment of a partition debug file generated when running the model described in Figure 7.2 with the partition shown next.

```
0 : generadorCalor generadorFrio
0 : superficie(0,0)..(2,4)
1 : superficie(3,0)..(4,4)
```

Figure 55. Partition for the heat diffusion model of Figure 7.2

```
Model: ParallelRoot
 Machines:
        Machine: 0  ProcId: 0 < master >

Model: top
 Machines:
        Machine: 0  ProcId: 30 < master >

Model: superficie
 Machines:
        Machine: 0  ProcId: 1 < master >
        Machine: 1  ProcId: 2 < local >

Model: superficie(0,0)
 Machines:
```

```
        Machine: 0  ProcId: 3 < master >

...

Model: superficie(3,0)
 Machines:
        Machine: 1  ProcId: 18 < local >  < master >

Model: superficie(3,1)
 Machines:
        Machine: 1  ProcId: 19 < local >  < master >

Model: superficie(3,2)
 Machines:
        Machine: 1  ProcId: 20 < local >  < master >


Setting up the logical process
Total objects: 31
Local objects: 11
Total machines: 2

About to create the LP
LP has been created. Now registering processors.
 Registering processor superficie(2)
 Registering processor superficie(3,0)(18)
 Registering processor superficie(3,1)(19)
 Registering processor superficie(3,2)(20)
 Registering processor superficie(3,3)(21)
 Registering processor superficie(3,4)(22)
 Registering processor superficie(4,0)(23)
 Registering processor superficie(4,1)(24)
 Registering processor superficie(4,2)(25)
 Registering processor superficie(4,3)(26)
 Registering processor superficie(4,4)(27)

Current processors:
Processor Id: 2      Description: superficie
 Model Id: 2 superficie(02)
 Parent Id: 30

...

Processor Id: 27     Description: superficie(4,4)
 Model Id: 27 superficie(4,4)(27)
 Parent Id: 2
All objects have been registered!
Initializing Object superficie(2): OK
Initializing Object superficie(3,0)(18): OK
Initializing Object superficie(3,1)(19): OK
Initializing Object superficie(3,2)(20): OK
Initializing Object superficie(3,3)(21): OK
Initializing Object superficie(3,4)(22): OK
Initializing Object superficie(4,0)(23): OK
Initializing Object superficie(4,1)(24): OK
Initializing Object superficie(4,2)(25): OK
Initializing Object superficie(4,3)(26): OK
Initializing Object superficie(4,4)(27): OK
After Initialize....OK
```

Figure 56. Partition debug information file calor.pardeb01 (LP 1)

6.4 Output generated by the Parser Debug Mode

When the simulator is invoked with the option **–p**, the debug mode for the parser is activated. In debug mode, the parser will write the parse tree as it reads the rules. All tokens that are successfully processed are shown and if there is a syntax error, the place were the error was detected is specified.

Figure 7.7 shows the output generated for the *Game Life* model as implemented in section 12.1.

```
********* BUFFER ********
 1 100 { (0,0) = 1 and (truecount = 3 or truecount = 4) } 1 100 { (0,0) = 0
and truecount = 3 } 0 100 { t } 0 100 { t }
Number 1 analyzed
Number 100 analyzed
Number 0 analyzed
Number 0 analyzed
OP_REL parsed (=)
Number 1 analyzed
AND parsed
COUNT parsed (truecount)
OP_REL parsed (=)
Number 3 analyzed
OR parsed
COUNT parsed (truecount)
OP_REL parsed (=)
Number 4 analyzed
Number 1 analyzed
Number 100 analyzed
Number 0 analyzed
Number 0 analyzed
OP_REL parsed (=)
Number 0 analyzed
AND parsed
COUNT parsed (truecount)
OP_REL parsed (=)
Number 3 analyzed
Number 0 analyzed
Number 100 analyzed
BOOL parsed (t)
Number 0 analyzed
Number 100 analyzed
BOOL parsed (t)
```

Figure 57. Output generated in the Parser Debug Mode for the Game of Life

6.5 Rule evaluation debugging

Using the **–v** command line argument, a debug mode for cell rules evaluation is enabled. This will cause the simulator to log all intermediate values for each rule as it is evaluated.

Figure 7.8 shows a fragment of the output generated for the Game of the Life model of section 12.1. Line numbers have been added to make the following explanations clear.

The first two lines indicate the beginning of a new evaluation. Line 2 begins the evaluation of the first rule for the first cell. Each evaluated argument is listed with the partial result for the expression. Line 2 shows the evaluation of the cell reference (0,0), which turned out to be 0. In line 3, the integer constant 1 is evaluated, which is later compared to 0, evaluating to 0 (false). *BinaryOp* indicates that a binary operation is being performed. The operator name will be included between brackets, as well as the value of each of the operands.

Line 13 shows the final result for the condition of the rule, which was false in this case.

```
00 +----------------------------------------------------------------------+
01 New Evaluation:
02 Evaluate: Cell Reference(0,0) = 0
03 Evaluate: Constant = 1
04 Evaluate: BinaryOp(0, 1) =  (=) 0
05 Evaluate: CountNode(1) = 1
06 Evaluate: Constant = 3
07 Evaluate: BinaryOp(1, 3) =  (=) 0
08 Evaluate: CountNode(1) = 1
09 Evaluate: Constant = 4
10 Evaluate: BinaryOp(1, 4) =  (=) 0
11 Evaluate: BinaryOp(0, 0) =  (or) 0
12 Evaluate: BinaryOp(0, 0) =  (and) 0
13 Evaluate: Rule  = False
14
15 Evaluate: Cell Reference(0,0) = 0
16 Evaluate: Constant = 0
17 Evaluate: BinaryOp(0, 0) =  (=) 1
18 Evaluate: CountNode(1) = 1
19 Evaluate: Constant = 3
20 Evaluate: BinaryOp(1, 3) =  (=) 0
21 Evaluate: BinaryOp(1, 0) =  (and) 0
22 Evaluate: Rule  = False
23
24 Evaluate: Constant = 1
25 Evaluate: Rule  = True
26
27 Evaluate: Constant = 100
28 Evaluate: Constant = 0
29 +----------------------------------------------------------------------+
30 ...
31 ...
32 ...
33 ...
34 +----------------------------------------------------------------------+
35 New Evaluation:
36 Evaluate: Cell Reference(0,0) = 1
37 Evaluate: Constant = 1
38 Evaluate: BinaryOp(1, 1) =  (=) 1
39 Evaluate: CountNode(1) = 4
40 Evaluate: Constant = 3
41 Evaluate: BinaryOp(4, 3) =  (=) 0
42 Evaluate: CountNode(1) = 4
43 Evaluate: Constant = 4
44 Evaluate: BinaryOp(4, 4) =  (=) 1
45 Evaluate: BinaryOp(0, 1) =  (or) 1
46 Evaluate: BinaryOp(1, 1) =  (and) 1
47 Evaluate: Rule  = True
48
49 Evaluate: Constant = 100
50 Evaluate: Constant = 1
51 +----------------------------------------------------------------------+
52 ...
53 ...
54 ...
```

Figure 58. ***Fragment of the output generated by the debug mode for the Evaluation or Rules***

8 Model simulation

8.1 Simulation through the console

In this section you will be guided on running a simulation using the command line. Once again the existing model, called `life` will be used. The life example can be downloaded at http://www.sce.carleton.ca/faculty/wainer/wbgraf as mentioned in section 1. To start the simulation, unzipped the life example to a folder. Copy the simulator file, `simu.exe`, into the folder, where the life example was unzipped. Run the batch file (`demo.bat`) to simulate the model. This batch file will execute simu.exe with the defined parameters.

To simulate the life model manually (without batch file) type the following command in the command line:

```
simu –mlife.ma –t00:01:00:000 –llife.log
```

Once the simulation is finished, a log file is created and can be viewed. Further information about simulating through the command line will be explained in section 8.1

Note: The `life` model example simulated in this section is a Cell-DEVS model. If a DEVS model is to be simulated, the model must be compiled first. This will be explained in more detail in section 8.1.

To configure the execution of the simulator, the following parameters are valid:

–h: shows this help:

```
simu [-ehlmotdpvbfrsqw]
e: events file (default: none)
h: show this help
l: message log file (default: /dev/null)
m: model file (default : model.ma)
o: output (default: /dev/null)
t: stop time (default: Infinity)
d: set tolerance used to compare real numbers
p: print extra info when the parsing occurs (only for cells models)
v: evaluate debug mode (only for cells models)
b: bypass the preprocessor (macros are ignored)
f: flat debug mode (only for flat cells models)
r: debug cell rules mode (only for cells models)
s: show the virtual time when the simulation ends (on stderr)
q: use quantum to calculate cells values
w: sets the width and precision (with form xx-yy) to show numbers
```

–**e**: External events filename. If this parameter is omitted, the simulator will not use external events. The format used to describe the external events is showed in the section 6.3.

–**l**: Log filename. This file is used to store the messages received and emitted by each model within the simulation. If this parameter is omitted, the simulator will not generate activity log. If you wish to get the log on standard output, you should write **–l**). The format used by the log is described in the section 7.2.

–**m**: Model description filename. This parameter indicates the name of the file that contains the description of all models to simulate. If this parameter is omitted, the simulator will try to load the models from the *model.ma* file.

–**o**: output filename. This parameter indicates the name of the file that will be used to store the output generated by the simulator. If this parameter is omitted, the simulator will not generate any output. If you wish to get the results on standard output, simply write **–o**. The format of this

output is showed in the section 7.1.

–**t**: Sets the maximum time to simulate. If this parameter is omitted, the simulator will stop only when it will not have more events (internal or external). The format used to set the time is HH:MM:SS:MS, where:

HH: hours
MM: minutes (0 to 59)
SS: seconds (0 to 59)
MS: thousandth of second (0 to 999)

–d: Defines the tolerance used to compare real numbers. The value passed with the **–d** parameter will be used as the new tolerance value. By default, the value used is 10^-8.

–p: Shows additional information on parsing the cell model's rules. The parameter must be accompanied with the filename that will be used to store the detail. This mode is useful when a syntax error occurs on complex rules.
The format used to store the output is showed in the section 7.4.

–v: Enable the debug mode on the evaluation of all cell model's rules. For each rule to be evaluated it will be showed the results of the evaluation of each function and operator that they compose it. In addition, this mode evaluates the rules in complete form, that is, it doesn't use the rule's optimization. The parameter must be accompanied with the filename that will be used to store the rule's evaluation. The format of the output generated when this mode is enabled is showed in the section 7.5.

–b: Bypass the preprocessor. When this parameter is set, the macros will be ignored.

–f: Enable the debug mode on flat cell models. This allows to show the state of a flat-coupled model on each time change. When you used flat models, the simulation process does not send messages between the atomic cells that compound it, and then, the log will not store these messages. When you run the *DrawLog*, it will be unable to show the state of the model at each time. The parameter must be accompanied with the filename that will be used to store the states. If you wish to show the results on the standard output, simply write **–f**.

–r: Enable the debug mode that validates the rules used to define the behaviour of the cells models. When this mode is enabled, the simulator checks for the existence of multiple valid rules at runtime. If this condition is true, the simulation will be aborted. This mode is available only in standalone mode. There are special cases to consider: if you are using a stochastic model (i.e. the model uses random numbers generators) must happen that multiple rules will be valid, or than none of them will be. In both cases, the simulator will notify this situation to the user, showing a warning message on standard output, but the simulation will not be aborted. For the former case, the first valid rule will be considered. For the second case, the cell will have an undefined value (**?**), and the delay time will be the default delay time specified for the model. If this parameter is not used when the simulator is invoked, the mode is disabled and only will be considered the first valid rule.

–s: Show the simulation's end time on stdErr.

–q: Allows to use a *quantum* value. This permit to quantify the value returned by the local computing function evaluated on each cell of the model. Thus, all the values will be rounded to

the near maximum multiple of the quantum value minor than the original value. This mechanism decreases the number of messages transmitted in the simulation, but the results of the simulation will not be exact. For example, if the quantum value is 0.01 and the value returned by the local computing function is 0.2371, the state of the cell will be 0.23. The value used as quantum must be declared next to the parameter–q, for example: to set the quantum value as 0.01 the parameter must be –q0.001. If the *quantum* value is 0 or the parameter **–q** is not used, the use of the quantum will be disabled, and the value returned by the local computing function will be directly the value of the cell.

–w: Allows to set the wide and precision of the real values displayed on the outputs (log file, external events file, evaluation results file, etc). By default, the wide is 12 characters and the precision is of five digits. Thus, of the 12 characters of wide, 5 will be for the precision, 1 for the decimal point, and the rest will be used for the integer part that will include a character for the sign if the value is negative. To set new values for the wide and precision, the –w parameter must be used, followed of the number of characters for the wide, a hyphen, and the number of characters for the decimal part. For example to use a wide of 10 characters and 3 for the decimal digits, you must write **–w10–3**.
Any numerical value that must be showed by the simulator will be formatted using these values, and it will be rounded if necessary. Thus, if a cell has the value 7.0007 and the parameter –w10–3 is declared on the invocation of the simulator, the value showed for the cell on all outputs will be 7.001, but the internal value stored will not be affected.

The drawlog command is another simulation tool provided by CD++. It is use to view the state of a cellular model after every simulation cycle as the simulation advances. Please refer to section 9 for more detailed information about the drawlog command.

8.2 Simulation through Eclipse

CD++Builder plugin offers many tools to simulate/view CD++ models. This section describes how to use CD++Builder to create a model from a smulator. In order to use CD++Builder, you must use the CD++Builder perspective. This can be done by selectin “Window” on the top menu bar, selecting “Open Perspective” and then clicking on “Other”. The following perspective screen will pop up where you can select the CD++Builder Perspective

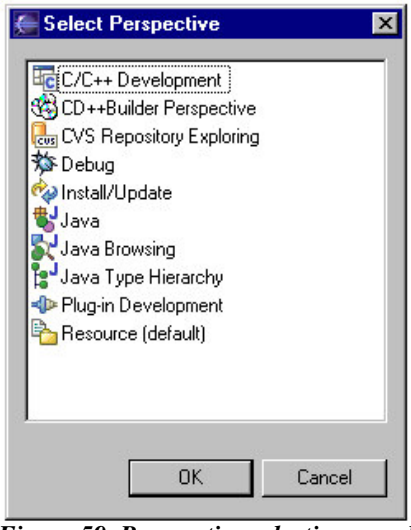


Figure 59. Perspective selection panel

The CD++ tool set provided by the CD++Builder plugin are the following.

8.2.1 Compiling a new model

To compile a CD++ m, select any file in the project you want to compile and click on the “Build” button . The project must contain DEVS models only. This button will automatically create a makefile for you that is unique fo  our project and then it runs the make command on the makefile. This takes all the cpp file from the project and compiles it. It then creates the simu executable that is necessary run the simulation button. Before the compilation takes place, the build tool will ask if you want to run this tool in verbose mode.

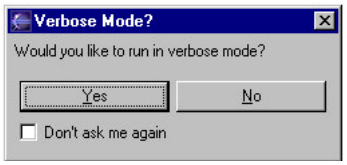


Figure 60. Verbose Mode Message

This means that it will list out all the directions and messages the tool outputs when it is compiling.

8.2.2 Simulating a model

To simulate a project, you can click on the simu button  .This will bring up a panel (shown in figure 8.3) where you can specify your parameters for running a simulation.

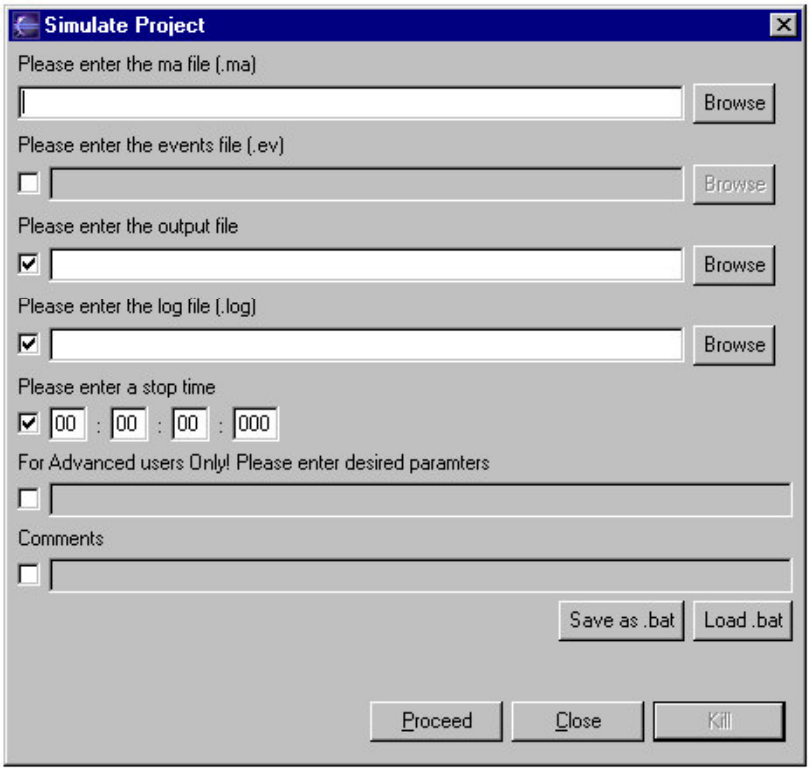


Figure 61. Simulation panel

You can enable each of the parameters by checking its respective box next to its name. Some parameters such as .ma, .ev, .out and .log, require a file input. If you know the name of your file that is in your project, you can type them in. If you would like to choose from a list of files, you can select the “browse” button which will

present you with a list of all the file types.

The last three parameters are the stop time, advanced settings and comments. The stop time is separated into 4 different type, hour, minute, second and millisecond. To enter a particular type of the stop time, you can simply enter the time or you can use the up/down arrow from the keyboard to set the desired time. This time indicates when you want the simulation to stop.

The advanced parameter box is used for users who want to run less-common parameters in the simulation. Finally, the comments section is used to enter fascinating comments about the current selection of parameters. This portion is generally used for saving your settings to a file or saving your settings to a batch file.

When you save settings to a file/batch, you will have to specify the name of the file you want to save it to. When you load settings from a file/batch, you will specify its location. By clicking proceed, the simulation will start and send all the information to the bottom component of the panel. Clicking on the output text and moving up and down with the up and down keys, will let you review the simulation and it's results. If your coupling file and/or events file is non existent or corrupted, the program will send an error pop up window to tell you. (Note – you must be in the current project of the project you want to simulate.)

8.2.3 Creating drawlog file for models

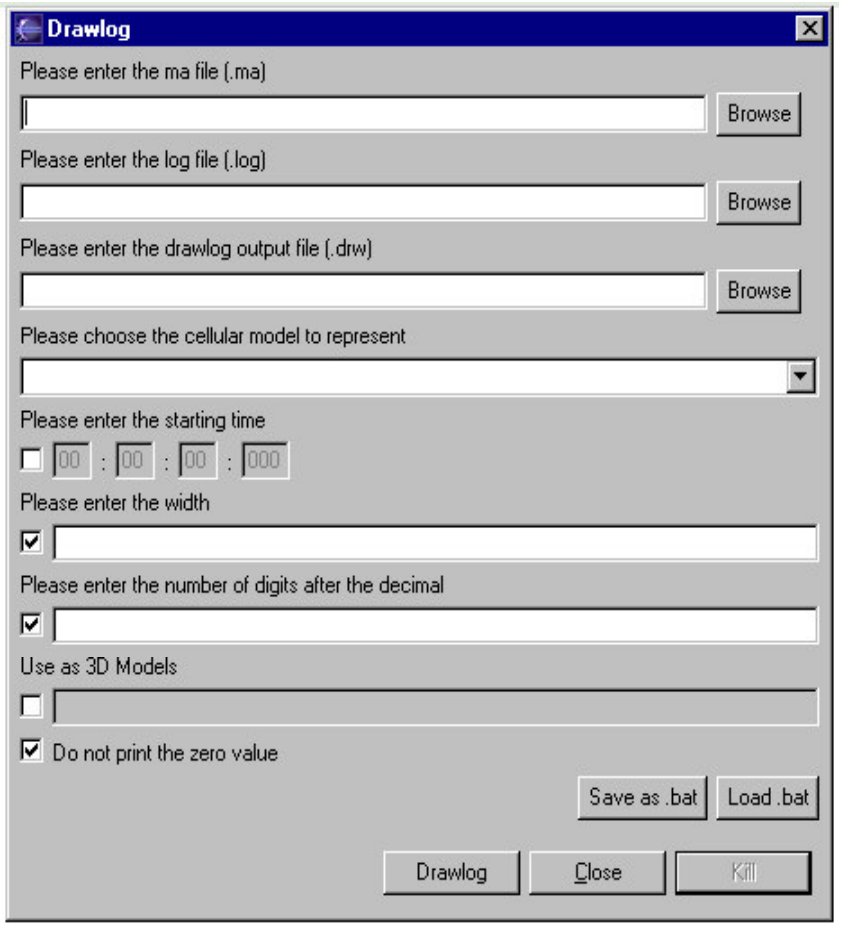


Figure 62. Drawlog panel

The DrawLog utility is used to view the state of a cellular model after each simulation cycle as the simulation advances. Using the log as input, drawlog parses the Y messages to update the state of each cell in the model. When a simulation cycle finishes, the state of the whole model is printed.

To start the drawlog tool, you can click on the drawlog button  .This will bring up a panel (shown in figure 8.4) where you can specify your parameters for running the drawlog just like the simulation button.

You can enable each of the parameters by checking its respective box next to its name. Parameters such as .ma, .log and the output file .drw, are required inputs. If you know the name of these files that are in your project, you can type them in. If you would like to choose from a list of files, you can select the "browse" button which will present you with a list of all the file types.

Once a .ma file is selected, a drop down menu will be available from the cellular model text field allowing the user to choose the desired model to be drawn. The other parameters that can be entered are the stop time, the width and precision used to represent numeric value, the number of slice shown when representing a 3D model and the choice to print the zero value.

The stop time is also separated into 4 different type, hour, minute, second and millisecond. To enter a particular type of the stop time, you can simply enter the time or you can use the up/down arrow from the keyboard to set the desired time. This time indicates when you want the simulation to stop.

Similar to the SIMU button, an user has the choice to save the settings used in the drawlog to a batch file. When you save settings to a file/batch, you will have to specify the name of the file you want to save it to. When you load settings from a file/batch, you will specify its location. By clicking proceed, the simulation will start and send all the information to the bottom component of the panel. Clicking on the output text and moving up and down with the up and down keys, will let you review the simulation and it's results. If your coupling file and/or events file is non existent or corrupted, the program will send an error pop up window to tell you. (Note – you must be in the current project of the project you want to run the draw log.)

8.2.4 CD Modeler

To launch the CD Modeler, click on the CD Modeler button  . The CD Modeler is a GUI, which is used for creating atomic models and coupled models for the ND-C++ tool. The basic function of the GUI includes: create atomic model, create coupled model, retrieve parent class of the coupled model, and run external DOS-style command. The GUI also includes a simple text editor.This GUI is shown in figure 8.5. For more information about the CD Modeler please refer to the user manual in Appendix E (DRAFT VERSION).

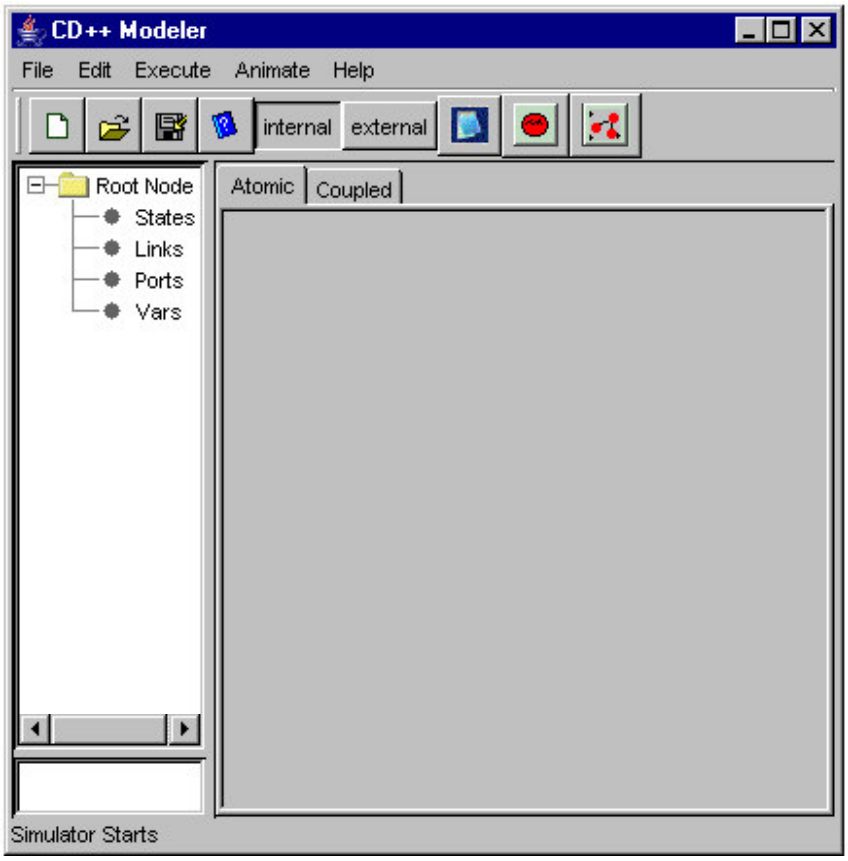


Figure 63. CD Modeler panel

6.6 Parallel simulator

To run CD++, type the following command:

```
./mpirun -np n ./cd++ [-ehlmotdvbfrspqw]
```

Here n indicates the number of machines that will be required. It is important this is the same number of machines specified in the partition file or the simulation will not work.

Usage:

```
./cd++ [-ehlLmotdpPDvbfrsqw]
 e: events file (default: none)
 h: show this help
 l: logs all messages to a log file (default: /dev/null)
 L[I*@XYDS]: log modifiers (logs only the specified messages)
 m: model file (default : model.ma)
 o: output (default: /dev/null)
 t: stop time (default: Infinity)
 d: set tolerance used to compare real numbers
 p: print extra info when the parsing occurs (only for cells models)
 D: partition details file (default: /dev/null)
 P: parallel partition file (will run parallel simulation)
 v: evaluate debug mode (only for cells models)
 b: bypass the preprocessor (macros are ignored)
 f: flat debug mode (only for flat cells models)
```

```
r: debug cell rules mode (only for cells models)
s: show the virtual time when the simulation ends (on stderr)
q: use quantum to compute cell values
y: use dynamic quantum (strategy 1) to compute cells values
Y: use dynamic quantum (strategy 2) to compute cells values
w: sets the width and precision (with form xx-yy) to show numbers
```

Figure 64. CD++ command line options

The command line options allowed are:

–**efilename**: External events filename. If this parameter is omitted, the simulator will not use external events. The format for external event files is described in section 6.3

–**lfilename**: Log filename. When this parameter is specified, all messages received by each DEVS processor will be logged. If filename is omitted (only **–l** is specified) all log activity will be sent to the standard output. But if a filename is given, one log file will be created for each DEVS processor. The file **filename** will list all models and the name of the corresponding logfiles. These file will be named **filename.XXX** where XXX is a number. When this option is used and no addition log modifiers are defined, all received messages are logged.

The log file format is described in the section 7.2

–**L[I*@XYDS]**: allows to define which messages will be logged. This option is useful to reduce the log overhead. The following messages are supported:

I :	Initialization messages
* :	(*,t) Internal messages.
@:	(@,t) Collect messages
X:	(q,t) External messages
Y:	(y,t) Output messages
D:	(done,t) Done messages
S:	All sent messages

When using drawlog, only Y messages are required. Use the –LY option to reduce execution time.

–**mfilename**: Model filename. This parameter indicates the name of the file that contains the model definition. If this parameter is omitted, the simulator will try to load the models from the *model.ma* file.

-**Pfilename**: Partition definition filename. A partition file is used to specify the machine where each atomic model will run on. Only the location of the atomic models needs to be specified. CD++ will then determine where the coordinators should be placed.

This file is only required for parallel simulation. If standalone simulation is used, this setting will be ignored.

The format for a partition file is described in section 6.4.

–**ofilename**: output filename. This parameter indicates the name of the file that will be used to store the output generated by the simulator. If this parameter is omitted, the simulator will not generate any output. If you wish to get the results on standard output, simply write **–o**.

The format for the generated output is described in section 7.1.

–Dfilename: debug filename for partition debug information. When this option is used, one file for each LP will be created. This file will list all the identification of all DEVS processors running on it.

–**t**: Sets the simulation finishing time. If this parameter is omitted, the simulator will stop only when there are no more events (internal or external) to process. The format used to set the time is HH:MM:SS:MS, where:

HH:	hours
MM:	minutes (0 to 59)
SS:	seconds (0 to 59)

MS: thousandths of second (0 to 999)

–d: Defines the tolerance used to compare real numbers. The value passed with the **–d** parameter will be used as the new tolerance value.
By default, the value used is 10^{-8}.

–**pfilename**: Shows additional information when parsing a cell's local transition rules. The parameter must be accompanied with the name of the file that will be used to store the detail. This mode is useful when a syntax error occurs on complex rules.
The format used to store the output is showed in the section 7.4.

–vfilename: Enables verbose evaluation of the local transition rules. For each rule that is evaluated, the result of each function and operator will be showed. In addition, this mode will cause complete evaluation of the rules, i.e. it doesn't use rule optimization. The parameter must be accompanied with the filename that will be used to store the evaluation results.

The format of the output generated when this mode is enabled is described in section 7.5.

–b: Bypass the preprocessor. When this parameter is set, the macros will be ignored.

–r: Enables the rule checking mode. When this mode is enabled, the simulator checks for the existence of multiple valid rules at runtime. If this condition is true, the simulation will be aborted. This mode is available in standalone mode.

There are a few special cases to consider: if a stochastic model is used (i.e. a model that uses random numbers generators) it might either happen that multiple rules are be valid or that none of them is. In any case, the simulator will notify this situation to the user, showing a warning message on standard output, but the simulation will not be aborted. For the first case, the first valid rule will be considered. For the second case, the cell will have an undefined value (**?**), and the delay time will be the default delay time specified for the model.
If this parameter is not used when the simulator is invoked, the mode is disabled and only will be considered the first valid rule.

–s: Show the simulation's finishing time on stderr.

–qvalue: Sets the value for the *quantum*.

The value used as quantum must be declared next to the parameter–q, for example: to set the quantum value as 0.01 the parameter must be –q0.001.

If the *quantum* value is 0 or the parameter **–q** is not used, the use of the quantum will be disabled, and the value returned by the local computing function will be directly the value of the cell.

–w: Allows to set the wide and precision of the real values displayed on the outputs (log file, external events file, evaluation results file, etc).
By default, the wide is 12 characters and the precision is of five digits. Thus, of the 12 characters of wide, 5 will be for the precision, 1 for the decimal point, and the rest will be used for the integer part that will include a character for the sign if the value is negative.
To set new values for the wide and precision, the –w parameter must be used, followed of the number of characters for the wide, a hyphen, and the number of characters for the decimal part. For example to use a wide of 10 characters and 3 for the decimal digits, you must write **–w10–3**.
Any numerical value that must be showed by the simulator will be formatted using these values, and it will be rounded if necessary. Thus, if a cell has the value 7.0007 and the parameter –w10–3 is declared on the invocation of the simulator, the value showed for the cell on all outputs will be 7.001, but the internal value stored will not be affected.

7 Utility programs

7.1 Drawlog

The DrawLog utility is used to view the state of a cellular model after each simulation cycle as the simulation advances. Using the log as input, drawlog parses the Y messages to update the state of each cell in the model. When a simulation cycle finishes, the state of the whole model is printed.

Drawlog can read the log from a file or from the standard input. Its command line parameters are shown next:

```
drawlog -[?hmtclwp0]

where:
        ?       Show this message
        h       Show this message
        m       Specify file containing the model (.ma)
        t       Initial time
        c       Specify the coupled model to draw
        l       Log file containing the output generated by SIMU
        w       Width (in characters) used to represent numeric values
        p       Precision used to represent numeric values (in characters)
        0       Don't print the zero value
        f       Only cell values on a specified slice in 3D models
```

Figure 65. Help shown by DrawLog

–?: similar to –**h**.

–m: Specifies the filename that contains the definition of the models. This parameter is required

–t: Starting time. Sets the time for the first state output. If not specified, 00:00:00:000 will be used.

–c: Name of the cellular model to represent. This parameter is obligatory required because a .ma file may define more than one cellular model.

–l: Name of the log file. If this parameter is omitted, *Drawlog* will take the data of the standard input.

–w: Allows to define the print width, in characters, for numeric values. This width will include the decimal point and sign. For example, **–w7** defines a fixed size for each value of 7 positions. Small numbers will be padded with spaces.

By default, *Drawlog* uses a width of 10 characters. For correct results a width that is bigger than the precision (defined with the parameter **–p**) + 3 is recommended.

–p: Defines the number of digits to be displayed after the decimal point. If a value of 0 is used, then all the real values will be truncated to integer values. This parameter is generally used in combination with the option **–w**.

As an example, consider using the command line arguments **–w6 –p2.** This will set the

By default, *DrawLog* assumes 3 characters for the precision.

–0: When this option is specified, a value of 0 zero will no be shown.

-**f**: Draws a 3D model as a 2D model. Only the specified plane will be drawn. To draw plane 0, -**f**0 should be used.

Figure 9.2 shows two different ways of starting drawlog. The first uses a log file as input. The second one, instead, takes its input from the standard input.

```
drawlog –mlife.ma –clife –llife.log –w7 –p2 –0

or

pcd –mlife.ma –l– | drawlog –mlife.ma –clife –w7 –p2 –0
```

Figure 66. Examples for the invocation to DrawLog

When parallel simulation is used, the standard input can not be directly used by drawlog because log messages may arrive out of order. Therefore, it is necessary to sort the messages first. A utility called logbuffer (described next) has been written for that purpose.

The output format of *DrawLog* will depend on the number of dimensions of the cellular model.

- Output for bidimensional cellular models.
- Output for three–dimensional cellular models.
- Output for cellular models with 4 or more dimensions.

7.1.1 Bidimensional cellular models

A 2 dimensions model will be displayed as a matrix of values. Figure 9.3 shows a fragment of the output generated by DrawLog for a two-dimensional model of size (10, 10). The number width has been set to 5 and the precision to 1.

```
Line : 238 – Time: 00:00:00:000
         0    1    2    3    4    5    6    7    8    9
     +-------------------------------------------------+
   0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   2| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   5| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
     +-------------------------------------------------+

Line : 358 – Time: 00:00:01:000
         0    1    2    3    4    5    6    7    8    9
     +-------------------------------------------------+
   0| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   1| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   2| 24.0 24.0 35.8 24.0 24.0 24.0 24.0 24.0 –6.3 24.0|
   3| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   4| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   5| 24.0 24.0 24.0 24.0 24.0 39.5 24.0 24.0 24.0 24.0|
   6| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   7| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
   8| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 –4.0 24.0|
   9| 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0 24.0|
     +-------------------------------------------------+
```

Figure 67. Fragment of the output generated for a bidimensional cellular model

7.1.2 Three dimensional models

For three dimensional models, a matrix representation will be used. Each matrix is one plane of the cell space. The first plane shown will correspond to (x,y,0), the second one to (x,y,1), and so on.

Figure 9.4 shows the output of *Drawlog* when used to draw a cellular space of size (5,5,4) with a number width of 1, a precision of 0 and zero values not displayed.

```
Line : 247 - Time: 00:00:00:000
     01234       01234       01234       01234
    +-----+     +-----+     +-----+     +-----+
   0|1    |    0|     |    0|1    |    0|     |
   1|1 1  |    1|11  1|    1|  111|    1|   11|
   2| 1   |    2|   11|    2| 1 11|    2|    1|
   3|     |    3|  1  |    3|    1|    3|    1|
   4|  1 1|    4|  1 1|    4| 1  1|    4|    1|
    +-----+     +-----+     +-----+     +-----+

Line : 557 - Time: 00:00:00:100
     01234       01234       01234       01234
    +-----+     +-----+     +-----+     +-----+
   0|     |    0|11 11|    0|1  11|    0|   11|
   1|     |    1|     |    1|1    |    1|    1|
   2|     |    2|1   1|    2|1    |    2|   11|
   3|   1 |    3|  11 |    3|1  11|    3|1   1|
   4|     |    4|     |    4|     |    4|     |
    +-----+     +-----+     +-----+     +-----+

Line : 829 - Time: 00:00:00:200
     01234       01234       01234       01234
    +-----+     +-----+     +-----+     +-----+
   0|     |    0|     |    0|1   1|    0|     |
   1|    1|    1|    1|    1|   11|    1|    1|
   2|     |    2|     |    2|1   1|    2|     |
   3|     |    3|     |    3|1  1 |    3|     |
   4|     |    4|    1|    4|1  11|    4|    1|
    +-----+     +-----+     +-----+     +-----+
```

Figure 68. Fragment of the output generated for a three–dimensional cellular model

7.1.3 Cellular models of more than 3 dimensions

For models of 4 or more dimensions, the matrix representation will not be used. Instead, the values for each cell will be listed. The options defined with –**p, -w** and –**0** will be ignored.

Figure 9.5 shows a fragment of the output generated by *DrawLog* for a model of size (2, 10, 3, 4).

```
Line : 506 - Time: 00:00:00:000
 (0,0,0,0) = ?
 (0,0,0,1) = 0
 (0,0,0,2) = 9
 (0,0,0,3) = 0
 (0,0,1,0) = 21
 ...     ...     ...
 ...     ...     ...
 (1,9,1,0) = 0
 (1,9,1,1) = 4.333
 (1,9,1,2) = 0
```

```
 (1,9,1,3) = -2
 (1,9,2,0) = 6
 (1,9,2,1) = 0
 (1,9,2,2) = 7
 (1,9,2,3) = 0


Line : 789 - Time: 00:00:00:100
 (0,0,0,0) = 0
 (0,0,0,1) = 0
 (0,0,0,2) = 13.33
 (0,0,0,3) = 0
 (0,0,1,0) = 5.75
 ...    ...    ...
 ...    ...    ...
 (1,9,1,0) = 6.165
 (1,9,1,1) = 2
 (1,9,1,2) = 0
 (1,9,1,3) = 1.14
 (1,9,2,0) = 0
 (1,9,2,1) = 0
 (1,9,2,2) = 5.25
 (1,9,2,3) = 0
```

Figure 69. Fragment of the output generated for a model with dimension 4

7.2 Random Initial States – MakeRand

MakeRand is a tool to create a .val file with a random initial state for a cellular model.

Usage:

```
makerand -[?hmcs]

where:
        ?       Show this message
        h       Show this message
        m       Specify file containig the model (.ma)
        c       Specify the Cell model within the .ma file
        s       Specify the value set
                 s0   = Use the values 0 & 1 (Uniform Distribution)
                 s1-n = Use the value 1 for n cells & 0 for the rest
                 s2-n = Makes random states for the Pinball Model
                 s3-n = Random states for the Gas Dispersion Model
```

Figure 70. MakeRand command line options

–?: similar to –**h**.
–m: Specifies the filename for the model definition file (.ma)
–c: Name of the cellular model. This parameter is required because the size of the model needs to be known.
–s: Specifies the type of initial state to be created:

–s0: For each cell of the model, a value will be chosen randomly belonging to the set {0, 1} with the same probability for each value.

–s1–n: Indicates that the model initially will have *n* cells with value 1 (distributed randomly according to an uniform distribution) and the rest of the cells will have the value 0. If *n* is bigger to the quantity of cells of the model, then an error will occur and the initial state

will not be generated.
For example, if we have a 40x40 cellular and we want 75% of the cells (1200 cells) to have an initial value of 1, and the remaining cells an initial value of 0, then **–s1–1200** should be used.

–s2–n: Generates a random initial state for the Pinball model. For this model a value between 1 and 8 will be randomly generated and randomly place inside the cellular space. In addition, *n* cells will be randomly chosen to represent the walls. The rest of the them will have an initial value of 0.

–s3–n: Creates an initial state for the gas dispersion model with n particles.

The output will be created in a .val file with the same name as the model file.

7.3 Converting .VAL files to Map of Values – *ToMap*

The tool *ToMap* allows to creates a .map (section 6.2) file from a .val file (section 6.1).

Usage:

```
toMap –[?hmci]

where:
        ?       Show this message
        h       Show this message
        m       Specify file containig the model (.ma)
        c       Specify the Cell model within the .ma file
        i       Specify the input .VAL file
```

Figure 71. Command line arguments for toMap

–?: same as –**h**. Shows the command line help.

–m: Specifies the filename (.ma file) with the model definition.

–c: Name of the cellular model.

–i: Specifies the name of the .val file that contains the list of values that it will be used for the creation of the .map file.

ToMap uses all values in the .val file to create a map of values. If the .val file does not specify a value for every cell, then the default value, as specified by the *InitialValue* parameter, will be used.

The output file will have the same name as the .ma file but the extension .map will be used instead.

7.4 Error Calculation

The program ERRORQ accept a DRAWLOG output and generates a new output (on standard output –must be redirected to a file--) with six columns. The DRAWLOG output must be wit the lines and times titles (parameters –f or –e). Other way, comparison is not possible, because a different offset on each file can occur.

These columns are:

0) Counter of times simulations.

1) Time of the block-simulation in comparison

2) (1-s/q)/n = Error introduced on the indicated time. Not summarized (only the error generated on this time).

3) sum[(1-s/q)]/n= The same as 2 but accumulated until current time.

4) (s-q)/n= The same as 2 but with a different formula. Not summarized (only the error generated on this time).

5) sum[s-q]/n= The same as 4 but accumulated until current time.

7.4.1 Columns showed

Column 0

Line Counter

Column 1

Simulation Time

Column 2

$(1-s/q) / n = \sum (1-s_i/q_i)/n$ (for 0<=i <=n = number of cells)

s_i = Value of cell i on the output without quantum.

q_i = Value of cell i on the output with quantum.

n = number of cells

This value is set to 0 after each block-time.

Column 3

(is the same as 2 but accumulated)

$sum[(1-s/q)]/n = \sum [\sum (1-s_i/q_i)]_j /n$ (for 0<=i <=n = number of cells, 0<=j<=current time)

s_i = Value of cell i on the output without quantum.

q_i = Value of cell i on the output with quantum.

n = number of cells

Column 4

$(s-q) / n = \sum (s_i-q_i)/n$ (for 0<=i <=n = number of cells)

s_i = Value of cell i on the output without quantum.

q_i = Value of cell i on the output with quantum.

n = number of cells

This value is set to 0 after each block-time.

Column 5

(is the same as 4 but accumulated)

sum[(s-q)]/n = $\sum [\sum (s_i-q_i)]_j /n$ (for 0<=i <=n = number of cells, 0<=j<=current time)

s_i = Value of cell i on the output without quantum.

q_i = Value of cell i on the output with quantum.

n = number of cells

When the error is accumulated, means the error accumulated until time indicated in column 1.

The program ERRORQ receives two arguments:

1) The name of the original draw output file (original means without quantum)

2) The name of the draw output file obtained with quantum

Both files must be generated with DrawLog without the option –f, because the time is neaded to synchronize the files and calculate the error. For example:

errorq output1.drw outputqq.drw

will show you:

```
Archivo original: sinq.drw quantificado:conq.drw
Dimension detectada del modelo: 5 x 5
  Descripcion de Columnas
    0) Contador de bloques comparados
    1) Tiempo de simulaicon del bloque en comparacion
    2) (1-s/q)/n= Error '/' introducido en el bloque time. Sin acumular.
    3) sum[(1-s/q)]/n= Error '/' acumulado hasta time.
    4) (s-q)/n= Error '-' introducido en el bloque time. Sin acumular.
    5) sum[s-q]/n= Error '-' acumulado hasta el bloque time.
0,1,2,3,4,5
t,time,(1-s/q)/25,sum[(1-s/q)]/25,(s-q)/25,sum[(s-q)]/25
0,00:00:00:000,0,0,0,0
1,00:00:00:230,0,0,0,0
2,00:00:00:325,0,0,0,0
3,00:00:00:355,0.00289157,0.00289157,0.24,0.24
4,00:00:00:595,0.0446277,0.0475192,0.459642,0.699642
5,00:00:00:680,0.0580914,0.105611,0.868588,1.56823
6,00:00:00:710,0.056741,0.162352,0.981349,2.54958
7,00:00:00:945,0.115058,0.27741,3.8915,6.44108
8,00:00:00:995,0.0714215,0.348831,1.33647,7.77755
9,00:00:01:030,0.0944933,0.443325,1.65318,9.43073
10,00:00:01:060,0.105529,0.548853,2.00511,11.4358
11,00:00:01:095,0.161452,0.710305,2.69788,14.1337
```

if you do (for example, in MSWindows):

Errorq output1.drw outputqq.drw > error.csv

This will generate a comma separated value (because the program shows the columns separated with commas) file and can be opened with Excel or a similar application.

7.5 GrafCell

The program GRAFCELL accept a DRAWLOG output and show you a graphic on the screen with all the cells and the function graphic for each one on a Grid.

The program GRAFCELL receives five or six arguments (depending of the use)

(On the next description, x means the values for x edges and y the values for y edges of the plane graphic)

3) The name of the DRAWLOG output file.

4) The minimum value for x (most times is 0 –cero—because of the starting time of a simulation is cero).

5) The minimum value for y. This is the minimum value that a cell can reach on the simulation.

6) The maximum value for x. This is the maximum time of the simulation, converted to milliseconds, but however, not all simulation times are showed on the drawlog, so you will nead to adjust this parameter trying with different ones until the correct scale of the grid is showed.

7) The maximum value for y. This is the maximum value that a cell can reach on the simulation.

8) <Optional> -t With "-t" argument, GrafCell will show the current time of the simulation when drawing. The time will be converted to milliseconds and divided by the default cell delay, but however, not all times simulations are showed on drawlog, so a better adjustment will be necessary. If the drawlog file does not include the titles and times, a counter will be showed. WARNING: Woth this option, the drawing can be VERY SLOW, depending on the model size.

NOTE: Arguments 2, 3, 4 and 5 are only to adjust the scale of the grid. You can try different ones to have a nicer view.

GrafCell output.log 0 -16.6 35000 94.3

This will show you the graphics on a newer window screen. The number of cells will be automatically detected by GrafCell.

RESTRICTIONS: GrafCell will work propertly only with nxm (and only with n=m) outputs (with drawlog outputs of one slide (parameter –f or –e).

*To graph an output of a simulation **with quantum**, is better to use a DRAWLOG output generated with –f option, because –f option shows outputs for all time simulations, and that produces a better graphic. However, this is a suggestion and the improvement depends on the model.*

7.6 Message counter

The program CONTART accept a simu LOG and gives you the number of messages used on that simulation.

The program CONTART receives one two arguments (depending of the use)

The name of the SIMU LOG file

<optional> The Time until you want to count messages.

contart output.log

This will show you:

```
Archivo a contar mensajes: output.log
Cantidad de mensajes en output.log hasta 00:02:00:000(EOF) -> 264878
#*=72290
#X=48000
#Y=24290
#D=120294
#I=4
```

This means that in the log output.log, until EOF (because the title "hasta 00:02:00:000(EOF) means that end of file was reached) there are a total of 264878 messages and this is the detail:

72290 messages of type "*"

48000 messages of type "X"

24290 messages of type "Y"

120294 messages of type "D"

4 messages of type "I"

If you use the second argument, the program will count messages until the indicated time is reached or EOF (the first that occurs).

With the same log as Example1, we can do:

contart output.log 00:01:10:250

And this will show you:

```
Archivo a contar mensajes: output.log
Hasta: 00:01:10:250
Cantidad de mensajes en output.log hasta 00:01:10:250 -> 155011
#*=42301
#X=28100
#Y=14201
#D=70405
#I=4
```

This means that in the log output.log, until 00:01:10:250 simulation time (because the second parameter is in use "hasta 00:01:10:250") there are a total of 155011 messages and the detailed messages. If you use, for this example,

contart output.log 00:03:00:000

You will get the same results as in Example1, because EOF will be reached before the 3 minutes indicated on the 2nd parameter, and you will see the title EOF as in example1.

This is useful when you have to compare logs of simulations ended at different simulation-time.

7.7 Bitmap Translator

For Cell DEVS, we can initialize cells by two ways: either initializing directly in ma file or in value file (val file). In many cases, initial data can be available in the form of image, and the goal of this utility is to convert image data into value file. Also we can use multiple images to initialize different planes in z-direction in case of 3D cell DEVS models.

The scope of this utility is standard 24-bit bitmap images. If image data is available in other forms, we can first convert that into 24-bitmap using any standard Image Viewer tool.

The tool bmptoval allows to convert bitmap image data into value file. The possible parameters are

```
Welcome to Bitmap Translator: Version 1.0
-----------------------------------------
bmptoval -[?hmcblupv]

Where:
        ?       Show usage help
        h       Show usage help
        m      Specify file containing the model (.ma)
        c       Specify the coupled model having dimension of model
        b       Bitmap file to be translated (.bmp)
        l        Initial value for normalization
        u       Maximum value for normalization
        p       Specify precision of index
        v       Specify Value file name
```

-h: show the above help

-?: same as h

-m: Specifies the file name that contains the definition of coupled model. This parameter is mandatory.

-c: Name of the cellular model to represent. This parameter is mandatory because the file specified with **–m** can contains the description of many models. Only cellular models are allowed.

-b: Name of the bitmap image file. This parameter can be repeated for multiple bitmap image files. This parameter is mandatory.

-l: Initial value for normalization. This is mandatory. User can convert color index to normalized scale and this parameter represents lower index for that scale range.

-u: Upper value for normalization. This is mandatory. User can convert color index to normalized scale and this parameter represents maximum index for that scale range.

-p: This is optional parameter to define the precision, in integers, of the normalized color index in the value file. Bitmap Translator assumes precision of 4 by default. his parameter should be more than 0.

```
 Using one image file

./bmptoval -mEdge.ma -cEdge -bImage1.bmp -l0 -u100 -p3

Using multiple image files

./bmptoval -mEdge.ma -cEdge -bImage1.bmp -bImage2.bmp -l0 -u100 -p3
```

Note: Multiple images should be used only for 3D CELL DEVS and size of 3rd dimension should not be less than number of images.

7.8 LTRANS (Lattice Translator)

Ltrans is a tool that implements the two functions mentioned before. This implementation only works in 2D models and only using nearest neighborhood. The last restriction is a necessary condition so that LTRANS works correctly. CD++ has a specification language to define the cells behavior based on rules and a neighborhood definition, LTRANS translates hexagonal or triangular rules to square CD++ compatible rules. LTRANS receives a set of rules based on a hexagonal or triangular geometry and translates it in rules based on square geometry to be included in a model to be simulated with CD++.

To run Ltrans, type:

```
ltrans [-hmotp]
```

where

h: show the help
m: model file (default : modelH.ma)
o: model translated file (default: model.ma)
t: mapping type (default: Hexagonal)
p: parser debug filename

The command line options allowed are:

-m*filename* : Model file. This parameter indicates the name of the file that contains the Rules to be translated. If this parameter is omitted, the simulator will try to load the rules from the modelH.ma file.

-o*filename* : Model Translated file. This parameter indicates the name of the file that contains the translated Rules. If this parameter is omitted, the simulator will try to save the rules to the model.ma file.

-t*[hexagonal/triangular]* : Mapping type. This parameter indicates the type of mapping hezagonal to square or triangular to square.

–**p**filename: Shows additional information when parsing a cell's local transition rules. The parameter must be accompanied with the name of the file that will be used to store the detail. This mode is useful when a syntax error occurs on complex rules.

A model file is used to define the rules to be translated. This file consists in a set of rules based on hexagonal o triangular geometry. The language used to modeling cell's behavior in a hexagonal or triangular geometry is the same that is used in CD++ but the only difference is the way that a neighbor is referenced. In CD++ a cell (belonging a 2D space) is referenced using a tuple (x,y) where x (row) and y (col) are relative position of a cells. As Ltrans only support nearest neighbors, was necessary define nearest neighbors for hexagonal and triangular geometry. Figure 5 shows the way to define nearest neighbors in each geometry. For both geometries each nearest neighbor is referencied using [*n*] where *n* is the number assigned to each nearest neighbor.

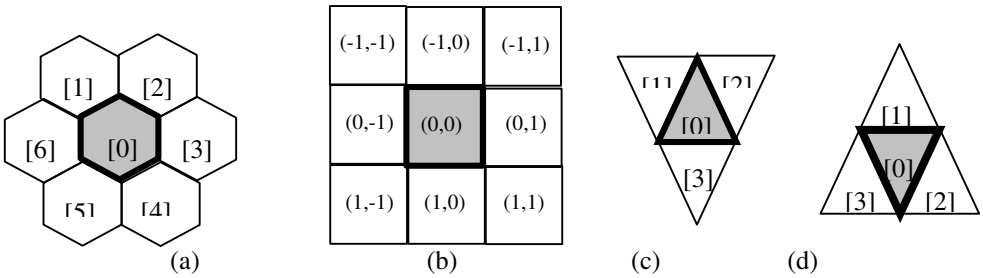


Figure 72. (a) nearest neighbors used for hexagonal geometry. (b) nearest neighbors used for square geometry (used in CD++). (c) and (d) nearest neighbors used for triangular geometry, note that there are two

different kind of nearest neighbor and depends on the orientation of the cell

The cellular space in a Cell-DEVS model is named grid or lattice. This lattice is a homogeneous and regular set of cells with an specific geometry. CD++ only allows square geometry, but there are others kind of geometries that could be used to modeling different phenomena. These are:

- **Triangular**: The advantage of this type of geometry is that every cell has a limited number of nearby neighbors (three) which in some models is very necessary. On the other hand, the disadvantages are the difficulty of representation and visualization.

- **Hexagonal**: The advantage of this type of geometry is the higher **isotropy**, that means that the simulations are more natural and in some cases it is absolutely necessary to simulate certain phenomena. The disadvantage is that it is very difficult to represent and to visualize.

Bearing in mind these advantages and disadvantages of every geometry a translator was developed to be used in conjunction with CD++, using two geometries translation functions **[Wei97]** for 2D lattice.

Different kind of functions can be writing but at the time of visualization's results the interpretation could be not easy. The main idea is to use a function that shifts alternate rows in opposite directions, as shown in Figure 1. That function maintains the boundary conditions in the square lattice. The visualization is very simple, ignoring the factor that introduces the shift of every second row, a cell in hexagonal space can be found in the square lattice (see colors).

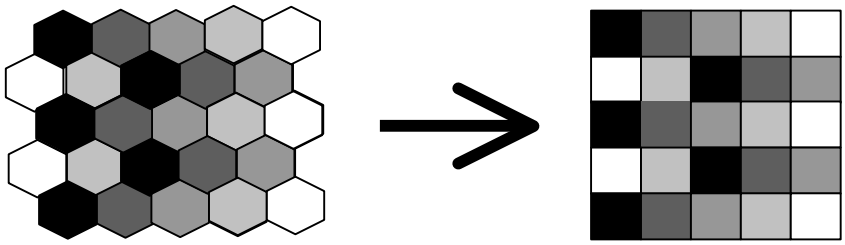


Figure 73. Shift mapping of the hexagonal lattice to the square lattice.

Let **(x,y)** the position of a cell, where **x** represent the row and **y** represent the column (remenber that the function only can be applied in 2D space). The neighborhood relation is transformed defferently depending on whether the row index **x** is even or odd, as shown in Figure 2.

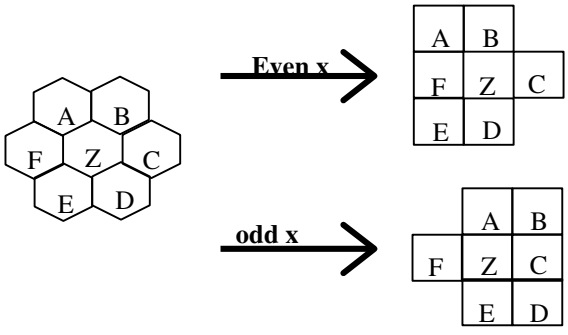


Figure 74. Neighborhood relation in hexagonal to square mapping function

The mapping of the triangular lattice to the square lattice is similar to the shift mapping for the hexagonal lattice. In the triangular case, every second cell has a different orientation. The mapping function is shown in Figure 3. Each row of triangles is mapped to one row of square depending on the parity of **x+y**.

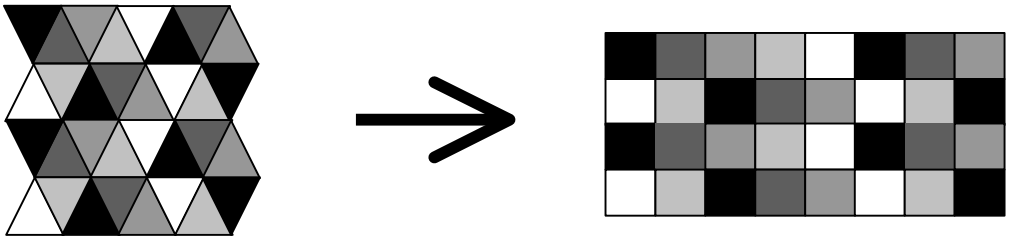


Figure 75. Visualization mapping of the triangular lattice to the square lattice.

The nearest neighborhood mapping is shown in Figure 4

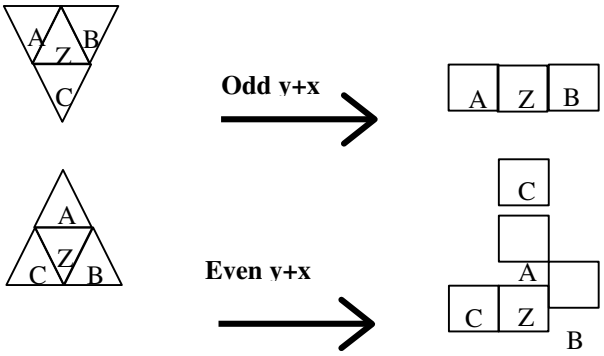


Figure 76. Nearest neighbors in the triangular mapping.

This translated file consists in the set of rules that can be simulated with CD++. Before to simulate the model, it must be completed with the other parameters that define a Cell-DEVS model (space dimention, type of border, default delay, etc.). Note: the neighborhood definition added must be the nearest neighbors as is show in figure 5b.

The Life Game was presented in Scientific American by the well known mathematician Martin Gardner. In this game, living cells will live or die. The rules for life evolution are as follows:

- An active cell will remain in this state if it has two or three active neighbors.
- An inactive cell will pass to active state if it has two active neighbors exactly.
- In any other case, the cell will die

The rules that define the Cell's behavior mentioned above in a hexagonal geometry is as follows:

```
rule: 1 100 { [0] = 1 and (truecount = 3 or truecount = 4) }
rule: 1 100 { [0] = 0 and truecount = 2 }
rule: 0 100 { t }
```

Figure 77. File lifegame.rules

Then we use LTRANS to translate the rules in a hexagonal space geometry as follow:

```
C:> ltrans -mlifegame.rules -thexagonal -olifegame.rules.HtoS
```

After applied LTRANS we obtain the following result:

```
rule: 1 100 { ( ( (0,0) = 1 ) and ( ( if((truecount - (if((-1,1) = 1,1,0)) -
(if((1,1) = 1,1,0))) < 0 , 0 , (truecount - (if((-1,1) = 1,1,0)) - (if((1,1)
= 1,1,0)))) = 3 ) or ( if((truecount - (if((-1,1) = 1,1,0)) - (if((1,1) =
1,1,0))) < 0 , 0 , (truecount - (if((-1,1) = 1,1,0)) - (if((1,1) = 1,1,0))))
= 4 ) ) ) and even(cellpos(1)) }
```

```
rule: 1 100 { ( ( (0,0) = 1 ) and ( ( if((truecount - (if((-1,-1) = 1,1,0))
- (if((1,-1) = 1,1,0))) < 0,0,(truecount - (if((-1,-1) = 1,1,0)) - (if((1,-
1) = 1,1,0)))) = 3 ) or ( if((truecount - (if((-1,-1) = 1,1,0)) - (if((1,-1)
= 1,1,0))) < 0,0,(truecount - (if((-1,-1) = 1,1,0)) - (if((1,-1) = 1,1,0))))
= 4 ) ) ) and odd(cellpos(1)) }

rule: 1 100 { ( ( (0,0) = 0 ) and ( if((truecount - (if((-1,1) = 1,1,0)) -
(if((1,1) = 1,1,0))) < 0 , 0 , (truecount - (if((-1,1) = 1,1,0)) - (if((1,1)
= 1,1,0)))) = 2 ) ) and even(cellpos(1)) }
rule: 1 100 { ( ( (0,0) = 0 ) and ( if((truecount - (if((-1,-1) = 1,1,0)) -
(if((1,-1) = 1,1,0))) < 0,0,(truecount - (if((-1,-1) = 1,1,0)) - (if((1,-1)
= 1,1,0)))) = 2 ) ) and odd(cellpos(1)) }

rule: 0 100 { t and even(cellpos(1)) }

rule: 0 100 { t and odd(cellpos(1)) }
```

Figure 78. File lifegame.rules.HtoS

Finally using the LTRANS result we construct the final model to be simulated in CD++.

```
[top]
components : life
[life]
type : cell
width : 20
height : 20
delay : transport
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1,-1) life(1,0) life(1,1)
initialvalue : 0
initialrowvalue : 1 00010001111000000000
initialrowvalue : 2 00110111100010111100
initialrowvalue : 3 00110000011110000010
initialrowvalue : 4 00101111000111100011
initialrowvalue : 10 01111000111100011110
initialrowvalue : 11 00010001111000000000
localtransition : life-rule
[life-rule]
rule: 1 100 { ( ( (0,0) = 1 ) and ( ( if((truecount - (if((-1,1) = 1,1,0)) -
(if((1,1) = 1,1,0))) < 0 , 0 , (truecount - (if((-1,1) = 1,1,0)) - (if((1,1)
= 1,1,0)))) = 3 ) or ( if((truecount - (if((-1,1) = 1,1,0)) - (if((1,1) =
1,1,0))) < 0 , 0 , (truecount - (if((-1,1) = 1,1,0)) - (if((1,1) = 1,1,0))))
= 4 ) ) ) and even(cellpos(1)) }
rule: 1 100 { ( ( (0,0) = 1 ) and ( ( if((truecount - (if((-1,-1) = 1,1,0))
- (if((1,-1) = 1,1,0))) < 0,0,(truecount - (if((-1,-1) = 1,1,0)) - (if((1,-
1) = 1,1,0)))) = 3 ) or ( if((truecount - (if((-1,-1) = 1,1,0)) - (if((1,-1)
= 1,1,0))) < 0,0,(truecount - (if((-1,-1) = 1,1,0)) - (if((1,-1) = 1,1,0))))
= 4 ) ) ) and odd(cellpos(1)) }
rule: 1 100 { ( ( (0,0) = 0 ) and ( if((truecount - (if((-1,1) = 1,1,0)) -
(if((1,1) = 1,1,0))) < 0 , 0 , (truecount - (if((-1,1) = 1,1,0)) - (if((1,1)
= 1,1,0)))) = 2 ) ) and even(cellpos(1)) }
rule : 1 100 { ( ( (0,0) = 0 ) and ( if((truecount - (if((-1,-1) = 1,1,0)) -
(if((1,-1) = 1,1,0))) < 0,0,(truecount - (if((-1,-1) = 1,1,0)) - (if((1,-1)
= 1,1,0)))) = 2 ) ) and odd(cellpos(1)) }
rule: 0 100 { t and even(cellpos(1)) }
rule: 0 100 { t and odd(cellpos(1)) }
```

Figure 79. Model file used as CD++ imput

7.9 Parlog

Parlog is a utility used to asses the parallelism of a running model. It uses the model log as input and counts the number of (*,t) messages received by each LP during a simulation cycle. After a simulation cycle has been completed, a list with the number of messages received by each LP will be printed.

Parlog reads the log from the standard input. *LogBuffer* should be used for correct results.
Usage:

```
PARLOG: An utility to determine the level of parallelism
usage: parlog -[?hmP]

where:
        ?       Show this message
        h       Show this message
 P       Partition file name
```

Figure 80. Parlog command line options

–h : Displays help.
-**?** :Displays help.
-**P**: Specifies the partition file name. This paramter is required because parlog needs to know how many LPs are being used.

Figure 9.7 shows the output generated by parlog with a model running in for machines.

```
Time/LP 0       1       2       3
00:00:00:000    629     626     626     626
00:00:10:000    5       0       2       3
00:00:11:000    12      3       12      14
00:00:12:000    31      7       32      35
00:00:13:000    60      13      62      66
00:00:14:000    99      21      102     107
00:00:15:000    148     31      152     158
00:00:16:000    207     43      212     219
00:00:17:000    276     57      282     290
00:00:18:000    351     73      358     367
00:00:19:000    428     91      436     446
00:00:20:000    509     131     495     486
00:00:21:000    543     192     531     522
00:00:22:000    575     254     563     554
00:00:23:000    603     317     591     582
00:00:24:000    625     376     614     606
00:00:25:000    627     450     625     626
```

Figure 81. Parlog output for a 4 machines partition.

7.10 Logbuffer

Logbuffer is a utility that buffers log messages received through the standard input, sorts them according to their time, and outputs them to the standard output. It should be used when running *drawlog* or *parlog* piped with the simulator.

To run logbuffer use,

logbuffer [-b]
-**b**n Sets the size of the buffer. The default size is 200.

Both *drawlog* and *parlog* require that, for correct results to be obtained, that log messages be processed in the order determined by their timestamps. When parallel simulation is run and the log is sent to the standard output, there is no guarantee that messages will be displayed in the same order that they were generated. Therefore, a sorted buffer is needed.

Logbuffer has an internal buffer of a used defined size, which is always kept sorted. When the simulation is started, this buffer is empty. Every new message that arrives is buffered, and no output is sent till the buffer is full. Once it is full, every new message that arrives causes a new message to be sent to the standard output. When the simulation finishes, all buffered messages are sent.

Figure 82. Logbuffer receives a message with timestamp 3 and then two messages with timestamp 2. Logbuffer sorts and sent in the correct order.

Logbuffer can only guarantee correct results for misplaced messages that occur within a distance smaller than the size of the buffer.

```
>./mpirun -np 4 ./pcd -mcalor.ma -Pcalor.par4 -t00:01:00:000 -l |
./logbuffer -b5000 | ./drawlog -mcalor.ma -csuperficie -w6-p2 > calor.drw

> ./mpirun -np 4 ./pcd -mcalor.ma -Pcalor.par4 -t00:01:00:000 -l |
./logbuffer -b5000 | ./parlog -Pcalor.par4 > calor.p
```

Figure 83. Running pcd with logbuffer.

7.11 Tools in CD++Builder

Most of the tools presented in this section are available to use in CD++Builder. Select the Cell-DEVS perspective from the Perspectives menu.

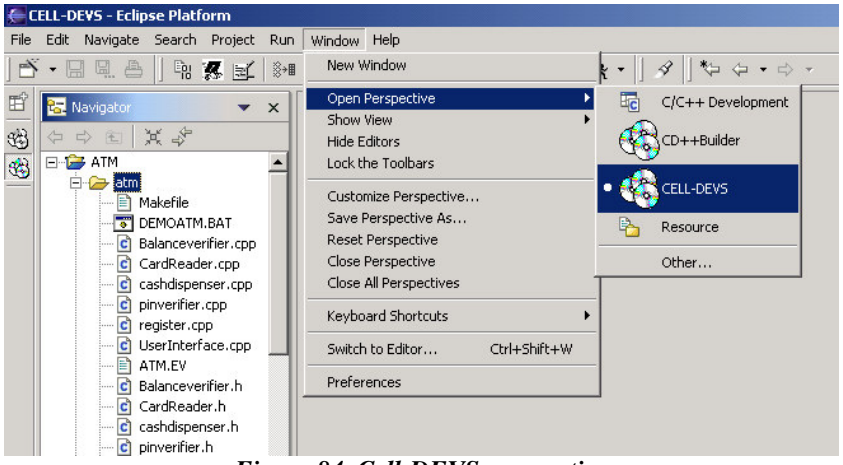


Figure 84. Cell-DEVS perspective

The toolbar contains buttons activating the different utilities presented in this section.

Figure 85. Cell-DEVS toolbar.

8 Appendix A – Installation and Technical notes

8.1 Installing CD++Builder

You will need three main programs to run this plugin in windows...

a) Java JRE - Java JRE (Java runtime environment) will enable you to run java based applications such as Eclipse.

b) Eclipse - Eclipse is an software development workbench. It provides a plugin based framework that makes it easier to create and utilize software tools. CD++ Builder is a plugin which will be incorporated in Eclipse as a plugin.

c) Cygwin - Cygwin is a Unix emulator for Windows. Since the CD++ toolkit was originally developed in the Linux/Unix platform it needs a Unix emulator to run some it's binaries (*not needed if running CD++ under Linux/Unix).*.

Installing Java

1) go to http://java.sun.com/j2se/1.4.2/download.html

2) Download the windows version of J2SE 1.4.2. You will see 2 types of Java SDK and JRE. Scroll down to the second set of downloads which is indicated by the heading "***J2SE V1.4.2_05 JRE*** *includes the jvm Technology*". Click on the link indicating Java SE 1.4.2 that is JRE. It should bring you to a page where you can right click and save the installer.

3) Run the installer. It will scan your computer to check if what you already have installed Java. Upon completion, you will be asked to accept a user's agreement. Clicking on "next" should bring you to a screen asking you to associate java to your web browsers (you can choose either depenting on your needs). By clicking next once again, the installation will begin.

Installing Eclipse (only if you have installed Java)

1) Go to http://www.eclipse.org/downloads/index.php.

2) Here you will get a list of Eclipse versions. Download one of the 2.X versions (CD++Builder is only compatible with Eclipse 2.X). Click on the link which is the latest version of Eclipse 2.X. If you need further information, check http://www.eclipse.org/downloads/index.php.

3) In the next page, select the Windows platform you use, and download the file (a zip file). It is highly recommended that you just unzip eclipse to the "C:\eclipse".

   If Eclipse or the project you are working on has spaces in the directory names then it may malfunction.

4) Go to the "C:\eclipse" folder, and double click on "eclipse.exe", which will load Eclipse for the first time and complete the installation. Include a shortcut in your desktop and your Start Menu, to easily access the application.

Installing Cygwin

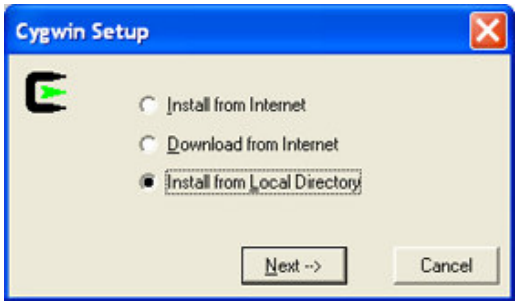
1) Download the Cygwin setup file from
   http://www.sce.carleton.ca/faculty/wainer/wbgraf/distrib/cygwin.zip

   and unzip it (Note: **use ONLY this version of cygwin; we cannot guarantee that the tools will**

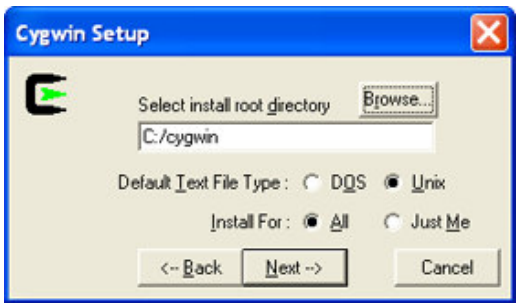
work with other versions). We suppose you will unzip it on "c:\cygwin". Go to that folder, and click on "setup.exe". The folowing window will appear:

For information about Cygwin is visit http://www.cygwin.com

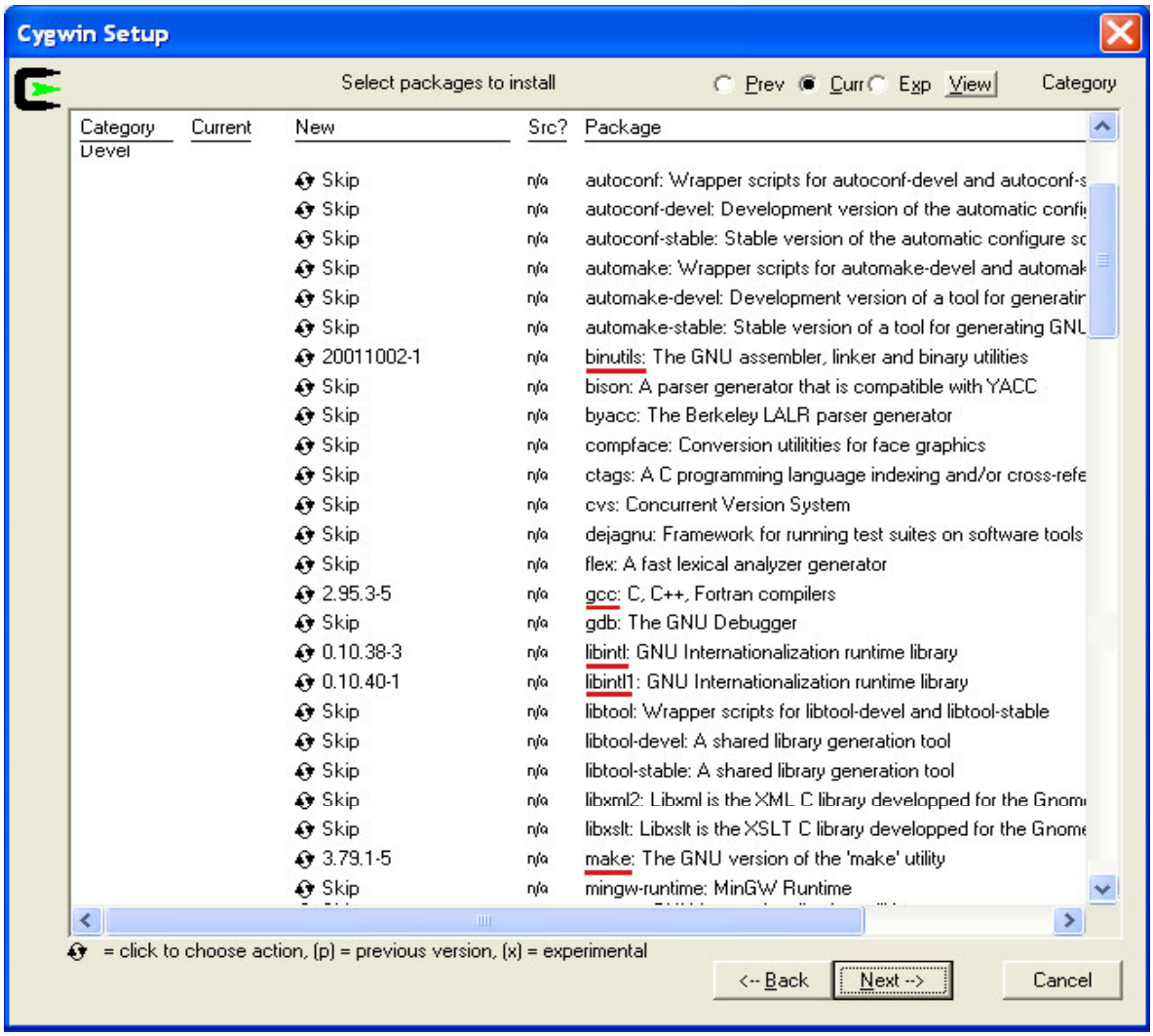
2) Click next and select install from *local directory*



2) If you are installing this for other accounts to use as well click on *install for* ***ALL*** and press next**.** The next window will show the location of where it is installing from. You should point to the folder where you unzipped the pack (in our example, "C:\cygwin").



4) Look for "Devel" under the *Category* column. Some of the tools under this category must be installed (underlined in red in the following figure). To install a particular tool click on the word "skip" under the *New* column, untill it changes to the tool's version to be installed.



5) Click on "next" to begin installation. When this is done, the installation tool will ask you to create shortcuts on the desktop and startup menu.

6) Set "c:\cygwin\bin" to your path. This can be done going to *Control panel->system*. Click on the "Advance" tab. Then depending on your version of windows there should be a section on environment variables. Here you can find the **Path** variable (usually under *system variables*) and add "c:\cygwin\bin;" to it.

   You may need to logout and re-login for the change occur.

7) Create a folder *"C:\tmp"* (do not confuse with c:\temp), and give read/write access to it.

8) Define cygwin's working directory in the drive where eclipse is installed on. To insure that this path has been set correctly, open a cygwin shell window (double-click on cygwin's icon on your desktop or startup menu). Assuming eclipse is installed on the C: drive, if you type in 'pwd', the output should be `"/cygdrive/c"` (i.e., the c: drive if eclipse was installed on this drive). If the path appears, for instance, as `"/cygdrive/m"` (or some other drive), you must modify the "home" environment variable.

   To set the path to the c: drive please go to the "C:\cygwin\etc" folder. Within this folder you will find a

file called *"profile"*. Open it with an editor. Below this line:

```
USER="`id -un`"
```

you must type the following (*there should not be any spaces in the first line*):

```
HOME="/cygdrive/c"

Export HOME
```

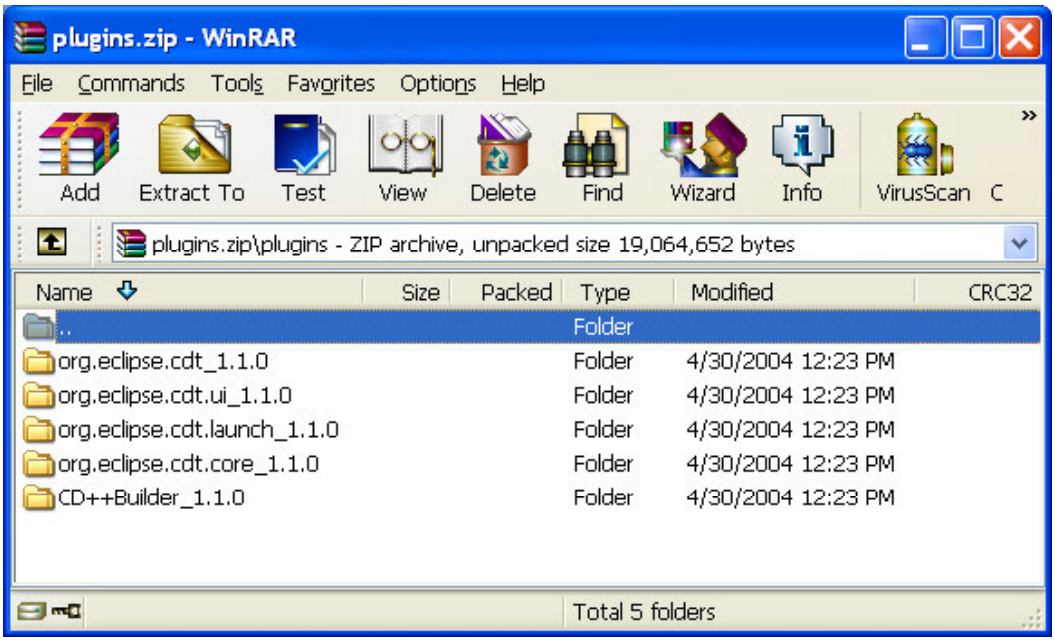
Then, save the file. Now, every time a cygwin shell is opened the working directory will be set to the C drive. To check if the change has been made close all cygwin windows and open a new one. Type "pwd", this time it should return `"/cygdrive/c"`

Installing the Plugin (only if you have installed the three items above)

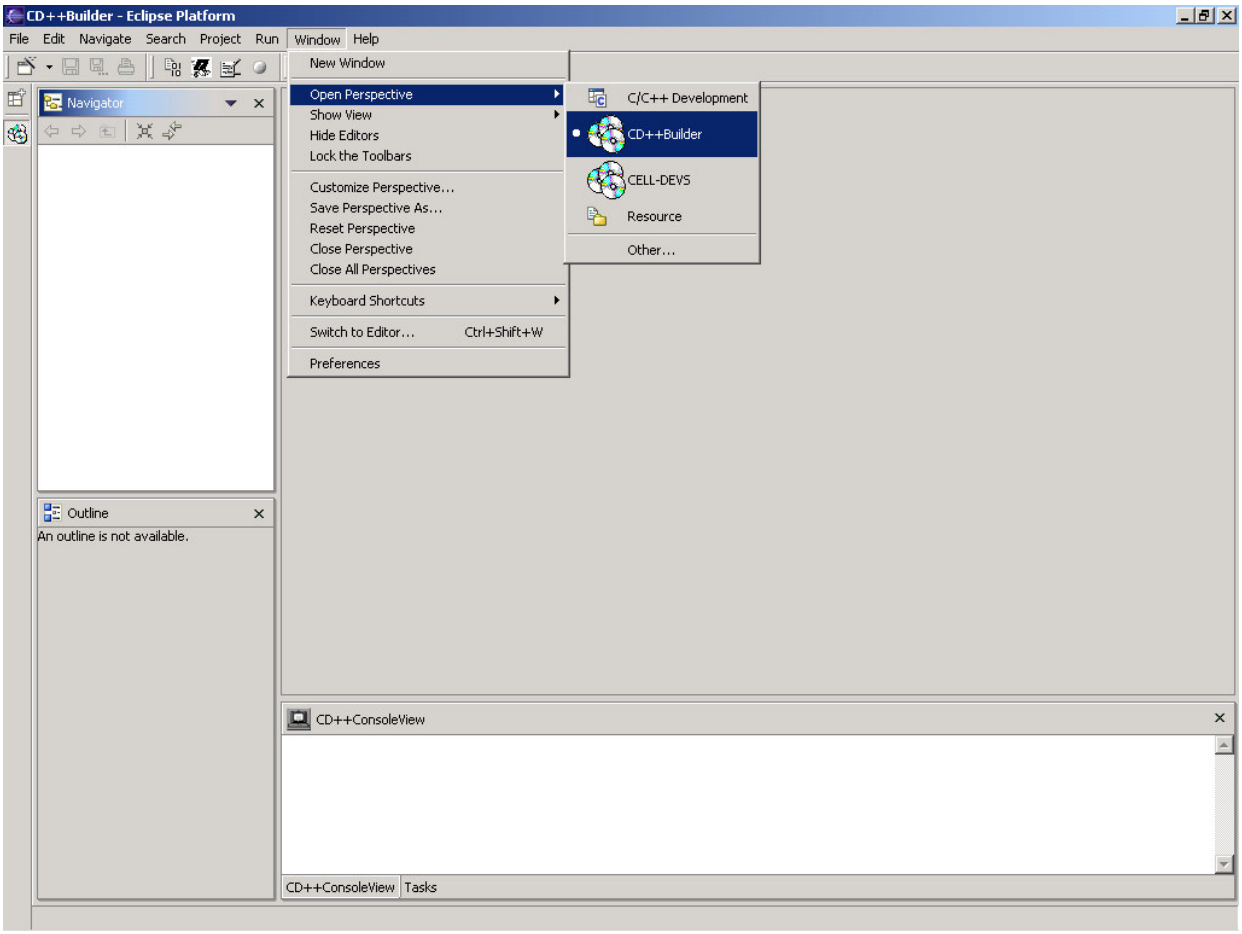
1. Download the Plugin from:

   http://www.sce.carleton.ca/faculty/wainer/wbgraf/distrib/CD++BuilderV1.2.zip

2. The plugin zip file contains a folder (called "plugins") with has five plugins. If you go to "c:\eclipse", you will find a folder called "plugins". Unzip the file and extract the plugions onto the Eclipse plugins folder ("c:\eclipse\plugins"), overwriting all of the existing files.



3. If you open eclipse, the new tools should be available. If you go to the *Windows->Open Perspective* menu, and choose "CD++Builder" as a perspective, you will a window like this one.



4. Download and run the examples explained in the first section of the User Manual. If you have problems, reinstall the tools.

Known Bugs

- Coupling Syntax Editor is case sensitive

- coupling editor provides coloring for only general keywords

- ***Folder Names with space (" ") "sometimes" corrupts your current project (if this happens, rename your folders)***

8.2 Command line installation

If you are planning to run the toolkit from the command line, the instructions are simple:

1. If you are using a Windows environment, refer to section 2.1, and download and install Cygwin

2. If you are using Linux, step 1 is not required.

3. Download CD++ toolkit from the toolkit website:

http://www.sce.carleton.ca/faculty/wainer/wbgraf/distrib/CD++BuilderV1.2.zip

CD++ toolkit files comes packed into a zip file (you will find different versions in the website). After having downloaded the toolkit, proceed to unzip the files to a directory. To execute a model, download it from http://www.sce.carleton.ca/faculty/wainer/wbgraf and unzip it in the same directory than the one where you included the toolkit. If it is a Cell-DEVS models, you just need to run the script (*.bat file) containing the execution commands for the model. If it is a DEVS model, you have to modify the Makefile (as explained in Section <> of the User Manual), and recompile the tool.

8.3 Installation for parallel simulation

Parallel CD++ was developed to run in UNIX and Windows NT environments that support the MPI library. It has been successfully tested in clusters of Linux machines running on Pentium processors. It supports both, parallel and standalone simulation.

The standalone version can also be compiled to run under Windows systems.

The CD++ distribution includes the following utilities:

- Drawlog: draws the evolution of a cellular model.
- Parlog: Counts the number of (*,t) messages received by each LP during each simulation cycle.
- Logbuffer: required by drawlog and parlog when parallel simulation is used. Sorts the log messages that are sent to standard output to ensure they are processed in the correct order.
- ToMap: creates the initial state cell map file from a .ma file.
- MakeRand: generates a random initial state cell map file.

The latest version of CD++ is distributed as a .tar.gz file and to install and compile CD++ the following utilities will be required:

- makedepend: current version released with X11R6 (part of X-windows software)
- GNU Make makefile utility (part of GNU software)
- g++: the GNU C++ compiler and accompanying libc, version 2.7.0 or later (part of GNU software)
- an implementation of MPI (e.g. MPICH) (for parallel simulation)
- GNU bison
- GNU flex

For parallel simulation, an implementation of MPI is required. If MPI is already installed in your system, find out if its includes and lib directories have been already added to the corresponding environment variables. Otherwise, take note of these directories because they will be required later on.

If MPI is not installed on your system, then it is recommended you install MPICH version 1.2.0, which can be downloaded from http://www.mcs.anl.gov/home/lusk/mpich/index.html. You can then install MPICH in a shared location (special permissions will be required) or in your home directory. Basic installation instructions will be provided.

The installation instructions here presented are based on personal experience installing in on Linux machines. If in doubt, please, check the mpich installation instructions found in **install.ps** in the /doc directory.

1. Uncompress the distribution files

```
gunzip –c mpich.tar.gz | tar xovf
```

2. Run

```
./configure
```

This script will try to set the optimum parameters for compilation on your system. If mpich will be installed in a shared location, then run (on your preferred location)

```
./configure –prefix= /usr/local/mpich-1.2.0.
```

3. Compile mpich by running

```
make >& make.log
```

This might take several minutes to an hour, depending on your system.

4. Edit the util/machines/machines.LINUX file and set the list of available machines in the cluster.

5. (Optional) Install mpich on a shared location

```
make install
```

If the default settings have not been changed, MPICH will use rsh to run the remote programs. For rsh to work properly, please check

1. Machine names are properly resolved, either using a DNS or the /etc/hosts file.
2. The inet services must be enabled in all the machines.
3. If you want to be able to run rsh without being prompted for a password, you will have to create a .rhosts file with the names of the machines in the cluster. The .rhost file must not have any group permissions enabled. Run chmod 600 .rhosts.
4. If the filesystem is not shared between all of the machines in the cluster, then a copy of CD++ and any model files will be required on each machine.

To install CD++, gunzip and untar the distribution file. On most Linux machines the command

```
gunzip –c pcd-3.x.x.tar.gz | tar xovf
```

will just do this.

The following directory structure will be created

```
CD++
    +----------   warped
                      +---------- TimeWarp
                      +---------- NoTime
                      +---------- Sequential
                      +---------- common
    +----------   models
                      +---------- net
                      +---------- airport
```

You must then edit Makefile.common and set the desired compilation options:

1. Set the source code location. If running parallel simulation, you will also need to indicate the location of the MPI include and lib files.

```
#CD++ Makefile.common

#============================================================================
#CD++ Directory Details
export MAINDIR=/USERDEFINEDPATH/CD++


#============================================================================
#MPI Directory Details
export MPIDIR=/USERDEFINEDPATH/mpich-1.2.0
export LDFLAGS +=-L$(MPIDIR)/lib/
export INCLUDES_CPP += -I$(MPIDIR)/include
#============================================================================
```

Figure 86. Makefile.common – Setting the source location

Specify whether parallel or stand alone simulation will be used. For stand alone simulation, the NoTime simulation kernel must be used. For parallel simulation, you can choose from the TimeWarp and NoTime kernel. If not sure, the NoTime kernel is recommended.

```
#If running parallel simulation, uncomment the following lines
export DEFINES_CPP += -DMPI
export LIBMPI = -lmpich
#============================================================================

#============================================================================
#WARPED CONFIGURATION
#============================================================================
#Warped Directory Details
#For the TimeWarp kernel uncomment the following
#export DEFINES_CPP += -DKERNEL_TIMEWARP
#export TWDIR=$(MAINDIR)/warped/TimeWarp/src
#export PLIBS += -lTW -lm -lnsl $(LIBMPI)
#export TWLIB = libTW.a

#For the NoTimeKernel, uncomment the following
export DEFINES_CPP += -DKERNEL_NOTIME
export TWDIR=$(MAINDIR)/warped/NoTime/src
export PLIBS += -lNoTime -lm -lnsl $(LIBMPI)
export TWLIB = libNoTime.a
#============================================================================
```

Figure 87. Makefile.common – Choosing the Warped kernel

2. Decide which atomic models will be included by removing the necessary comments.

```
##############################################################################################
#MODELS
#Let's define here which models we would like to include in our distribution
#Basic models
EXAMPLESOBJS=queue.o main.o generat.o cpu.o transduc.o distri.o com.o linpack.o
register.o

#Uncomment these lines to include the airport models
#DEFINES_CPP += -DDEVS_AIRPORT
#INCLUDES_CPP += -I./models/airport
#LDFLAGS += -L./models/airport
#LIBS += -lairport

#Uncomment these lines to include the net models
```

```
#DEFINES_CPP += -DDEVS_NET
#INCLUDES_CPP += -I./models/net
#LDFLAGS += -L./models/net
#LIBS += -lnet
################################################################################
```

Figure 88. Makefile.common – Model selection

After you have edited Makefile.common, you are ready to build CD++. To build CD++ and all the accompanying utilities, issue the following commands:

```
make depend
make
```

If you change any settings in Makefile.common you will need to rebuild CD++ again. To do this,

```
make clean
make
```

9 Appendix B - Local transition functions for Cell-DEVS models.

Local transition functions for cellular models are defined as groups in the .ma file. They are not tied to a particular model, so they can be used for more than one cellular model at the same time. A local transition is made of a set of rules of the form:

```
rule : result  delay { condition }
```

A rule is composed of three elements: a *condition*, a *delay* and a *result*. To calculate the new value for a cell's state, the simulator takes each rule (in the order in that they were defined) and evaluates the condition clause. If the condition evaluates to true, then the result and delay clause are evaluated. The result will be the new cell state and will be sent as an output after the obtained delay. Whether the previous sate values will be still sent as outputs or not will depend on the delay type of the cells. Inertial delay cells will preempt any scheduled outputs. On the other hand, transport delay cells will keep them.

Rules whose condition clause evaluates to false are skipped. If all the rules are evaluated without one having a true condition, then the simulation will be aborted. If there is more than one rule with a condition that evaluates to true, the first one will be the one that determines the new cell's state. If the delay clause of a cell evaluates to undefined, then the simulation will be automatically cancelled.

9.1 A grammar for writing the rules

The BNF for the grammar used for the rules is shown in Figure 11.1. Words written in bold lowercase represent terminals symbols, while those written in uppercase represent non terminals.

```
RULELIST     = RULE  |  RULE RULELIST
RULE         = RESULT RESULT { BOOLEXP }
RESULT       = CONSTANT |  { REALEXP }
BOOLEXP      = BOOL | ( BOOLEXP ) |  REALRELEXP |  not BOOLEXP
               |  BOOLEXP OP_BOOL BOOLEXP
OP_BOOL      = and  |  or  |  xor  |  imp  |  eqv
REALRELEXP   = REALEXP OP_REL REALEXP | COND_REAL_FUNC(REALEXP)
REALEXP      = IDREF |  ( REALEXP ) | REALEXP OPER REALEXP
IDREF        = CELLREF |  CONSTANT  |  FUNCTION |  portValue(PORTNAME)
                 |  send(PORTNAME, REALEXP) |  cellPos(REALEXP)
CONSTANT     = INT |  REAL |  CONSTFUNC |  ?
FUNCTION     = UNARY_FUNC(REALEXP) |  WITHOUT_PARAM_FUNC
                 |  BINARY_FUNC(REALEXP, REALEXP)
                 |  if(BOOLEXP, REALEXP, REALEXP)
                 |  ifu(BOOLEXP, REALEXP, REALEXP, REALEXP)
CELLREF      = (INT, INT REST_TUPLE
REST_TUPLE   = , INT REST_TUPLE |  )
BOOL         = t  |  f  |  ?
OP_REL       = !=  |  =  |  >  |  <  |  >=  |  <=
OPER         = +  |  -  |  *  |  /
INT          = [SIGN] DIGIT {DIGIT}
REAL         = INT  |  [SIGN] {DIGIT}.DIGIT {DIGIT}
SIGN         = +  |  -
DIGIT        = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
PORTNAME     = thisPort | STRING
STRING       = LETTER {LETTER}
LETTER       = a | b | c |...| z | A | B | C |...| Z
CONSTFUNC    = pi | e | inf | grav | accel | light | planck | avogadro |
               faraday | rydberg | euler_gamma | bohr_radius | boltzmann |
               bohr_magneton | golden | catalan | amu | electron_charge |
               ideal_gas | stefan_boltzmann | proton_mass | electron_mass |
```

```
                neutron_mass | pem
WITHOUT_PARAM_FUNC = truecount | falsecount | undefcount | time | random |
                     randomSign
UNARY_FUNC    = abs | acos | acosh | asin | asinh | atan | atanh | cos |  sec
              | sech | exp | cosh | fact | fractional | ln | log | round | cotan
              | cosec | cosech | sign | sin | sinh | statecount | sqrt | tan | tanh
              | trunc | truncUpper | poisson | exponential | randInt | chi | asec
              | acotan | asech | acosech | nextPrime | radToDeg | degToRad
              | nth_prime | acotanh | CtoF | CtoK | KtoC | KtoF | FtoC |  FtoK

BINARY_FUNC   = comb | logn | max | min | power | remainder | root | beta | gamma
              | lcm | gcd | normal | f | uniform | binomial | rectToPolar_r
              | rectToPolar_angle | polarToRect_x | hip | polarToRect_y

COND_REAL_FUNC = even | odd | isInt | isPrime | isUndefined
```

Figure 11.1: Grammar used for the definition of a cell's local transition

Basically, a rule is made of three expressions: a result expression, a delay expression and a boolean expression. The result expression should evaluate to any real value. The delay expression should also evaluate to any real value that will be truncated to the smallest integer.

9.2 Precedence Order and Associativity of Operators

The precedence order indicates which operation will be solved first. For example if we have:

C + B * A

where * and + are the sum and multiplication operations for real numbers, and *A*, *B* and *C* are real constants, then since * has higher precedence than +, *B* * *A* will be evaluated first. The sum will be evaluate in a second step. The result will be equivalent to solve *C* + (*B* * *A*).

The associativity indicates which of two operations of same precedence will be evaluated first. Operators are either left associative or right associative. The logical operators *AND* and *OR* are left associative, so the in the expression

```
C and B or D
```

will be solved as (*C* **and** *B*) **or** *D*

Clauses that are not associative cannot be combined simultaneously without another operator of different precedence.

The table of precedence and associativities for the rule specification language follows:

Order	Code	Associativity
Lower Precendence	[illegible]	[illegible]
	( )	

Figure 89. Precedence Order and Associativity used in CD++

9.3 Functions and Constants allowed by the language

9.3.1 Boolean Values

Boolean values in CD++ use trivalent logic.

The trivalent logic use the values **T** or **1** to represent to the value *TRUE*, **F** or **0** to represent the *FALSE*, and **?**

to represent to the *UNDEFINED*.

9.3.1.1 Boolean Operators

AND	**T**	**F**	**?**
T	T	F	?
F	F	F	F

Operator AND

The behavior of the operator ***AND*** is defined with the following table of truth:

Figure 90. operator AND truthtable

Operator *OR*

OR	**T**	**F**	**?**
T	T	T	T
F	T	F	?

The behavior of the operator OR is defined with the following table of truth:

Figure 91. Operator OR truthtable

Operator *NOT*

NOT	
T	F
F	T

The behavior of the operator NOT is defined with the following table of truth:

Figure 92. Behavior of the boolean operator NOT

Operator *XOR*

The behavior of the operator XOR is defined with the following table of truth:

XOR	**T**	**F**	**?**
T	F	T	?
F	T	F	?

Figure 93. Operator XOR truthtable

Operator *IMP*

IMP	**T**	**F**	**?**
T	T	F	?
F	T	T	T

IMP represents the logic implication, and its behavior is defined with the following table of truth:

Figure 94. Operator IMP truthtable

EQV	**T**	**F**	**?**
T	T	F	F
F	F	T	F

Operator *EQV*

EQV represents the equivalence between trivalent logic values, and its behavior is defined with the following table of truth:

Figure 95. Operator EQV truthtable

9.3.2 Functions and Operations on Real Numbers

9.3.2.1 Relational Operators

The relational operators work on real numbers[1] and return a boolean value pertaining to the previously defined trivalent logic. The language used by *CD++* allows the use of the operators ==, !=, >, <, >=, <= whose behavior is described next.

As opposed to the traditional definition of these operators, the introduction of an undefined value makes the definition of a total order impossible because the value **?** is not comparable with any existing real number.

Operator =

The operator **=** is used to test for equality of two real numbers.

[1] From here, when referring to the term "Real Number" a value in the set $\boldsymbol{R} \cup \{ \textbf{?} \}$ will be meant.

=	?	Real Number
?	T	?
Real Number	?	= of real number

Figure 96. ***Behavior of the Relational Operator =***

Operator !=

The operator **!=** is used to test if two real numbers are not equal. Its behavior is defined as follows:

!=	?	Real Number
?	F	?
Real Number	?	≠ of real number

Figure 97. Behavior of the Relational Operator !=

Operator >

The operator **>** is used to test if a real number is greater than another real number. Its behavior is defined as

>	?	Real Number
?	F	?
Real Number	?	> of real number

follows:

Figure 98. Behavior of the Relational Operator >

Operator <

The operator **<** is used to test if a real number is less then another real number. Its behavior is defined as

<	?	Real Number
?	F	?
Real Number	?	< of real number

follows:

Figure 99. Behavior of the Relational Operator <

Operator <=

<=	?	Real Number
?	T	?
Real Number	?	≤ of real number

The operator **<=** is used to test if a real number is less or equal to another real number. Its behavior is defined as follows:

Figure 100. Behavior of the Relational Operator <=

Operator >=

>=	?	Real Number
?	T	?
Real Number	?	≥ of real number

The operator **>=** is used to test if a real number is greater or equal to another real number. Its behavior is defined as follows:

Figure 101. Behavior of the Relational Operator >=

9.3.2.2 Arithmetic Operators

The traditional arithmetic operators are available. If any of the operands is undefined, then the result of the operation will be undefined. This is also valid for functions. If any of a function arguments is undefined, the result of evaluating the function will also be undefined.

The available operators are:

op1 + op2	returns the sum of the operators.
op1 – op2	returns the difference between the operators.
op1 / op2	returns the value of the op1 divided by op2.
op1 * op2	returns the product of the operators

Division by zero will result to the undefined value.

Figure 102. Arithmetic Operators

9.3.2.3 Functions on Real Numbers

Functions to Verify Properties of Real Numbers

The functions in this section allow to check for special properties of real numbers, such as parity, primality, etc.

Function Even

Signature: **even** : *Real* → *Bool*
Description: Returns *True* if the value is integer and even. If the value is undefined returns *Undefined*. In any other case it returns *False*.
Examples: even(?) = F
even(3.14) = F
even(3) = F
even(2) = T

Function Odd

Signature: **odd** : *Real* → *Bool*
Description: Returns *True* if the value is integer and odd. If the value is undefined returns Undefined. In any other case it returns *False*.
Examples: odd(?) = F
odd(3.14) = F
odd(3) = T
odd(2) = F

Function isInt

Signature: **isInt** : *Real* → *Bool*
Description: Returns *True* if the value is integer and not undefined. Any other case returns *False*.
Examples: isInt(?) = F
isInt(3.14) = F
isInt(3) = T

Function isPrime

Signature: **isPrime** : *Real* → *Bool*
Description: Returns *True* if the value is a prime number. Any other case returns *False*.
Examples: isPrime(?) = F
isPrime(3.14) = F
isPrime(6) = F
isPrime(5) = T

Function isUndefined

Signature: **isUndefined** : *Real* → *Bool*
Description: Returns *True* if the value is undefined, else returns *False*.
Examples: isUndefined(?) = T
isUndefined(4) = F

Mathematical Functions

This section describes commonly used mathematical functions.

Trigonometric Functions

Function tan

Signature: **tan** : *Real a* → *Real*
Description: Returns the tangent of *a* measured in radians.

For the values near to π/2 radians, returns the constant *INF*.
If *a* is undefined then return undefined.

Examples: tan(*PI* / 2) = *INF*
tan(?) = ?
tan(*PI*) = 0

Function sin

Signature: **sin** : *Real a* → *Real*
Description: Returns the sine of *a* measured in radians.
If *a* has the value **?** then returns **?**.

Function cos

Signature: **cos** : *Real a* → *Real*
Description: Returns the cosine of *a* measured in radians.
If *a* has the value**?** the returns**?**.

Function sec

Signature: **sec** : *Real a* → *Real*
Description: Returns the secant of *a* measured in radians.
If *a* has the value**?** then returns**?**.
If the angle is of the form π/2 + *x*.π, with *x* an integer number, then returns the constant *INF*.

Function cotan

Signature: **cotan** : *Real a* → *Real*
Description: Calculates the cotangent of *a*.
If *a* has the value**?** Then returns **?**.
If *a* is zero or multiple of π, then returns *INF*.

Function cosec

Signature: **cosec** : *Real a* → *Real*
Description: Calculates the cosecant of *a*.
If *a* has the value **?**, then returns**?**.
If *a* is zero or multiple of π, then returns *INF*.

Function atan

Signature: **atan** : *Real a* → *Real*
Description: Returns the arc tangent of *a* measured in radians, which is defined as the value *b* such tan(*b*) = *a*.
If *a* has the value**?** Then returns**?**.

Function asin

Signature: **asin** : *Real a* → *Real*
Description: Returns the arc sine of *a* measured in radians, which is defined as the value *b* such sin(*b*) = *a*.
If *a* has the value**?** or if $a \notin$ [-1, 1], then returns **?**.

Function acos

Signature: **acos** : *Real a* → *Real*
Description: Returns the arc cosine of *a* measured in radians, which is defined as the value *b* such cos(*b*) = *a*.
If *a* has the value**?** or if $a \notin$ [-1, 1], then returns **?**.

Function asec

Signature: **asec** : *Real a* → *Real*
Description: Returns the arc secant of *a* measured in radians, which is defined as the value *b* such sec(*b*) = *a*.
If *a* is undefined (**?**) or if |*a*| < 1, then returns **?**.

Function acotan

Signature: **acotan** : *Real a* → *Real*
Description: Returns the arc cotangent of *a* measured in radians, which is defined as the value *b* such cotan(*b*) = *a*.
If *a* is undefined (**?**), then returns **?**.

Function sinh

Signature: **sinh** : *Real a* → *Real*
Description: Returns the hyperbolic sine of *a* measured in radians.
If *a* has the value **?**, then returns **?**.

Function cosh

Signature: **cosh** : *Real a* → *Real*
Description: Returns the hyperbolic cosine of *a* measured in radians, which is defined as cosh(*x*) = $(e^{x} + e^{-x}) / 2$.
If *a* has the value **?**, then returns **?**.

Function tanh

Signature: **tanh** : *Real a* → *Real*
Description: Returns the hyperbolic tangent of *a* measured in radians, which is defined as *sinh(a) / cosh(a)*.
If *a* has the value**?**, then returns **?**.

Function sech

Signature: **sech** : *Real a* → *Real*
Description: Returns the hyperbolic secant of *a* measured in radians, which is defined as
1 / cosh(*a*)
If *a* has the value **?**, then returns **?**.

Function cosech

Signature: **cosech** : *Real a* → *Real*
Description: Returns the hyperbolic cosecant of *a* measured in radians.
If *a* has the value **?**, then returns **?**.

Function atanh

Signature: **atanh** : *Real a* → *Real*
Description: Returns the hyperbolic arc tangent of *a* measured in radians, which is defined as the value *b* such tanh(*b*) = *a*.
If *a* has the value **?**, or if its absolute value is greater than 1 (i.e., $a \notin [-1, 1]$), then returns **?**.

Function asinh

Signature: **asinh** : *Real a* → *Real*
Description: Returns the hyperbolic arc sine of *a* measured in radians, which is defined as the value *b* such sinh(*b*) = *a*.
If *a* has the value **?**, then returns **?**.

Function acosh

Signature: **acosh** : *Real a* → *Real*

Description: Returns the hyperbolic arc cosine of *a* measured in radians, which is defined as the value *b* such cosh(*b*) = *a*.
If *a* has the value **?** or is less than 1, then returns **?**.

Function asech

Signature: **asech** : *Real a* → *Real*
Description: Returns the hyperbolic arc secant of *a* measured in radians, which is defined as the value *b* such sech(*b*) = *a*.
If *a* is undefined, then return **?**. If it is zero, then returns the constant *INF*.

Function acosech

Signature: **acosech** : *Real a* → *Real*
Description: Returns the hyperbolic arc cosec of *a* measured in radians, which is defined as the value *b* such cosech(*b*) = *a*.
If *a* is undefined, then returns **?**. If it is zero, then returns the constant *INF*.

Function acotanh

Signature: **acotanh** : *Real a* → *Real*
Description: Returns the hyperbolic arc cotangent of *a* measured in radians, which is defined as the value *b* such cotanh(*b*) = *a*.
If *a* is undefined, then returns **?**. If is 1 then returns the constant *INF*.

Function hip

Signature: **hip** : *Real c1* x *Real c2* → *Real*
Description: Calculates the hypotenuse of the triangle composed by the side *c1* and *c2*. If *c1* or *c2* are undefined or negatives, then returns **?**.

Functions to calculate Roots, Powers and Logarithms.

Function sqrt

Signature: **sqrt** : *Real a* → *Real*
Description: Returns the square root of *a*.
If *a* is undefined or negative, then returns **?**.
Examples: sqrt(4) = 2
sqrt(2) = 1.41421
sqrt(0) = 0
sqrt(-2) = ?
sqrt(?) = ?
Note: sqrt(*x*) is equivalent to **root**(*x*, 2) $\forall x$

Function exp

Signature: **exp** : *Real x* → *Real*
Description: Returns the value of e^x.
If *x* is undefined, then return **?**.
Examples: exp(?) = ?
exp(–2) = 0.135335
exp(1) = 2.71828
exp(0) = 1

Function ln

Signature: **ln** : *Real a* → *Real*
Description: Returns the natural logarithm of *a*.
If *a* is undefined or is less or equal than zero, then returns **?**.
Examples: ln(–2) = ?
ln(0) = ?
ln(1) = 0

ln(?) = ?

Note: ln(x) is equivalent to **logn**(x, e) $\forall x$

Function log

Signature: **log** : *Real a* → *Real*

Description: Returns the logarithm in base 10 of a.
If a is undefined or less or equal to zero, then returns **?**.

Examples: log(3) = 0.477121
log(–2) = ?
log(?) = ?
log(0) = ?

Note: log(x) is equivalent to **logn**(x, 10) $\forall x$

Function logn

Signature: **logn** : *Real a* x *Real n* → *Real*

Description: Returns the logarithm in base n of the value a.
If a or n are undefined, negatives or zero, then returns **?**.

Notes: logn(x, e) is equivalent to **ln**(x) $\forall x$
logn(x, 10) is equivalent to **log**(x) $\forall x$

Function power

Signature: **power** : *Real a* x *Real b* → *Real*

Description: Returns a^b.
If a or b are undefined or b is not an integer, then returns **?**.

Function root

Signature: **root** : *Real a* x *Real n* → *Real*

Description: Returns the n–root of a.
If a or n are undefined, then returns **?**. Also, returns this value if a is negative or n is zero.

Examples: root(27, 3) = 3
root(8, 2) = 3
root(4, 2) = 2
root(2, ?) = ?
root(3, 0.5) = 9
root(–2, 2) = ?
root(0, 4) = 0
root(1, 3) = 1
root(4, 3) = 1.5874

Note: root(x, 2) is equivalent to **sqrt**(x) $\forall x$

Functions to calculate GCD, LCM and the Rest of the Numeric Division

Function LCM

Signature: **lcm** : *Real a* x *Real b* → *Real*

Description: Returns the Less Common Multiplier between a and b.
If a or b are undefined or non–integers, then returns **?**.
The value returned is always integer.

Function GCD

Signature: **gcd** : *Real a* x *Real b* → *Real*

Description: Calculates the Greater Common Divisor betweeen a and b.
If a or b are undefined or non–integers, then returns **?**.
The value returned is always integer.

Function remainder

Signature: **remainder** : *Real a* x *Real b* → *Real*

Description: Calculates the remaindert of the división between *a* and *b*. The returned value is: *a* – *n* * *b*, where *n* is the quotient *a*/*b* rounded as an integer.
If *a* or *b* are undefined, then returns **?**.

Examples: remainder(12, 3) = 0
remainder(14, 3) = 2
remainder(4, 2) = 0
remainder(0, y) = 0 ∀ y ≠ **?**
remainder(x, 0) = x ∀ x
remainder(1.25, 0.3) = 0.05
remainder(1.25, 0.25) = 0
remainder(?, 3) = ?
remainder(5, ?) = ?

Functions to Convert Real Values to Integers Values

This section presents functions available to convert real values to integers using the rounding and truncation techniques as detailed.

Function round

Signature: **round** : *Real a* → *Real*

Description: Rounds the value *a* to the nearest integer.
If *a* is undefined **?**, then returns **?**.

Examples: round(4) = 4
round(?) = ?
round(4.1) = 4
round(4.7) = 5
round(–3.6) = –4

Function trunc

Signature: **trunc**: *Real x* → *Real*

Description: Returns the greater integer number less or equal than *x*.
If *x* is undefined, then returns **?**.

Examples: trunc(4) = 4
trunc(?) = ?
trunc(4.1) = 4
trunc(4.7) = 4

Function truncUpper

Signature: **truncUpper**: *Real x* → *Real*

Description: Returns the smallest integer number greater or equal than *x*.
If *x* is undefined, then returns **?**.

Examples: truncUpper(4) = 4
truncUpper(?) = ?
truncUpper(4.1) = 5
truncUpper(4.7) = 5

Function fractional

Signature: **fractional** : *Real a* → *Real*

Description: Returns the fractional part of *a*, including the sign.
If *a* is undefined then returns **?**.

Examples: fractional(4.15) = 0.15
fractional(?) = ?
fractional(-3.6) = -0.6

Functions to manipulate the Sign of numerical values

Function abs

Signature: **abs** : *Real a* → *Real*

Description: Returns the absolute value of *a*.
If *a* is undefined then returns **?**.

Examples: abs(4.15) = 4.15
abs(?) = ?
abs(-3.6) = 3.6
abs(0) = 0

Function sign

Signature: **sign** : *Real a* → *Real*

Description: Returns the sign of *a* in the following form:
If $a > 0$ then returns 1.
If $a < 0$ then returns –1.
If $a = 0$ then returns 0.
If *a* = **?** then returns **?**.

Function randomSign

See the section of Probability Functions.

Functions to manipulate Prime numbers

This functions are used to test for primality. Although they are available, they are quite complex and can require a lot of time to solve.

Function isPrime

See the section of Functions to Verify Properties of Real Numbers.

Function nextPrime

Signature: **nextPrime** : Real *r* → *Real*

Description: Returns the next prime number greater than *r*.
If *r* is undefined then returns **?**.
If an overflow occur when calculating the next prime number, the constant *INF* is returned.

Function nth_Prime

Signature: **nth_Prime** : Real *n* → *Real*

Description: Returns the n^{th} prime number, considering as the first prime number the value 2.
If *n* is undefined or non–integer then returns **?**.
If an overflow occur when calculating the next prime number, the constant *INF* is returned.

Functions to calculate Minimum and Maximums

Function min

Signature: **min** : *Real a* x *Real b* → *Real*

Description: Return the minimum between *a* and *b*.
If *a* or *b* are undefined then returns **?**.

Function max

Signature: **max** : *Real a* x *Real b* → *Real*

Description: Returns the maximum between *a* and *b*.
If *a* or *b* are undefined then returns **?**.

Conditional Functions

The functions described in this section return a real value that depends on the evaluation of a specified logical

condition.

Function if

Signature: **if** : Bool *c* x Real *t* x Real *f* → *Real*

Description: If the condition *c* is evaluated to *TRUE*, then returns the evaluation of *t*, else returns the evaluation of *f*.
The values of *t* and *f* can even come from the evaluation of any expression that returns a real value, including another *if* sentence.

Examples: If you wish to return the value 1.5 when the natural logarithm of the cell (0, 0) is zero or negative, or 2 in another case. In this case, it will be written:

if (ln((0, 0)) = 0 or (0, 0) < 0, 1.5, 2)

If you wants to return the value of the cells (1, 1) + (2, 2) when the cell (0, 0) isn't zero; or the square root of (3, 3) in another case, it will be written:

if ((0, 0) != 0, (1, 1) + (2, 2), sqrt(3, 3))

It can also be used for the treatment of a numeric overflow. For example, if the factorial of the cell (0, 1) produces an overflows, then return –1, else return the obtained result. In this case, it will be written:

if (fact((0, 1)) = *INF*, –1, fact((0, 1)))

Function ifu

Signature: **ifu** : Bool *c* x Real *t* x Real *f* x Real *u* → *Real*

Description: If the condition *c* is evaluated to *TRUE*, then returns the evaluation of *t*. If it evaluates to *FALSE*, returns the evaluation of *f*. Else (i.e. is undefined), returns the evaluation of *u*.

Examples: If you wish to return the value of the cell (0, 0) if its value is distinct than zero and undefined, 1 if the value of the cell is 0, and π if the cell has the undefined value. In this case, it will be invoked:

ifu((0, 0) != 0, (0, 0), 1, *PI*)

Probabilistic Functions

Function randomSign

Signature: **randomSign** : → *Real*

Description: Randomly returns a numerical value that represents a sign (+1 or –1), with equal probability for both values.

Function random

Signature: **random** : → *Real*

Description: Returns a random real value pertaining to the interval (0, 1), with uniform distribution.

Note: random is equivalent to *uniform(0,1)*.

Function chi

Signature: **chi** : *Real df* → *Real*

Description: Returns a random real number with Chi–Squared distribution with *df* degree of freedom.
If *df* is undefined, negative or zero, then returns **?**.

Function beta

Signature: **beta** : *Real a* x *Real b* → *Real*

Description: Returns a random real number with Beta distribution, with parameters *a* and *b*.
If *a* or *b* are undefined or less than 10^{-37}, then returns **?**.

Function exponential

Signature: **exponential** : *Real av* → *Real*

Description: Returns a random real number with Exponential distribution, with average *av*.
If *av* is undefined or negative, then returns **?**.

Function f

Signature: **f** : *Real dfn* x *Real dfd* → *Real*

Description: Returns a random real number with F distribution, with *dfn* degree of freedom for de numerator, and *dfd* for the denominator.
If *dfn* or *dfd* are undefined, negatives or zero, then return **?**.

Function gamma

Signature: **gamma** : *Real a* x *Real b* → *Real*

Description: Returns a random real number with Gamma distribution with parameters (*a*, *b*).
If *a* or *b* are undefined, negatives or zero, then returns **?**.

Function normal

Signature: **normal** : *Real* μ x *Real* σ → *Real*

Description: Returns a random real number with Normal distribution (μ, σ), where μ is the average, and σ is the standard error.
If μ or σ are undefined, or σ is negative, returns **?**.

Function uniform

Signature: **uniform** : *Real a* x *Real b* → *Real*

Description: Returns a random real number with uniform distribution, pertaining to the interval (*a*, *b*).
If *a* or *b* are undefined, or *a* > *b*, then returns **?**.

Note: *uniform(0, 1)* is equivalent to the function *random*.

Function binomial

Signature: **binomial** : *Real n* x *Real p* → *Real*

Description: Returns a random number with Binomial distribution, where *n* is the number of attempts, and *p* is the success probability of an event.
If *n* or *p* are undefined, *n* is not integer or negative, or *p* not pertain to the interval [0, 1], then return **?**.
The returned number is always an integer.

Function poisson

Signature: **poisson** : *Real n* → *Real*

Description: Return a random number with Poisson distribution, with average *n*.
If *n* is undefined or negative, then returns **?**.
The returned number is always an integer.

Function randInt

Signature: **randInt** : *Real n* → *Real*

Description: Returns an integer random number contained in the interval [0, *n*], with uniform distribution.
If *n* is undefined, then returns **?**.

Note: *randInt*(*n*) is equivalent to *round*(*uniform*(*0*, *n*))

Functions to calculate Factorials and Combinatorial

Function fact

Signature: **fact** : *Real a* → *Real*

Description: Returns the factorial of *a*.
If *a* is undefined, negative or non–integer, then return **?**.
If an overflow occur when calculating the next prime number, the constant *INF* is returned.

Examples: fact(3) = 6
fact(0) = 1

fact(5) = 120
fact(13) = 1.93205e+09
fact(43) = *INF*

Function comb

Signature: **comb** : *Real a* x *Real b* → *Real*

Description: Returns the combinatory $\binom{a}{b}$

If *a* or *b* are undefined, negatives or zero, or non–integers, then returns **?**. This value is also returned if a < b.

If an overflow occur when calculating the next prime number, the constant *INF* is returned.

9.3.2.4 Functions for the Cells and his Neighborhood

This section details the functions that allow to count the quantity of cells belonging to the neighborhood whose state has certain value, as also the function *cellPos* that allows to project an element of the tupla that references to the cell.

Function stateCount

Signature: **stateCount** : *Real a* → *Real*

Description: Returns the quantity of neighbors of the cell whose state is equal to *a*.

Function trueCount

Signature: **trueCount** : → *Real*

Description: Returns the quantity of neighbors of the cell whose state is 1.

This function is equivalent to *stateCount*(1) and it is not removed from the language to offer compatibility with *CD++*.

Function falseCount

Signature: **falseCount** : → *Real*

Description: Returns the quantity of neighbors of the cell whose state is 0.

This function is equivalent to *stateCount*(1) and it is not removed from the language to offer compatibility with *CD++*.

Function undefCount

Signature: **undefCount** : → *Real*

Description: Returns the quantity of neighbors of the cell whose state is undefined (**?**).

This function is equivalent to *stateCount*(1) and it is not removed from the language to offer compatibility with *CD++*.

Function cellPos

Signature: **cellPos** : *Real i* → *Real*

Description: Returns the i^{th} position inside the tupla that references to the cell. That is to say, given the cell $(x_0,x_1,...,x_n)$, then *cellPos*(i) = x_i.

If the value of *i* is not integer, then it will be automatically truncated.

If $i \notin [0, n+1)$, where n is the dimension of the model, it will produce an error that will abort the simulation.

The value returned always will be an integer.

Examples: Given the cell (4, 3, 10, 2):
cellPos(0) = 4
cellPos(3.99) = cellPos(3) = 2

cellPos(1.5) = cellPos(1) = 3
cellPos(–1) y cellPos(4) will generate an error.

9.3.2.5 Functions to Get the Simulation Time

Function Time

Signature: **time** : → *Real*
Description: Returns the time of the simulation at the moment in that the rule this being evaluated, expressed in milliseconds.

9.3.2.6 Functions to Convert Values between different units

Functions to Convert Degrees to Radians

Function radToDeg

Signature: **radToDeg** : Real $r \rightarrow$ *Real*
Description: Converts the value r from radians to degrees.
If r is undefined then returns **?**.

Function degToRad

Signature: **degToRad** : Real $r \rightarrow$ *Real*
Description: Converts the value r from degrees to radians.
If r is undefined then returns **?**.

Functions to Convert Rectangular to Polar Coordinates

Function rectToPolar_r

Signature: **rectToPolar_r** : Real x x Real $y \rightarrow$ *Real*
Description: Converts the Cartesian coodinate (x, y) to the polar form (r, θ), and returns r.
If x or y are undefined then return **?**.

Function rectToPolar_angle

Signature: **rectToPolar_angle** : Real x x Real $y \rightarrow$ *Real*
Description: Converts the Cartesian coordinate (x, y) to the polar form (r, θ), and returns θ.
If x or y are undefined then return **?**.

Function polarToRect_x

Signature: **polarToRect_x** : Real r x Real $\theta \rightarrow$ *Real*
Description: Converts the polar coordinate (r, θ) to the Cartesian form (x, y), and returns x.
If r or θ are undefined, or r is negative, then returns **?**.

Function polarToRect_y

Signature: **polarToRect_y** : Real r x Real $\theta \rightarrow$ *Real*
Description: Converts the polar coordinate (r, θ) to the Cartesian form (x, y), and returns y.
If r or θ are undefined, or r is negative, then returns **?**.

Functions to Covert Temperatures between different units

Function CtoF

Signature: **CtoF** : Real → *Real*
Description: Converts a value expressed in Centigrade degrees to Fahrenheit degrees.
If the value is undefined then returns **?**.

Function CtoK

Signature: **CtoK** : Real → *Real*
Description: Converts a value expressed in Centigrade degrees to Kelvin degrees.

If the value is undefined then returns **?**.

Function KtoC

Signature: **KtoC** : Real → *Real*
Description: Converts a value expressed in Kelvin degrees to Centigrade degrees.

If the value is undefined then returns **?**.

Function KtoF

Signature: **KtoF** : Real → *Real*
Description: Converts a value expressed in Kelvin degrees to Fahrenheit degrees.

If the value is undefined then returns **?**.

Function FtoC

Signature: **FtoC** : Real → *Real*
Description: Converts a value expressed in Fahrenheit degrees to Centigrade degrees.

If the value is undefined then returns **?**.

Function FtoK

Signature: **FtoK** : Real → *Real*
Description: Converts a value expressed in Fahrenheit degrees to Kelvin degrees.

If the value is undefined then returns **?**.

Functions to manipulate the Values on the Input and Output Ports

Function portValue

Signature: **portValue** : String *p* → *Real*
Description: Returns the last value arrived through the input port *p* of the cell of the cell being evaluated. This function will only be available for *PortInTransition* rules (see section12.3) . Other uses will generate an error.

If no message has arrived through port *p* before *portValue* is evaluated, an undefined value (**?**) will be returned. Otherwise, the last value received through the port will be returned.

When the string "***thisPort***" is used as the port name, the value received through the port associated with the current *PortInTransition* will be returned. For example:

The following model has two different *PortInTransitions*

```
PortInTransition: portA@cell(0,0)     functionA
PortInTransition: portB@cell(1,1)     functionB

[functionA]
rule: 10     100    { portValue(portA) > 10 }
rule: 0      100    { t }

[functionB]
rule: 10     100    { portValue(portB) > 10 }
rule: 0      100    { t }
```

Figure 103. Example of use of the function portValue

If we wanted to avoid repeating the same transition twice, we could either give the two ports the same name or use *thisPort* as shown next:

```
PortInTransition: portA@cell(0,0)     functionA
PortInTransition: portB@cell(1,1)     functionA

[functionA]
rule: 10     100    { portValue(thisPort) > 10 }
rule: 0      100    { t }
```

Figure 104. Example of use of the function portValue with thisPort

Section 12.3 shows an example where the *portInTransition* clause is used.

Function send

Signature: **send** : String *p* x Real *x* → *0*
Description: Sends the value *x* through the output port *p*.

If the output port *p* has not been defined, an error will be raised and the simulation will be aborted. This function is usually used to send values to other DEVS models.

send always returns 0. This makes it possible to include the function ***send*** in the *result* section of a rule without modifying the actual results.

{ (0,0) + **send**(port1, 15 * log(10)) } 100 { (0,0) > 10 }

Note: **Send** is a function of the language that can be used in any expression, as for example, in the definition of a *condition*. However, this is not recommended because for every condition that is evaluated that includes the function send, a value will be sent. Instead, send should be used in the expression for the *delay* or the *value* of the cell.

9.3.3 Predefined Constants

The following constants frequently used in the domains of the physics and the chemistry are available.

Constant Pi

Returns 3.14159265358979323846, which represent the value of π, the relation between the circumference and the radius of the circle.

Constant e

Returns 2.7182818284590452353, which represent the value of the base of the natural logarithms.

Constant INF

This constant represents to the infinite value, although in fact it returns the maximum value valid for a *Double* number (in processors Intel 80x86, this number is 1.79769×10^{308}).
Note that if, for example, we make $x + INF - INF$, where x is any real value, we will get 0 as a result, because the operator + is associative to left, for that will be solved:

$$(x + INF) - INF = INF - INF = 0.$$

Note: When being generated a numeric overflows taken place by any operation, it is returned *INF* or *–INF*. For example: power(12333333, 78134577) = *INF*.

Constant electron_mass

Returns the mass of an electron, which is $9.1093898 \times 10^{-28}$ grams.

Constant proton_mass

Returns the mass of a proton, which is 1.6726231 x 10^{-24} grams.

Constant neutron_mass

Returns the mass of a neutron, which is 1.6749286 x 10^{-24} grams.

Constant Catalan

Returns the Catalan's constant, which is defined as $\sum_{k=0}^{\infty}(-1)^k.(2^k+1)^{-2}$, that is approximately 0.9159655941772.

Constant Rydberg

Returns the Rydberg's constant, which is defined as 10.973.731,534 / m.

Constant grav

Returns the gravitational constant, defined as 6,67259 x 10^{-11} m^3 / (kg . s^2)

Constant bohr_radius

Returns the Bohr's radius, defined as 0,529177249 x 10^{-10} m.

Constant bohr_magneton

Returns the value of the Bohr's magneton, defined as 9,2740154 x 10^{-24} joule / tesla.

Constant Boltzmann

Returns the value of the Boltzmann's constant, defined as 1,380658 x 10^{-23} joule / °K.

Constant accel

Returns the standard acceleration constant, defined as 9,80665 m / sec^2.

Constant light

Returns the constant that represents the light speed in a vacuum, defined as 299.792.458 m / sec.

Constant electron_charge

Returns the value of the electron charge, defined as 1,60217733 x 10^{-19} coulomb.

Constant Planck

Returns the Planck's constant, defined as 6,6260755 x 10^{-34} joule . sec.

Constant Avogadro

Returns the Avogadro's number, defined as 6,0221367 x 10^{23} mols.

Constant amu

Returns the Atomic Mass Unit, defined as 1,6605402 x 10^{-27} kg.

Constant pem

Returns the ratio between the proton and electron mass, defined as 1836,152701.

Constant ideal_gas

Returns the constant of the ideal gas, defined as 22,41410 litres / mols.

Constant Faraday

Returns the Faraday's constant, defined as 96485,309 coulomb / mol.

Constant Stefan_boltzmann

Returns the Stefan-Boltzmann's constant, defined as 5,67051 x 10^{-8} Watt / (m^2 . °K^4)

Constant golden

Returns the *Golden Ratio*, defined as $\frac{1+\sqrt{5}}{2}$.

Constant euler_gamma

Returns the value of the Euler's Gamma, defined as 0.5772156649015.

9.4 Techniques to Avoid the Repetition of Rules

This section describes different techniques that allow to avoid repeating rules. This helps to make models more readable.

9.4.1 Clause *Else*

When the clause **portInTransition** is used (see section12.3), it is possible to use the clause **else** to give an alternative rule in case that none of the rules evaluates to true.

Figure 11.19 shows a short example where the *Else* clause is used. The default local transition for the cells in this model is *default_rule.* In addition, cell (13,13) defines a special funcion to be used when an external event arrives through port *In*. If none of the conditions for the rules that make this functions is satisfied, then the else clause sets *default_rule* as the function to be evaluated.

```
[demoModel]
type: cell
...
link: in  in@demoModel(13,13)
localTransition: default_rule
portInTransition: in@demoModel(13,13)    another_rule

[default_rule]
rule: ...
...
rule: ...

[another_rule]
rule: 1 1000 { portValue(thisPort) = 0 }
...
else: default_rule
```

Figure 105. Example of the Else clause

The *Else* clause can point to any valid transition function. Care must be taken to avoid circular references, as in the example shown next.

```
[another_rule1]
rule: 1    1000 { portValue(thisPort) = 0 }
rule: 1.5  1000 { (0,0) = 5 }
rule: 3    1500 { (1,1) + (0,0) >= 1 }
else: another_rule2

[another_rule2]
rule: 1 1000 { (0,0) + portValue(thisPort) > 3 }
```

```
else: another_rule1
```

Figure 106. A circular reference produced by a bad use of the clause Else

CD++ will detect the special case shown in Figure 11.21, where the *else* clause references the same function being defined.

```
[another_rule]
rule: ...
rule: ...
else: another_rule
```

Figure 107. Example of a circular reference detected by the simulator

9.4.2 11.4.2 Preprocessor – Using Macros

CD++ has a preprocessor that will expand macros. If macros are not used, the preprocessor can be disabled using the command line argument **–b** to speed up model parsing.

Macros are usually defined in separate files that are included in the main .ma file be means of the preprocessor **#include** directive, which is of the form

#include(fileName)

where *fileName* is the name of the file that contains the definition of the macros. This file should be in the same directory where the main .ma file is.

More than one #include directive is allowed in the main .ma file, but no included files can have themselves the #include directive.

To define a macro, the directives **#BeginMacro** and **#EndMacro** are used.

A macro definition has the form:

```
#BeginMacro(macroName)
...
...definition of the macro...
...
#EndMacro
```

Figure 108. Definition of a macro

Macros can contain any valid text in any number of lines. The only restriction that applies is that they can not be used in the same file they are defined.

To expand a macro, the **#Macro** directive should be used in the place were the macro shoudl be expanded. A #macro directive is of the form

#Macro(*macroName*)

An included file can contain any number of macro definitions. Any text in these files that is outside the macro definitions is ignored. If a required macro is not found, an error will be reported.

An #*include* directive can be placed at any line of the .ma file, as long as the macros therein defined are used after the #*include*.

A macro can not make use of another macro.

Within a .ma file, the preprocessor allows comments. Comments begin with a **%** . All text between the **%** and the end of the line is ignored.

```
% Here begins the rules
Rule : 1  100  { truecount > 1 or (0,0,1) = 2 }    % Validate the existence
                                                   % of another individual.
```

Figure 109. A .ma file with comments

Section 12.5 shows a model where macros are used.

For special considerations regarding files created by the preprocessor, please see *Appendix C*.

10 Appendix C – Examples

10.1 The " Life Game"

The *Life Game* was presented in Scientific American by the well known mathematician Martin Gardner. In this game, living cells will live or die. The rules for life evolution are as follows:

- An active cell will remain in this state if it has two or three active neighbors.
- An inactive cell will pass to active state if it has two active neighbors exactly.
- In any other case, the cell will die

The implementation of this model in *CD++* is as follows:

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1,-1) life(1,0) life(1,1)
initialvalue : 0
initialrowvalue :  1  00010001111000000000
initialrowvalue :  2  00110111100010111100
initialrowvalue :  3  00110000011110000010
initialrowvalue :  4  00101111000111100011
initialrowvalue : 10  01111000111100011110
initialrowvalue : 11  00010001111000000000
localtransition : life-rule

[life-rule]
rule : 1 100 { (0,0) = 1 and (truecount = 3 or truecount = 4) }
rule : 1 100 { (0,0) = 0 and truecount = 2 }
rule : 0 100 { t }
```

Figure 110. Implementation of the Game of Life

10.2 A bouncing object

The following is the specification of a model that represents an object in movement that bounces against the borders of a room. This example is ideal to illustrate the use of a non toroidal cellular automata, where the cells of the border have different behavior to the rest of the cells.

For the representation of the problem, 5 different values are used for the states of each cell, these values are:

0 = represents an empty cell.
1 = represents the object moving toward the south east.
2 = represents the object moving toward the north east.
3 = represents the object moving toward the south west.
4 = represents the object moving toward the north west.

The specification of the model is:

```
[top]
components : rebound

[rebound]
type : cell
width : 20
height : 15
delay : transport
defaultDelayTime  : 100
border : nowrapped
neighbors : rebound(-1,-1)              rebound(-1,1)
neighbors :                rebound(0,0)
neighbors : rebound(1,-1)               rebound(1,1)
initialvalue : 0
initialrowvalue : 13     00000000000000000010
localtransition : move-rule
zone : cornerUL-rule { (0,0) }
zone : cornerUR-rule { (0,19) }
zone : cornerDL-rule { (14,0) }
zone : cornerDR-rule { (14,19) }
zone : top-rule { (0,1)..(0,18) }
zone : bottom-rule { (14,1)..(14,18) }
zone : left-rule { (1,0)..(13,0) }
zone : right-rule { (1,19)..(13,19) }

[move-rule]
rule : 1 100 { (-1,-1) = 1 }
rule : 2 100 { (1,-1) = 2 }
rule : 3 100 { (-1,1) = 3 }
rule : 4 100 { (1,1) = 4 }
rule : 0 100 { t }

[top-rule]
rule : 3 100 { (1,1) = 4 }
rule : 1 100 { (1,-1) = 2 }
rule : 0 100 { t }

[bottom-rule]
rule : 4 100 { (-1,1) = 3 }
rule : 2 100 { (-1,-1) = 1 }
rule : 0 100 { t }

[left-rule]
rule : 1 100 { (-1,1) = 3 }
rule : 2 100 { (1,1) = 4 }
rule : 0 100 { t }

[right-rule]
rule : 3 100 { (-1,-1) = 1 }
rule : 4 100 { (1,-1) = 2 }
rule : 0 100 { t }
```

```
[cornerUL-rule]
rule : 1 100 { (1,1) = 4 }
rule : 0 100 { t }

[cornerUR-rule]
rule : 3 100 { (1,-1) = 2 }
rule : 0 100 { t }

[cornerDL-rule]
rule : 2 100 { (-1,1) = 3 }
rule : 0 100 { t }

[cornerUR-rule]
rule : 4 100 { (-1,-1) = 1 }
rule : 0 100 { t }
```

Figure 111. Implementation of the Rebound of an Object

10.3 Classification of raw materials

The aim of this example is to show the use of special behavior that can be given to a cell when an external event arrives through an input port. We have a model that represents the packing and classification of certain raw material that contains 30% of carbon approximately. The model is made of a machine that loads 100 grams fractions of that substance in a carrying band. One a fraction reaches the end of the band, it is processed by a packager that takes these fractions until a kilogram is obtained. Then, the packed substance is classified. If each packet contains 30 ± 1 % of carbon, it is classified as of first quality; otherwise, it will be of second quality.

The model uses the atomic model *Generator* that generates values (in this case always the value 1) each *x* seconds (where *x* has and Exponential distribution with average 3). These values are passed to the carry band, represented by a cellular mode. At the end of the band, another cellular model makes the packaging and selection.

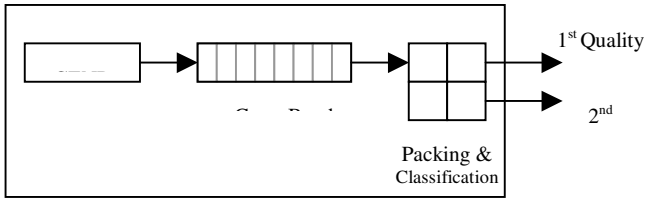


Figure 112. Coupling structure for the Classification of Substances

The following is the specification of the model:

```
[top]
components : genSubstances@Generator queue packing
out : outFirstQuality outSecondQuality
link : out@genSunstances in@queue
link : out@queue in@packing
link : out1@packing  outFirstQuality
link : out2@packing  outSecondQuality
```

```
[genSubstances]
distribution : exponential
mean : 3
initial : 1
increment : 0

[queue]
type : cell
width : 6
height : 1
delay : transport
defaultDelayTime  : 1
border : nowrapped
neighbors : queue(0,-1) queue(0,0) queue(0,1)
initialvalue : 0
in : in
out : out
link : in in@queue(0,0)
link : out@queue(0,5) out
localtransition : queue-rule
portInTransition : in@queue(0,0)  setSubstance

[queue-rule]
rule : 0           1 { (0,0) != 0 and (0,1) = 0 }
rule : { (0,-1) }  1 { (0,0) = 0 and (0,-1) != 0 and not isUndefined((0,-1))
}
rule : 0        3000 { (0,0) != 0 and isUndefined((0,1)) }
rule : { (0,0) }   1 { t }

[setSubstance]
rule : { 30 + normal(0,2) } 1000 { t }

[packing]
type : cell
width : 2
height : 2
delay : transport
defaultDelayTime  : 1000
border : nowrapped
neighbors : packing(-1,-1) packing(-1,0) packing(-1,1)
neighbors : packing(0,-1)  packing(0,0)  packing(0,1)
neighbors : packing(1,-1)  packing(1,0)  packing(1,1)
in : in
out : out1 out2
initialvalue : 0
initialrowvalue :  0      00
initialrowvalue :  1      00
link : in in@ packing(0,0)
link : in in@ packing(1,0)
link : out@ packing(0,1) out1
link : out@ packing(1,1) out2
localtransition : packing-rule
portInTransition : in@packing(0,0) add-rule
portInTransition : in@packing(1,0) incQuantity-rule

[packing-rule]
rule : 0   1000 { isUndefined((1,0)) and isUndefined((0,-1)) and (0,0) = 10
}
rule : 0   1000 { isUndefined((-1,0)) and isUndefined((0,-1)) and (1,0) = 10
}
```

```
rule : { (0,-1) / (1,-1) }  1000 { isUndefined((-1,0)) and
isUndefined((0,1))
                        and (1,-1) = 10 and abs( (0,-1) / (1,-1) - 30 ) <= 1
}
rule : { (-1,-1) / (0,-1) } 1000 { isUndefined((1,0)) and isUndefined((0,1))
                        and (0,-1) = 10 and abs( (-1,-1) / (0,-1) - 30 ) > 1
}
rule : { (0,0) } 1000 { t }

[add-rule]
rule : { portValue(thisPort) + (0,0) } 1000 { portValue(thisPort) != 0 }
rule : { (0,0) }                       1000 { t }

[incQuantity-rule]
rule : { 1 + (0,0) } 1000 { portValue(thisPort) != 0 }
rule : { (0,0) }     1000 { t }
```

Figure 113. **Implementation of the Model to Classify Substances**

The cellular model **queue** that represents the carry band makes use of the **portInTranstition** clause. As it was mentioned earlier, this clause is used to set the rule that will be evaluated when an external event is received by the cell through the specified port. This clause is then used again in the definition of the model *Packing* set the behavior of the cells upon the reception of a raw material from the carry band.

10.4 Life Game – 3D

The next example is an adaptation of the *Game of the Life* to a three dimensional space.

Figure 12.5 shows the model definition and Figure 12.6 lists the contents of file "3d-life.val" that contains the initial values for the cell.

```
[top]
components : 3d-life

[3d-life]
type : cell
dim : (7,7,3)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : 3d-life(-1,-1,-1) 3d-life(-1,0,-1) 3d-life(-1,1,-1)
neighbors : 3d-life(0,-1,-1)  3d-life(0,0,-1)  3d-life(0,1,-1)
neighbors : 3d-life(1,-1,-1)  3d-life(1,0,-1)  3d-life(1,1,-1)
neighbors : 3d-life(-1,-1,0)  3d-life(-1,0,0)  3d-life(-1,1,0)
neighbors : 3d-life(0,-1,0)   3d-life(0,0,0)   3d-life(0,1,0)
neighbors : 3d-life(1,-1,0)   3d-life(1,0,0)   3d-life(1,1,0)
neighbors : 3d-life(-1,-1,1)  3d-life(-1,0,1)  3d-life(-1,1,1)
neighbors : 3d-life(0,-1,1)   3d-life(0,0,1)   3d-life(0,1,1)
neighbors : 3d-life(1,-1,1)   3d-life(1,0,1)   3d-life(1,1,1)
initialvalue : 0
initialCellsValue : 3d-life.val
localtransition : 3d-life-rule

[3d-life-rule]
rule : 1 100 { (0,0,0) = 1 and (truecount = 8 or truecount = 10) }
rule : 1 100 { (0,0,0) = 0 and truecount >= 10 }
rule : 0 100 { t }
```

Figure 114. Implementation of the Game of Life – 3D

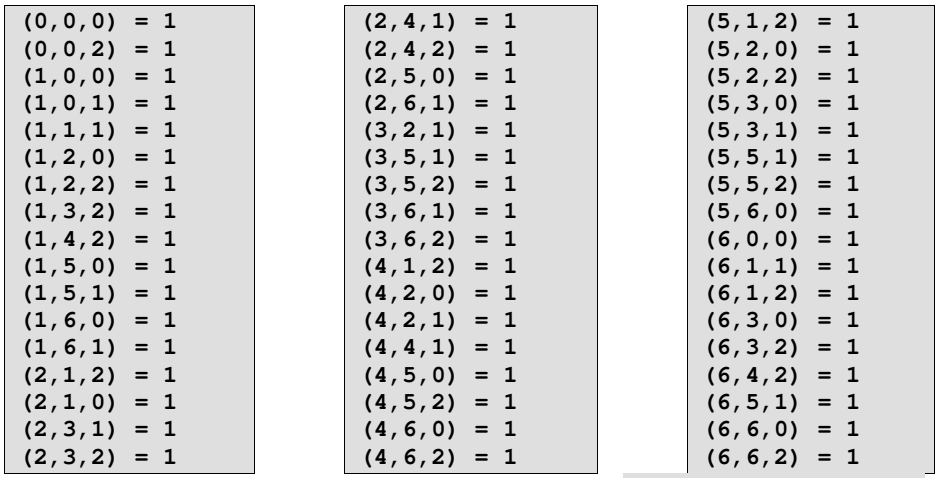


Figure 115. Initial values for the cells of the Game of Life – 3D

10.5 Use of Macros

The following example shows how macros can be used to write a new version of the *Game of the Life* for a 4 dimensional space. Macros can be defined in external files that are included in the main .ma file. More than one macro definition is may be included per file, but no macro can make use of an existing macro. A macro is defined between the *#BeginMacro* and a *#EndMacro* directives. All other text is ignored. The next figures show the contents of the four files that are used to completely define the new model.

```
#include(life.inc)
#include(life-1.inc)

[top]
components : life

[life]
type : cell
dim : (2,10,3,4)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors :                life(-1,-1,0,0) life(-1,0,0,0) life(-1,1,0,0)
neighbors : life(0,-8,0,0) life(0,-1,0,0)  life(0,0,0,0)  life(0,1,0,0)
neighbors :                life(1,-1,0,0)  life(1,0,0,0)  life(1,1,0,0)
initialvalue : 0
initialCellsValue :  life.val
localtransition : life-rule

[life-rule]
% Comment: Here starts the definition of rules
rule : 1                100 { #macro(Heat) or #macro(Rain) }
rule : 0                100 { (0,0,0,0) = ? OR (0,0,0,0) = 2 }
#macro(rule1)   % Another comment: A macro is invoked
rule : 1                100 { (0,0,0,0) = (1,0,0,0) AND (0,0,0,0) > 1 }
#macro(rule2)
```

Figure 116. Implementation of the Game of Life with 4 dimensions and using macros

```
(0,0,0,0) = ?
(1,0,0,0) = 25
(0,0,1,0) = 21
(0,1,2,2) = 28
(1, 4, 1,2) = 17
(1, 3, 2,1) = 15.44
```

Figure 117. File life.val that contains the initial values for the Game of Life in 4D

```
This is a comment: The macro Rule3 assigns the value 0 if the cell's value
is 3, and 4 if the cell's value is negative.

#BeginMacro(rule3)
rule : 0 100 { (0,0,0,0) = 3 }
rule : 4 100 { (0,0,0,0) < 0 }
#EndMacro


#BeginMacro(rule1)
rule : 0 100 { (0,0,0,0) + (1,0,0,0) + (1,1,0,0) + (0,-8,0,0) = 11 }
#EndMacro

#BeginMacro(Heat)
(0,0,0,0) > 30
#EndMacro
```

Figure 118. File life.inc that contains some macros used in the Game of Life 4D

```
#BeginMacro(Rule2)
rule : 0 100 { (0,0,0,0) = 7 }
rule : { (0,0,0,0) + 2 } 100 { t }
#EndMacro

#BeginMacro(Rain)
(0,-8,0,0) > 25
#EndMacro
```

Figure 119. File life–1.inc that contains the remaining macros for the Game of Life 4D

11 Appendix D– The preprocessor and temporary files.

When the preprocessor is used to resolve macros (by default the preprocessor is enabled), it will create a temporary file for the model with all macros expanded and all the comments erased. This temporary file is then passed to the simulator for its interpretation. If the use of the preprocessor with the parameter **-b** is disabled and macros are used, the model will not be processed correctly.

The name of the temporary file is the value returned by the instruction *tmpnam* of the *GCC*. The directory where the temporary files are located will be selected according to the following criteria:

1. When CD++ is compiled, the name of directory defined by *P_tmpdir* <*stdio.h*> will be used, unless this is the root directory.

   In *Linux* this variable usually has the value: "**/TMP**", while in the version of the *GCC* 2.8.1 for *Windows*–32 bits, this variable references to the root directory of the disk unit that is in use.

2. If *P_tmpdir* points to the root directory, then the name defined by the environment variable **TEMP** will be used.

3. If no **TEMP** variable is defined, then the value of the environment variable **TMP** will be used.

4. Finally, if the **TMP** is neither defined, the current directory will be used.

12 Appendix E – User Manual Of the CD++Modeler Tool (FIRST DRAFT)

This Graphical User Interface (GUI) can be used to create atomic and coupled models for the CD++ tool. The basic functions of the GUI include creating atomic model and coupled model, exporting the models to different formats and animating the simulations. The GUI also includes a simple text editor to modify Cell-DEVS models directly and the RUN options to execute de simulation with the cd++ and generate the draw information with the drawlog. The GUI is coded in Java, which enable it to run on various environments. The following sections explore the functions of the GUI with examples.

The manual describes in detail how to use the GUI for DEVS model input and the visualization of the results of the Cell-DEVS models, Atomic-DEVS models and Coupled-DEVS models. This GUI is developed with JDK 1.4.1, so it is platform portable and can be run on various environment such as Eclipse, JBuilder with JDK1.4.1.

To run the GUI, decompress the cdModeler.zip file under Windows environments. All the source code, class files and some example files will be in the directory cdModeler.

To use this tool, you should

Install a JDK 1.4.1 or above.

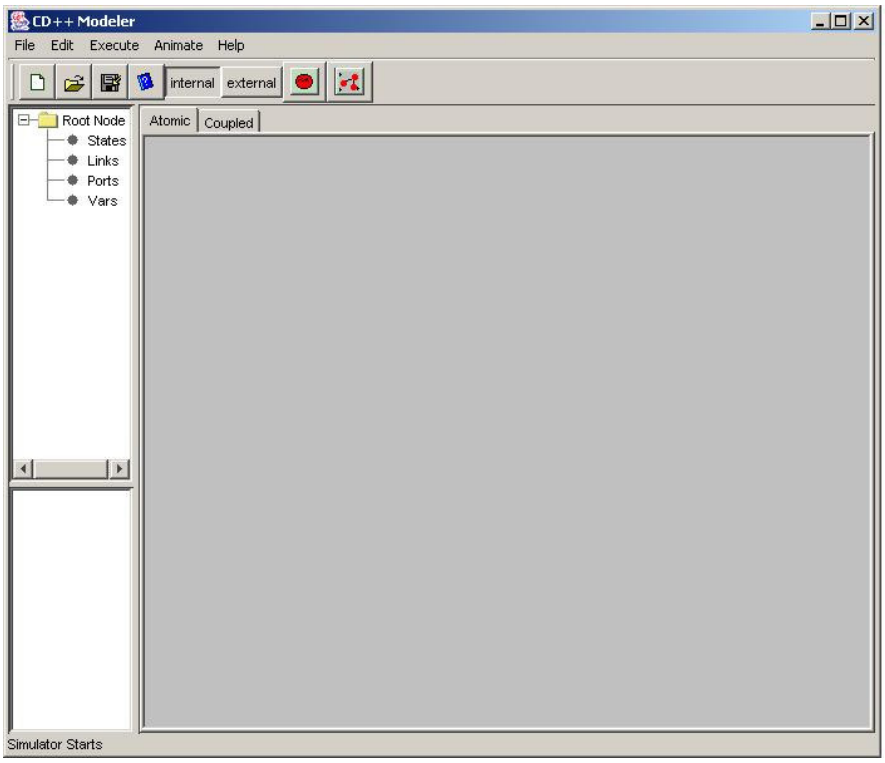
Set upo your JAVA_HOME enviroment variable pointing to your JDK directory.

Modify your PATH enviroment and add %JAVA_HOME%\bin;

Note : to modify the enviroment variables go to Control Panel -> System -> Enviroment Variables

In cdModeler directory, double click the bat file run.bat to start the program.

The program will start and show a Dialog like the following

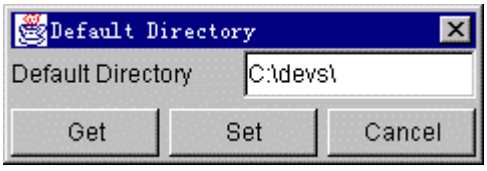


12.1 Open File Directory

Every time an Open Dialog is displayed, the application remembers the last acceded directory. This is very useful. Also, the application has an input dialog to set the current directory

Usually you should set the directory, which is used most frequently, as the default directory. Such as, you can use the directory where the result files reside as the default directory, so you can directly get to this directory every time you want to select a result file for visualization.

To set the home directory, select ”Home Directory” item in “File” menu of this GUI, a dialog like the following is shown.



To change this directory, you can use one of the following two methods:

- Input the new directory name in the text field or and press “Set” button.
- Click “Get” button, a file load dialog will open, then you get into the directory, which you want to set as the default directory.Click “Load” button, this directory will appear in the text field, and press “Set” button.

Then the GUI will use this new directory as the default directory, and its name will be saved to a file. Every time the GUI start up, it will read this file and set the default directory in this file.

12.2 Create DEVS Models

The design space is used for the user to define the model. Before creating the model, the user should select proper design space for the corresponding atomic model or coupled model by click on the tabs, which is indicated as “design space selector” in Fig 1. The information space helps the user to know the details of the model created on the design space.

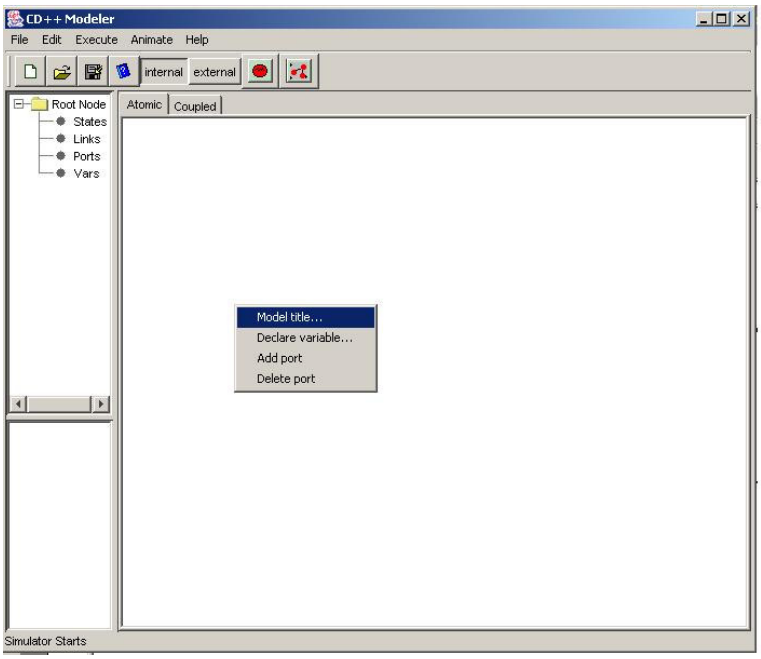
12.2.1 Create Atomic Model

The following section describes the basic steps that are needed to create an atomic model.

Select “Atomic” Tab from the Interface.

Select “File | New” from the main menu of the Interface.

Click right button of the mouse on the design space and a pop up menu will be shown. Fig 3 shows the result.



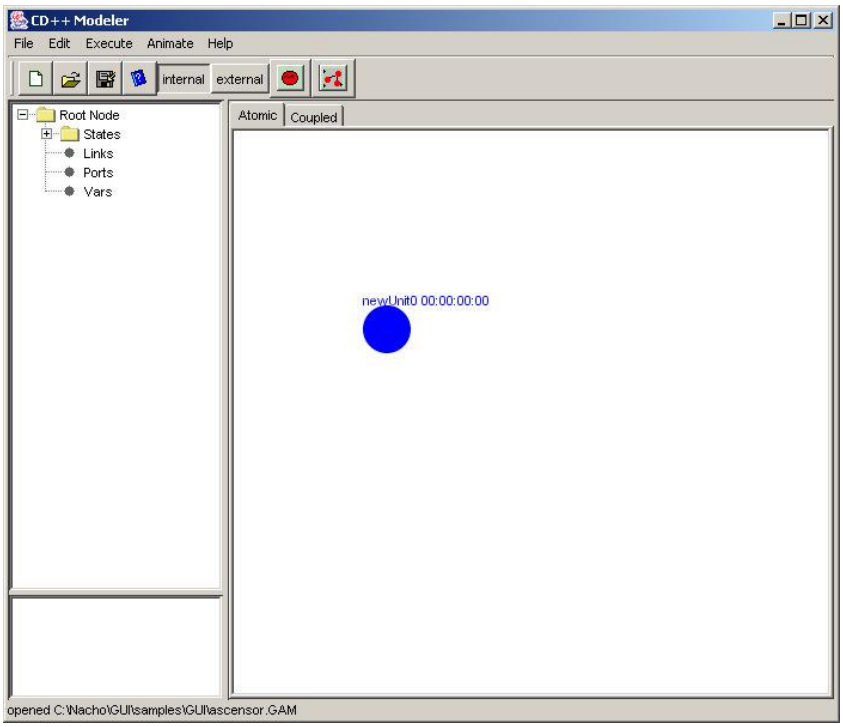
12.2.1.1 Setting Model Title

Click the “Model Title” item. A “Set Title” dialog will be shown.

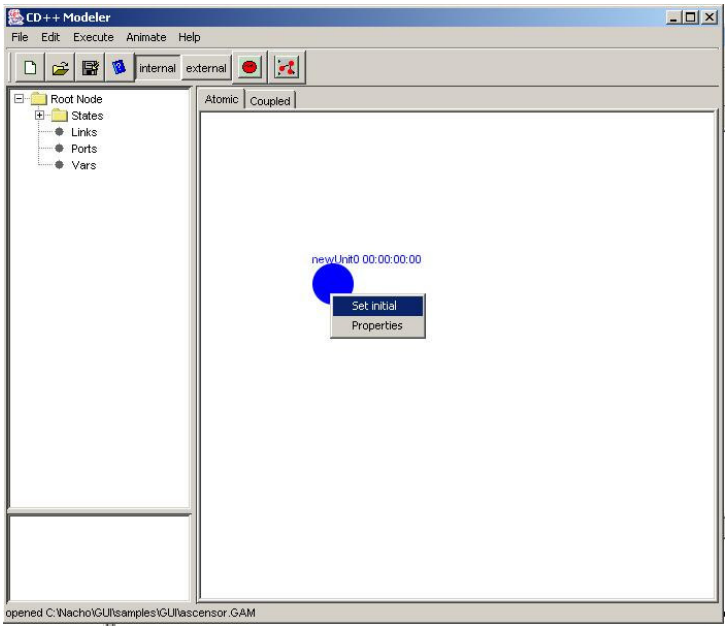
Complete the model name in the blank area and click the “Set” button.

12.2.1.2 Adding Units

Double click with the left mouse button on the design space. A unit will be drawn with blue color. Double click the left mouse button on the unit, and the unit will be selected with red color. Double click with the left mouse button on the selected unit, and the unit will be unselected (turns to be blue again).



Click the right mouse button on the unit, and a popup menu will be shown with various functions associated with this unit.



Select “Set Initial” to set this Unit as the initial unit of the model. Setting other unit as initial, deselects the previous one

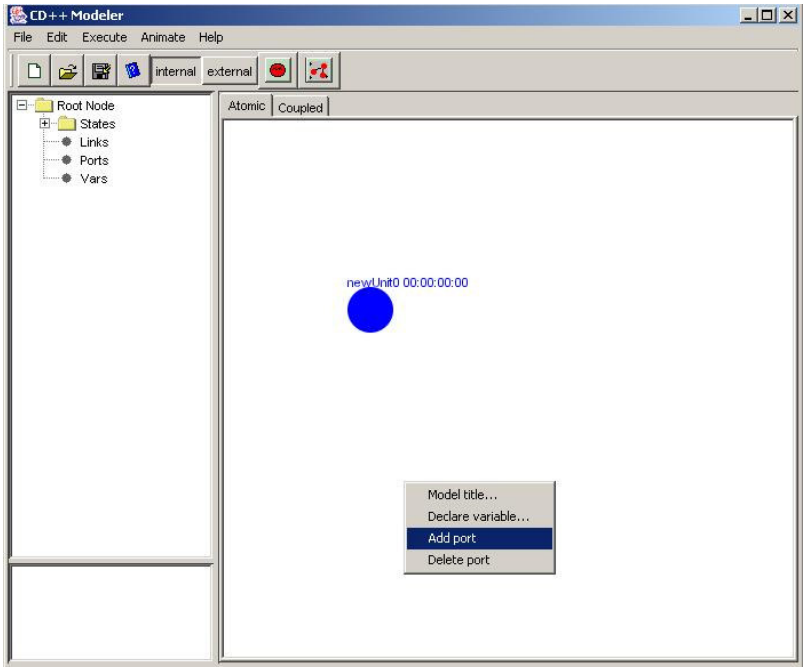
Select the “property” item, and a “state properties” dialog will be shown as in Fig 5. Here you can Fill the dialog with the state ID and Time to Leave.

To delete units, select the unit by double click on it, the color of the selected unit will become red. Then press the “Delete” button on the keyboard or select “Edit | Delete” from the main menu. All the selected units will be deleted.

When an unit is deleted, all the links connected to it will also be deleted.

12.2.1.3 Adding Ports

Click the right mouse button on the canvas and bring the popup menu. Select “Add port” item and an “Add Port” dialog will be shown. Fill the dialog, select proper properties for the port, click OK button. Repeat the procedure and add all of the necessary ports to the model.

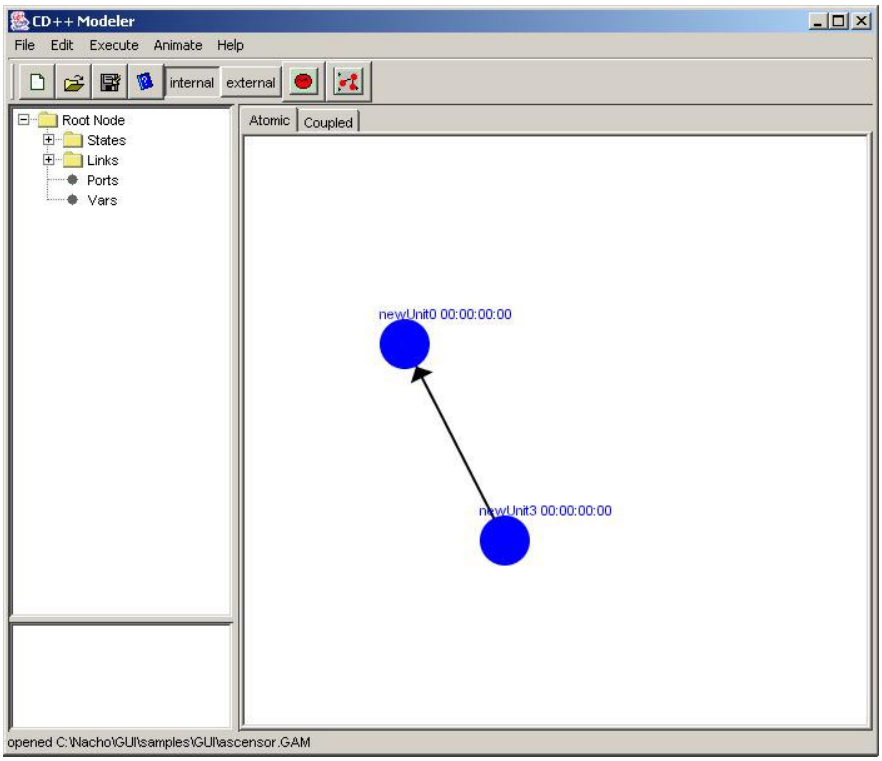


Selecting “Delete Port” the application shows a List of all the created ports and the user can select one to delete it.

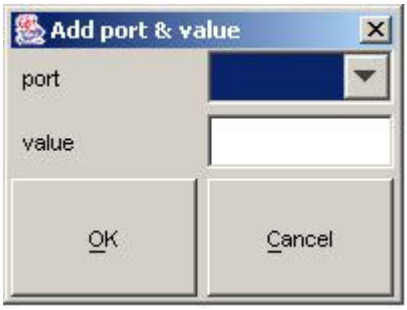
12.2.1.4 Adding Links

After creating all of the necessary Units and Ports according to the steps described in the above, we can draw links between Units, which represent the transition functions and attach them to Ports. The links in the atomic model are divided into two category: internal and external, which represent internal transition and external transition. Before we draw a link, the user should selected desirable link type by clicking “internal” or “external” button on the toolbar of the Interface.

To draw a link between two units, press the left button on one unit, hold the button and drag the mouse to another unit, release the button. A link with pre-selected type will be drawn. Fig 8 shows the result.



After drawing the links, we need connect the link with the port. To do this, right-click over the link and select "Add port & value" option from the popup menu. Select the port from the dialog, fill the value and click the OK button.



To delete links, select the link by double click on it, the color of the selected links will become red. Then press the "Delete" button on the keyboard or select "Edit | Delete" from the main menu. All the selected links will be deleted.

12.2.1.5 Saving and exporting

Whenever the user wants, the model can be saved to disk. To save the model, select "File | Save" or "File | Save As" as in any other application. The file will be saved as a .gam file.

After finishing drawing the model, we need to export the model to be a standard ".cdd" file that can used by the CD++ tools. To export the model, we select "File | Export" or "File | Save and Export" from the main menu, a file save dialog will be shown, select a directory and give the file a name. The file name must use ".cdd" as its extension.

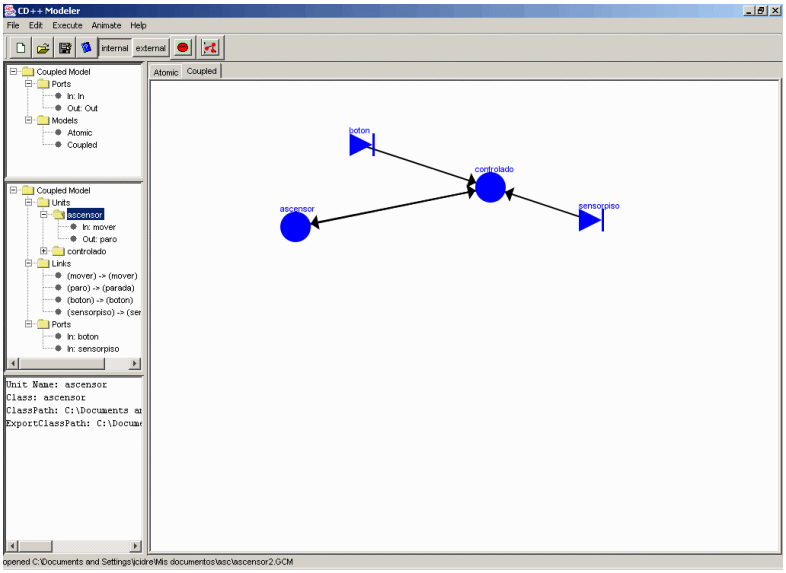
12.2.2 Create Coupled Models

The basic steps to create a coupled model are as same as creating an atomic model. In the following section, we will only highlight the special procedures of the coupled model.

The coupled model needs to be created on the coupled model work place, which can be selected by click on the "Coupled" tab above of the work place.

12.2.2.1 Distribution

The Coupled Models editor of CD++Modeler application is composed by a central panel, 3 lateral panels and an upper button bar



On the central panel appear the objects of the Coupled Model. It can be added Input Ports, Atomic Models, Coupled Models and Links between them.

On the first lateral panel they can be seen the objects to add for the definition of this model. It is divided in Ports (input and output), predefined Coupled Models and predefined Atomic Models.

On the second lateral panel they can be seen the objects that form the Coupled Model that is being defined. Units of the added Atomic or Coupled Models, Links between defined Models or Ports and Ports of the added Ports.

On the third lateral panel a description of the element selected on the second lateral panel.

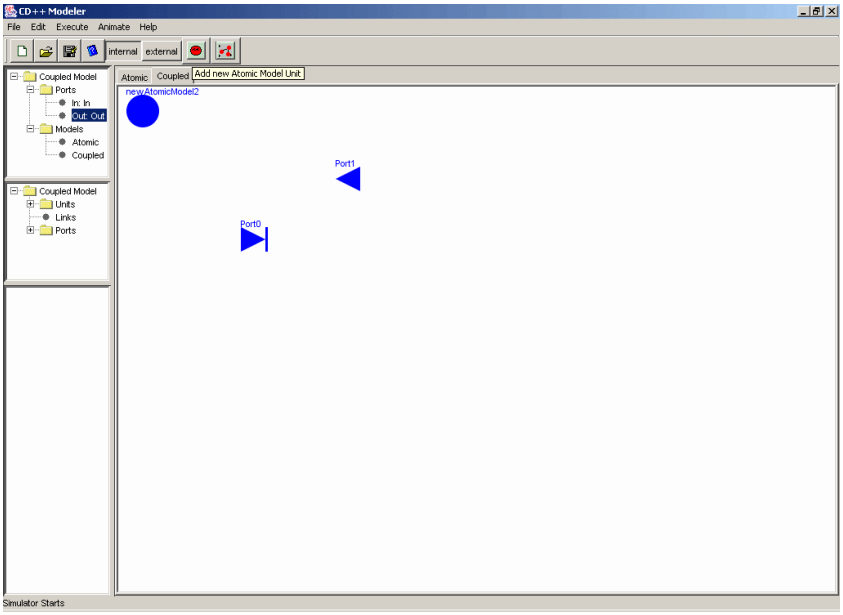
The button bar has the following buttons:

- **New,** Creates a new Coupled Model
- **Open,** Opens a previously saved Coupled Model.
- **Save,** It saves this Coupled Model
- **Help,** It brings the help

- **Internal,** Not used in Coupled Models
- **External,** Not used in Coupled Models
- **Add new Atomic Model Unit,** It adds a new Atomic Model to the Coupled Model under definition
- **Add new Coupled Model Unit,** It adds a new Coupled Model to the Coupled Model under definition

12.2.2.2 Adding Atomic Units

To add a new Atomic Model click on the "Add new Atomic Model Unit" button of the tool bar. Models are created in the left upper corner of the Modeling Panel. They are automatically created with a name. In this case newAtomicModel2.



A new Atomic Model is automatically generated with a name. This unit represents an atomic model not defined yet. To edit the unit, click the right button of the mouse to see the contextual menu. This menu has the following options:

- Properties
- Select Image
- Explode
- Reload

Properties allows to change unit name

Select image allows the choice of an icon for the unit.

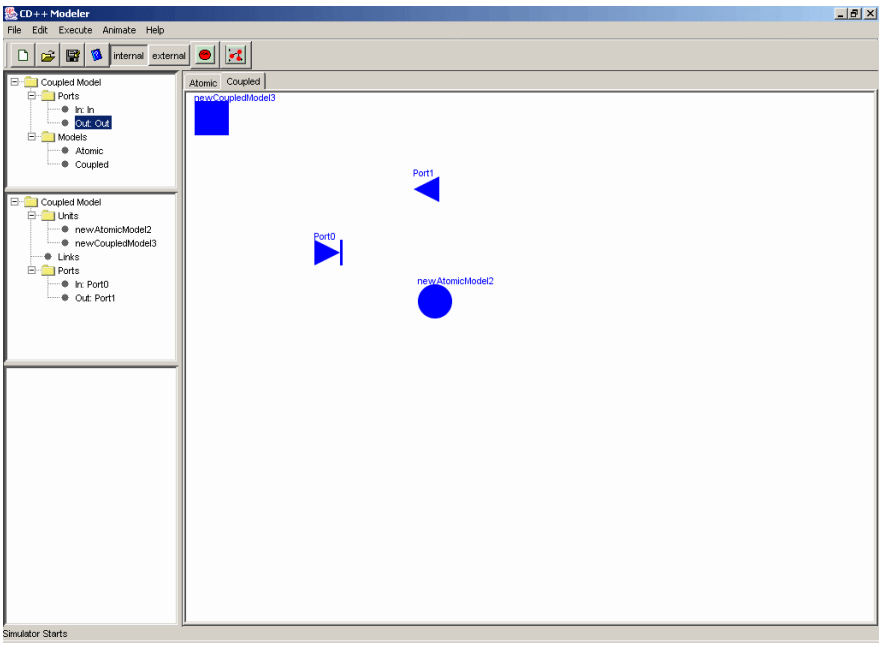
Explode allows to explode the unit for its definition.

Reload allows to reload the unit if its definition has been modified.

12.2.2.3 Adding Coupled Models

To add a new Coupled Model click on the "Add new Coupled Model Unit" button of the tool bar.

Models are created in the left upper corner of the Modelling Panel.They are automatically created with a name. In this case newCoupledModel3.



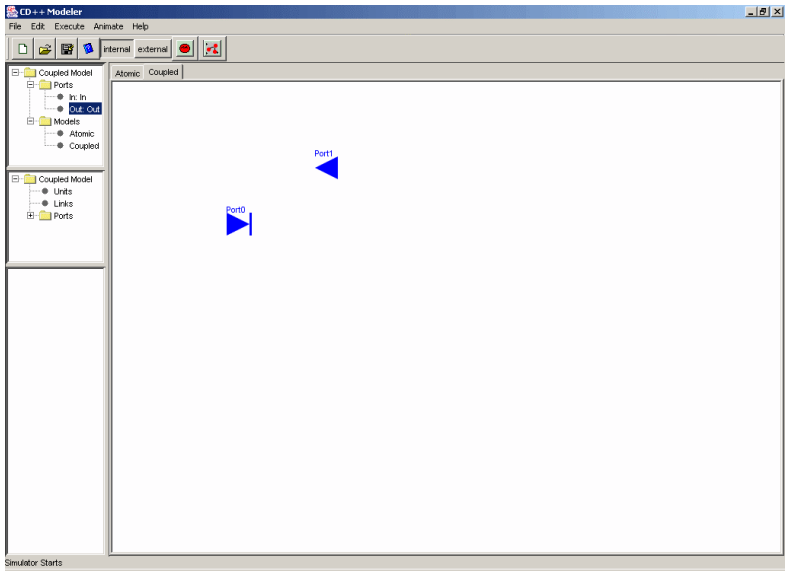
A new Coupled Model Unit is automatically generated with a name. This unit represents a Coupled Model not defined yet. To edit the unit, click the right button of the mouse to see the contextual menu. This menu has the following options:

- **Properties** allows to change unit name
- **Select image** allows the choice of an icon for the unit.
- **Explode** allows to explode the unit for its definition.
- **Reload** allows to reload the unit if its definition has been modified.

12.2.2.4 Adding Ports

In Coupled Models two types of ports can be created: Input Ports and Output Ports. They are characterized by their graphics.

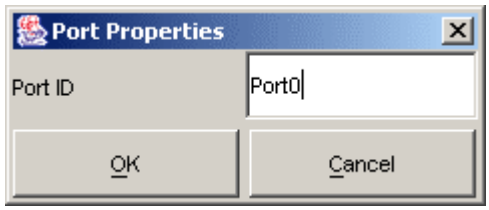
To add a Port, select the port in the lateral superior Panel and double click over the Modelling Panel



Ports are automatically created with a name. In this case, Port0 is an input Port and Port1 is an output Port.

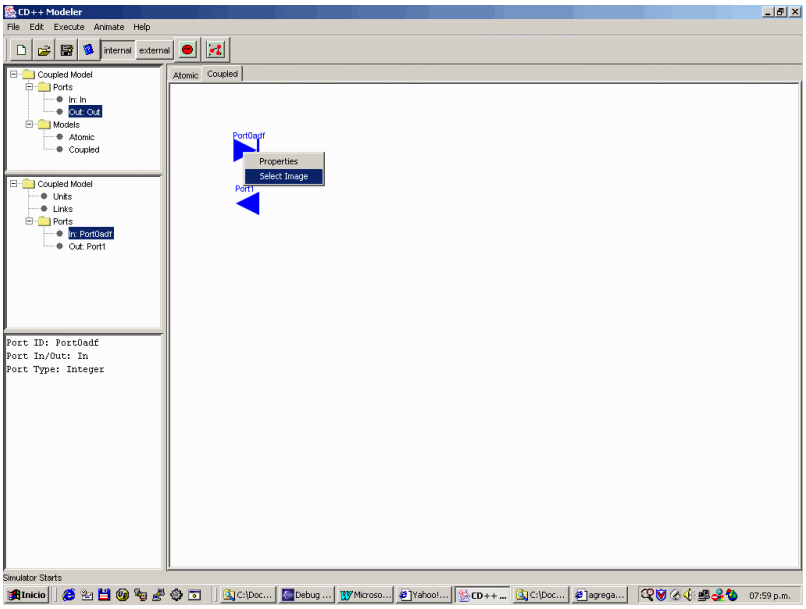
In this case Port0 is an input port and Port1 is an output port.

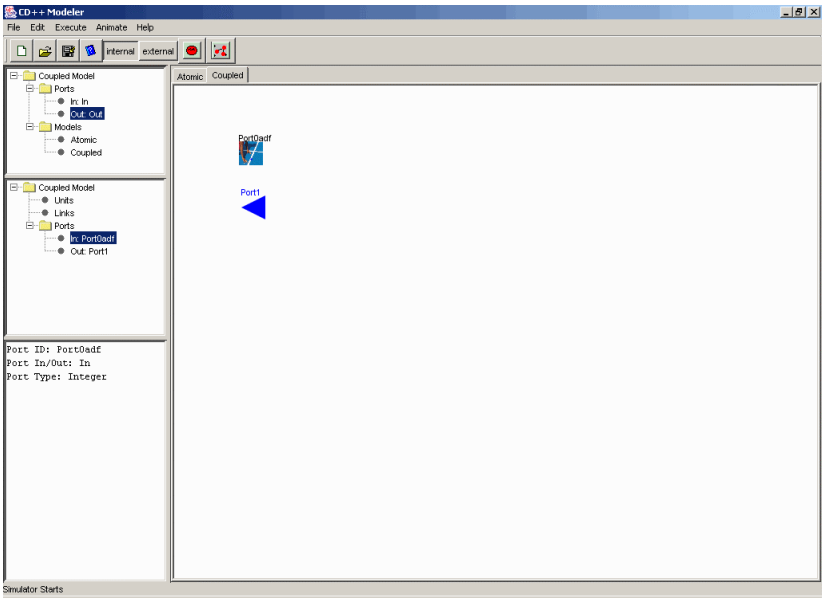
The Port's name can be changed either from the port contextual menu at the Modelling Panel or from the Units Tree. Another option is to double click on the Units Tree. The Cd++ Modeler opens the following window to select the port identifier.



From the contextual menu, an image for the ports can also be selected.

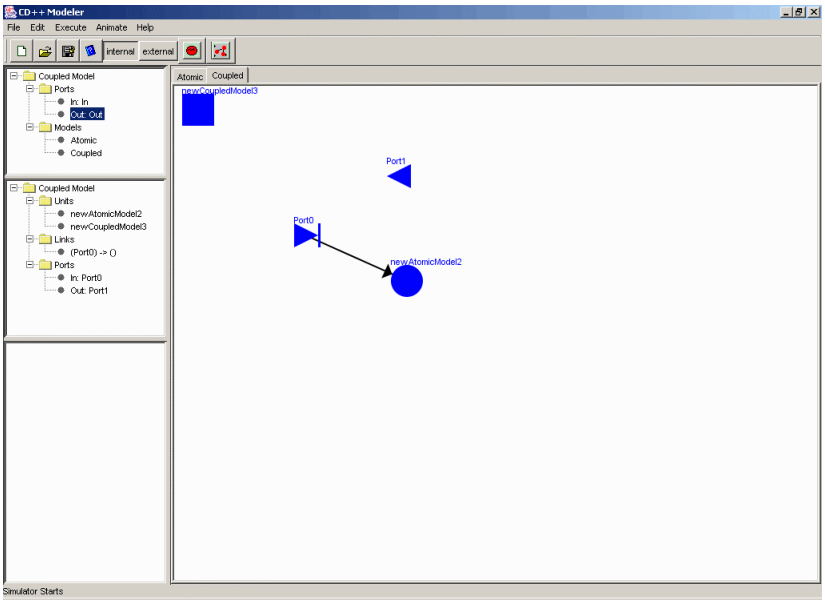
Selected image replaces default image.



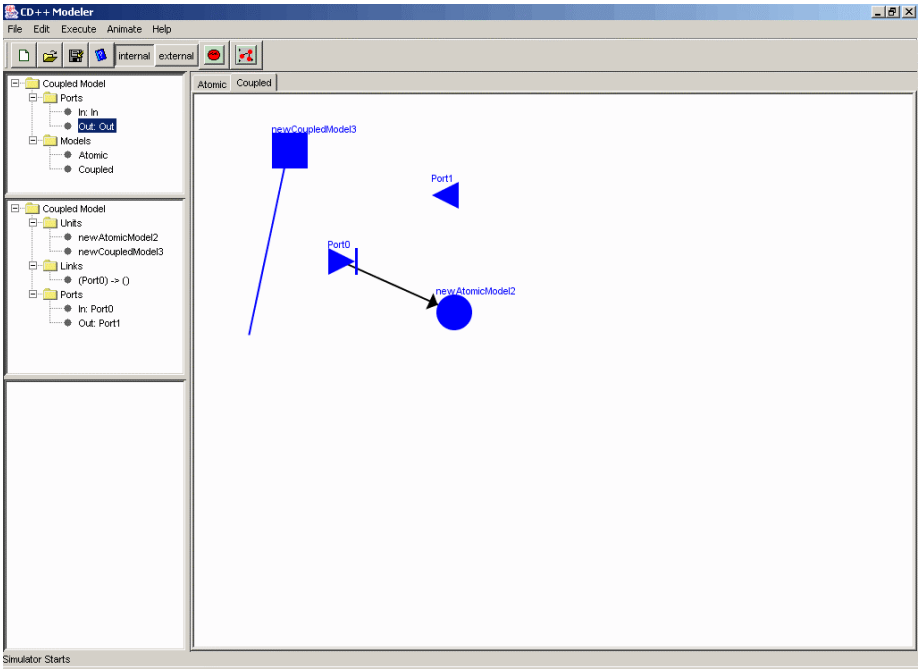


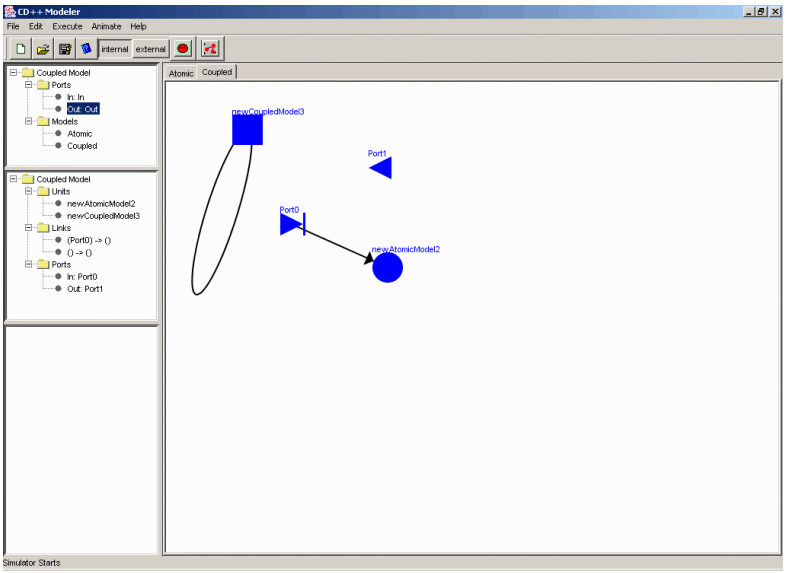
12.2.2.5 Adding Links

To add a Link click on the origin of it and drag until the destiny of the new Link. The Link will be created only if both, origin and destiny are valid. In this case a Link was created from Port0 to newAtomicModel2.



To create a Self Link from a Model to itself, click on the model of origin, drag well away and back to the Model and release the mouse button. CD++Modeler will ask if a Self Link must be created.



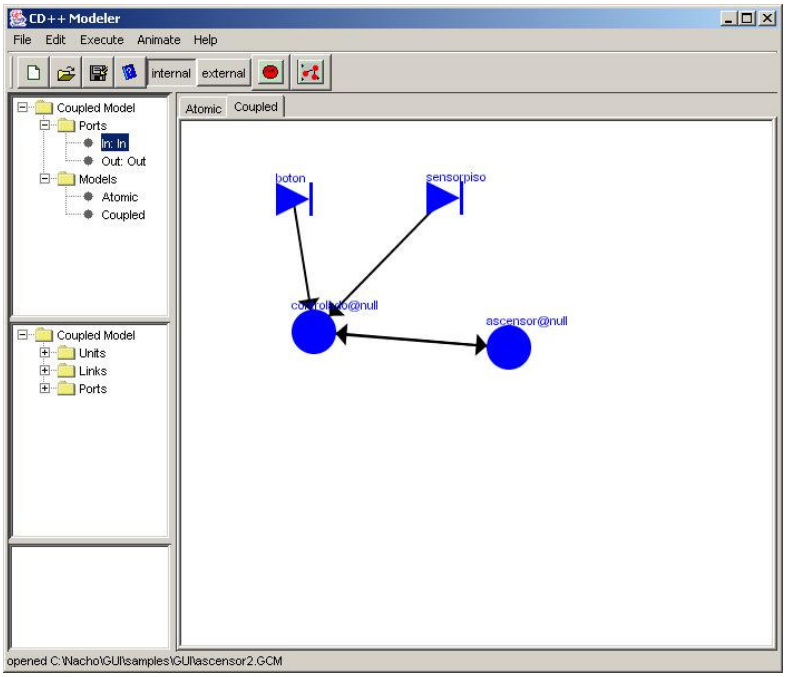


12.2.2.6 Importing Models

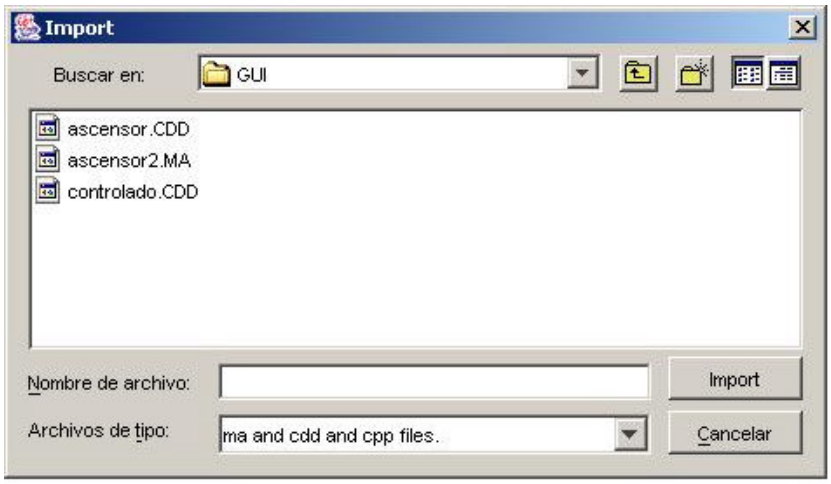
A Coupled Model is composed by Atomic Models and/or Coupled ones. These can be imported from predifined models from three diferent inputs:

- Predifined Coupled Models .ma
- Predifined Atomic Models .cdd
- Basic atomic models from register.cpp file

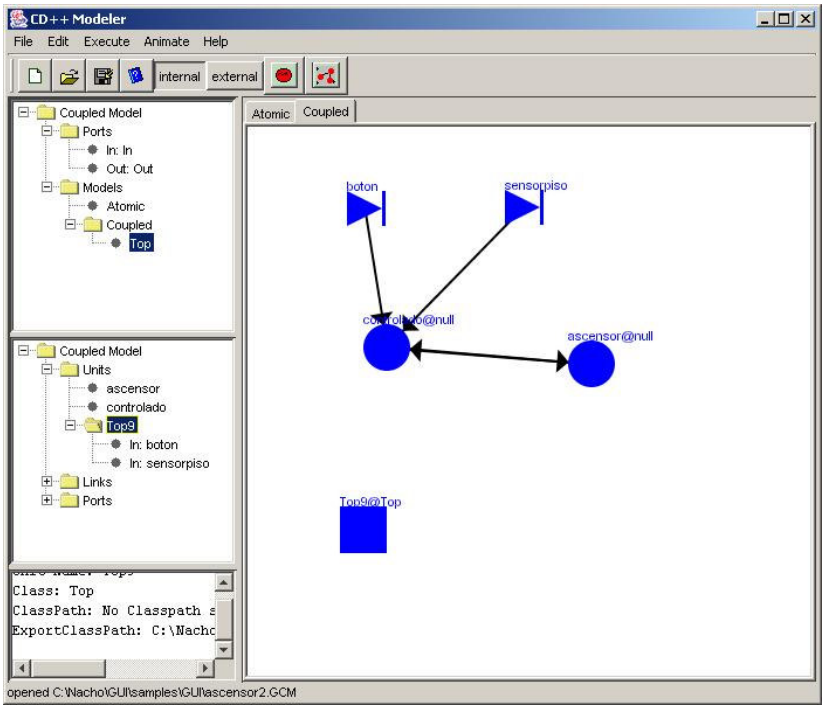
Having a Coupled Model like the following



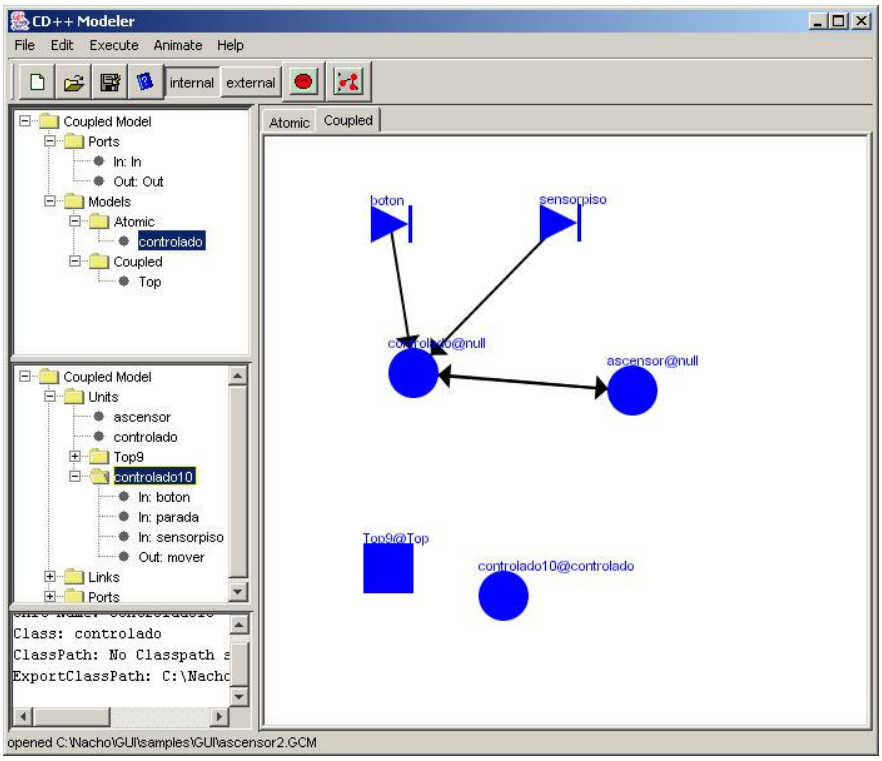
predefined models can be imported using File -> Import option
This opens an Open File dialog that allows the user to import a .cpp .cdd or .ma file



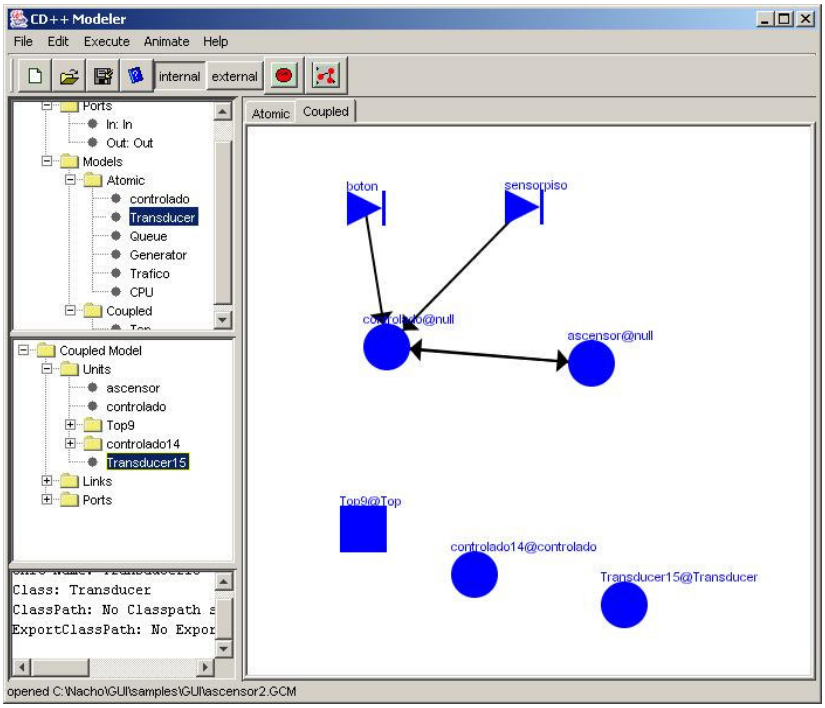
In this case, a coupled model definition file (.ma) was selected and imported. The imported model appears in the models Tree under the 'coupled' folder. When a Coupled Model Unit based on this imported model is added, it appears with the predifined input and output ports. This Unit cannot be modified



Next, an Atomic Model definition file (.cdd) was selected and imported. The imported model appears in the models Tree under the 'atomic' folder. When an Atomic Model Unit based on this imported model is added, it appears with the predifined input and output ports. This Unit cannot be modified

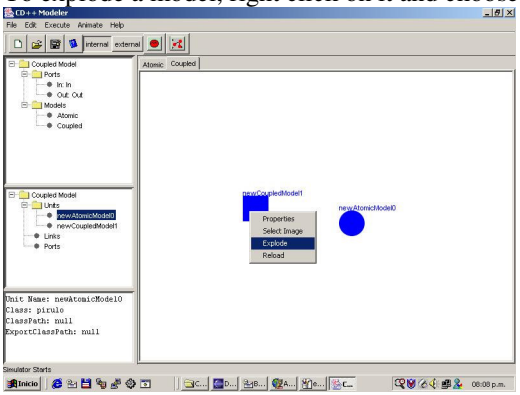


Next, a register.cpp file was selected and imported. All the models defined in the field appears in the models Tree under the 'atomic' folder. When an Atomic Model Unit based on one of these imported models is added, it appears without any predifined input or output ports since the register.cpp dont has this information.
Each imported model could be modified right clicking on it to add the ports. This Unit cannot be modified

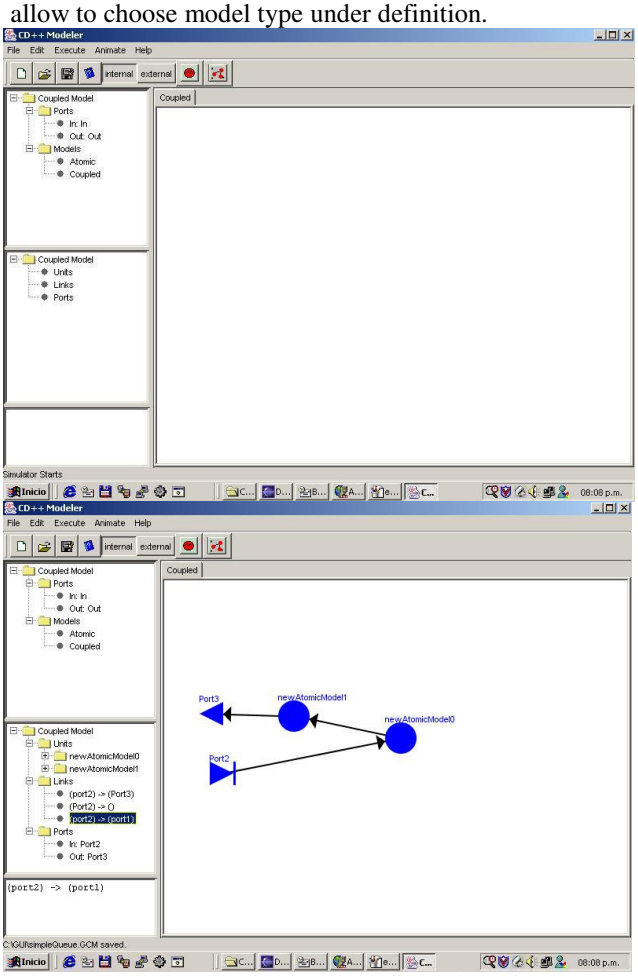


12.2.2.7 Exploding Models

A Coupled Model is composed by Atomic Models and/or Coupled ones.
These can be added from the predefined models list or as new models from the tools bar.
In both cases, added models may be inspected in order to be defined or modified.
To explode a model, right click on it and choose "Explode" option.

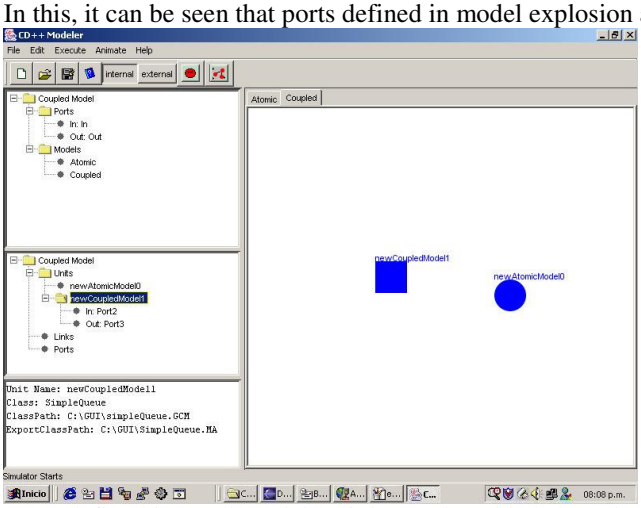


This action opens a new model editor to edit present exploded model and hides the original one.
If the model being exploded is an Atomic Model, it will be opened an Atomic Models editor;
if it is a Coupled Model an editor for Coupled Models will be opened.
Models so defined or edited work exactly the same as any other model except that they do not allow to choose model type under definition.

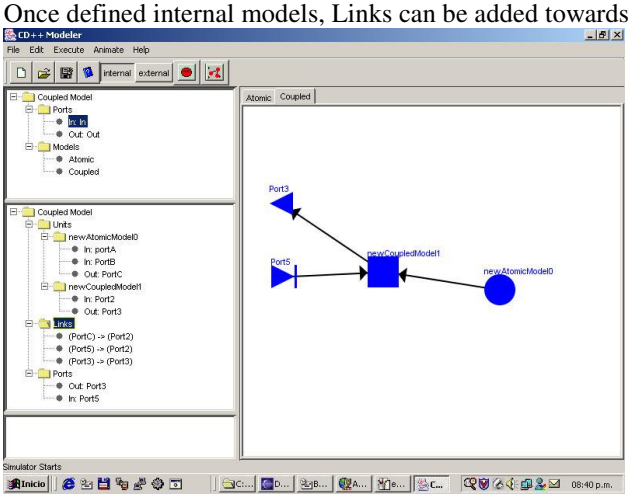


When exploded model is already defined the editor must be closed to return to the original model.

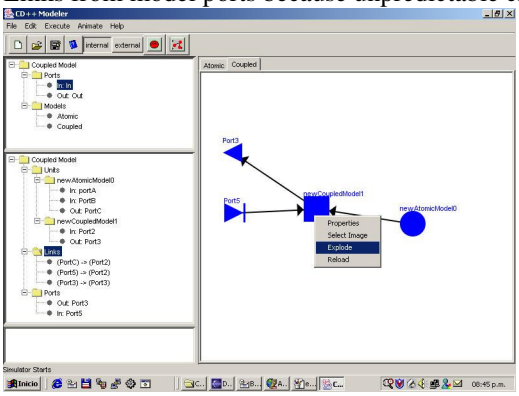
In this, it can be seen that ports defined in model explosion are accesible. See tree at left.

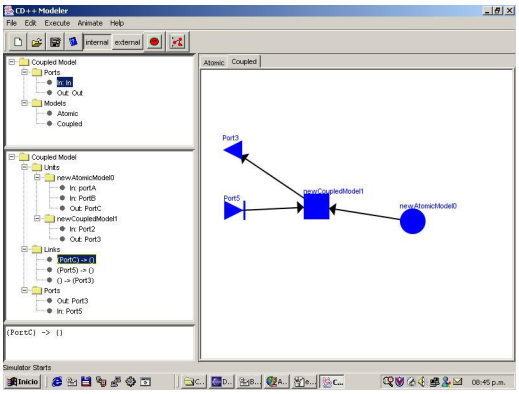


Once defined internal models, Links can be added towards ports defined in them.



If after definition of Links, a model is exploded for modification, system automatically disengages Links from model ports because unpredictable changes may happen.





12.3 Visualization of DEVS models

This section will describe how to visualize the result files of Cell-Atomic models, Cell-DEVS models and DEVD Coupled models.

Model type selection

Select the “Animate” from the main menu, a pull-down menu for model type selection will be shown as the following Fig. 12

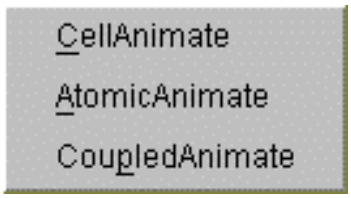


Figure 12

From this pull-down menu, Atomic-DEVS model, Cell-DEVS model and Coupled-DEVS model can be specified.

Atomic-DEVS model

Every DEVS model includes at least one atomic model. Usually a coupled DEVS model includes many atomic models. After the simulation finishes, a log file will be generated. The log file records all the messages sent between DEVS components, that is, all the messages sent and received by all the atomic models have also been saved in the log file. Therefore, with this log file all the values in the messages sent and received by a specific atomic model can be extracted and visualized.

To visualize the values messages sent and received by a specific atomic model, you can use the following steps:

Select “AtomicAnimate” item in the pull-down menu in Fig. 12.

An animate Dialog box will open, as the following Fig. 13

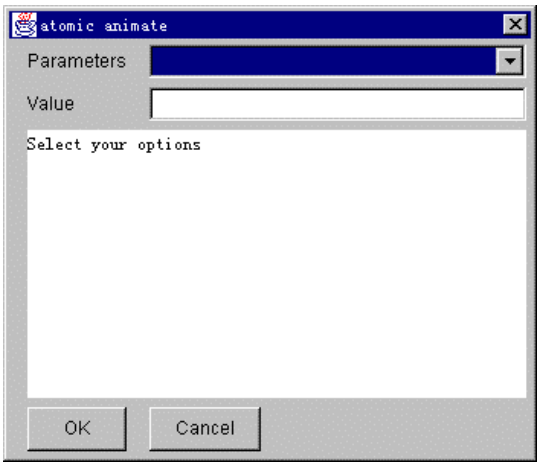


Figure 13

Select parameter “Log file”, a file load dialog will shown, a `log` file can be loaded. Some help and description information is also shown in the text area.

Click “OK” button after selecting the `log` file. Then the graphic display of the output of this atomic model will be shown as the following Fig. 14.

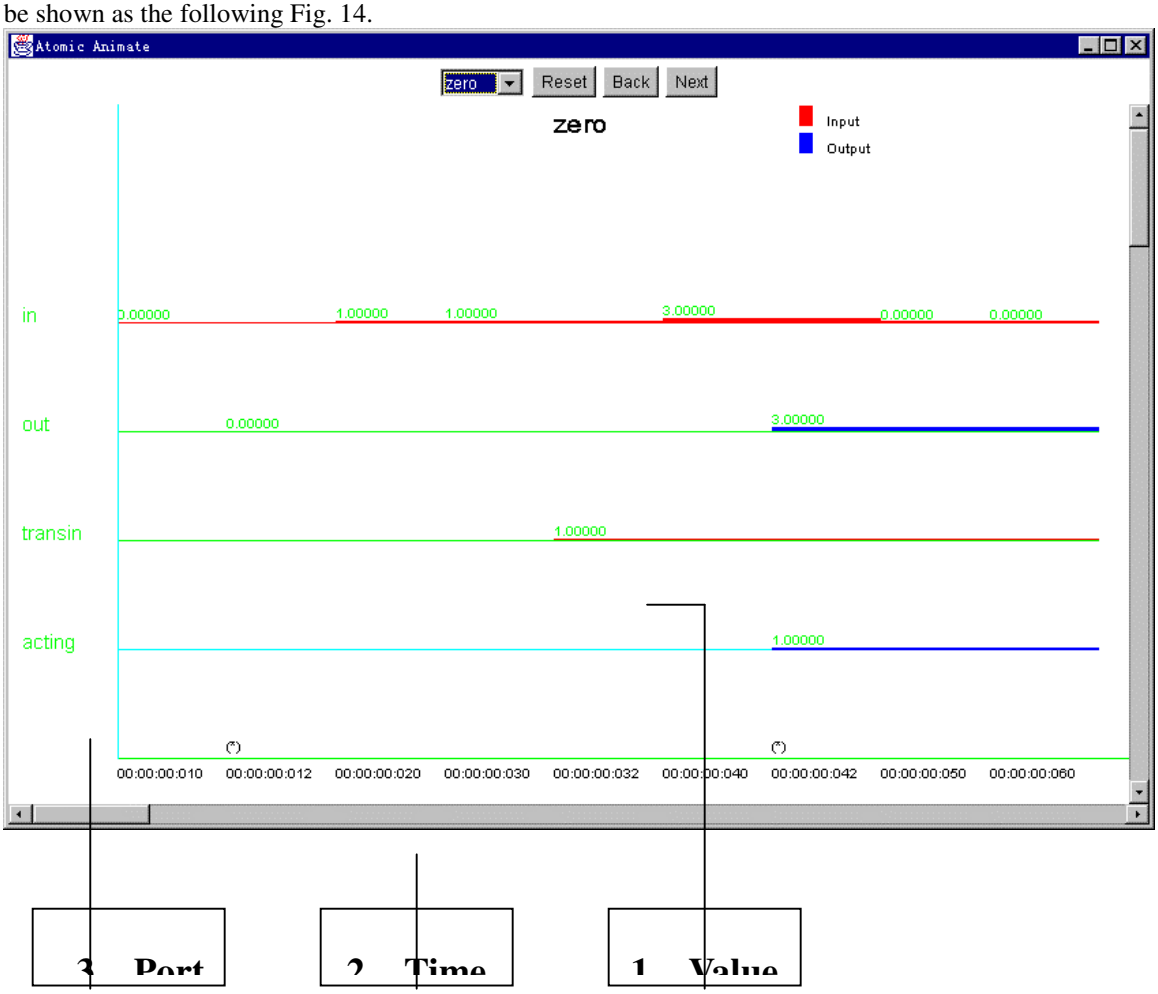


Figure 14

(*) means there is an internal transition at this time.

With the “Next” button and “Back” button, you can get to the graphic display of the output of a specific atomic model at any time until the end of the result.

With the “Reset” button, you can get to the beginning of the display at any time.

With selection list on the top, you can select any model at any time for visualization, such as, you want to change to `twenty` model, as the following Fig. 15.

Figure 15

Fig. 15 shows the output of the `zero` model in the following times by clicking the “Next” button in Fig. 14. And it shows that you want to change to see the `twenty` model. After you select the `twenty` model, its output will be displayed, as in following Fig.16.

Figure 16

12.4 Coupled-DEVS Model

Sometimes, it is useful to display the graph of the model, and the output values near the corresponding output port at the same time. To do this, you can use the following steps.

Select “CoupledAnimate” item in the pull-down menu in Fig. 12.

A coupled animate Dialog box will open, as the Fig. 27.

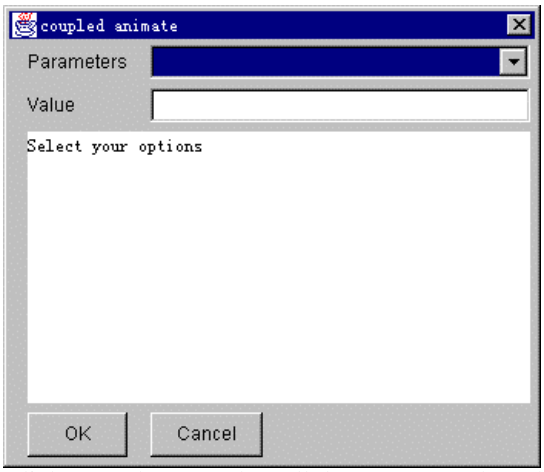
Select each parameter with the “parameters” list, and input or select their value respectively. Some help information will shown in the text area accordingly.

All the parameters are as follows:

Log File: the file used to record all the messages sent between DEVS components.

Coupled Model definition: the graph file (gui file) of the model.

Delay between displays: The delay between the continuous displays. It can be used to control the speed of the visualization.



Click “OK” button, when all the parameters have been specified.

Figure 27

The operations are all the same as those described for Cell-DEVS Model in section 3.2.

The following Fig. 28 is an example:

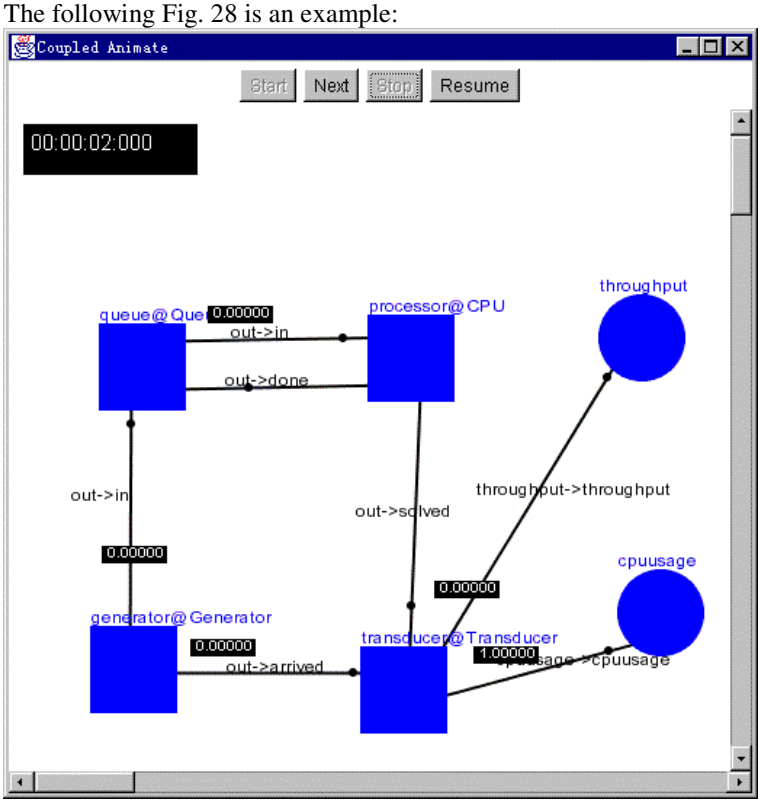


Figure 28
The visualization graph displays the model graph and the output value of all the output ports at the same time.

The four buttons are the same as those described for Cell-DEVS Model in section 3.2.

Warning Messages

If an input is wrong, some of the graphical outputs will not be able to run properly. When these situations occur, warning message is displayed.

For each dialog box of each model, if user forget to input some necessary parameter value and press a "Ok" button, warning message like the following Figure 29 will show:

Figure 29

If this happens, just close the warning window and go back to the dialog window, check and input the parameters values which you have forgotten, and try again.

Note if this warning appears you need to input the missing parameters again.

If inputting filename is needed in the dialog box, and you input the wrong file name or path, a warning will appear:

Figure 30

Also you need only to re-input the correct file name or path, and don't worry about other values you have already input. In the dialog box of each model, at any time, you could copy / paste the text by selecting the texts and using Ctrl-c to copy and ctrl-v to paste. The size of each dialog box can be enlarged by putting the mouse on the border and drag.

In order to load model, log files and multiple subcomponents, the following GUI components are added:

12.5 Visualizing Cell-DEVS models

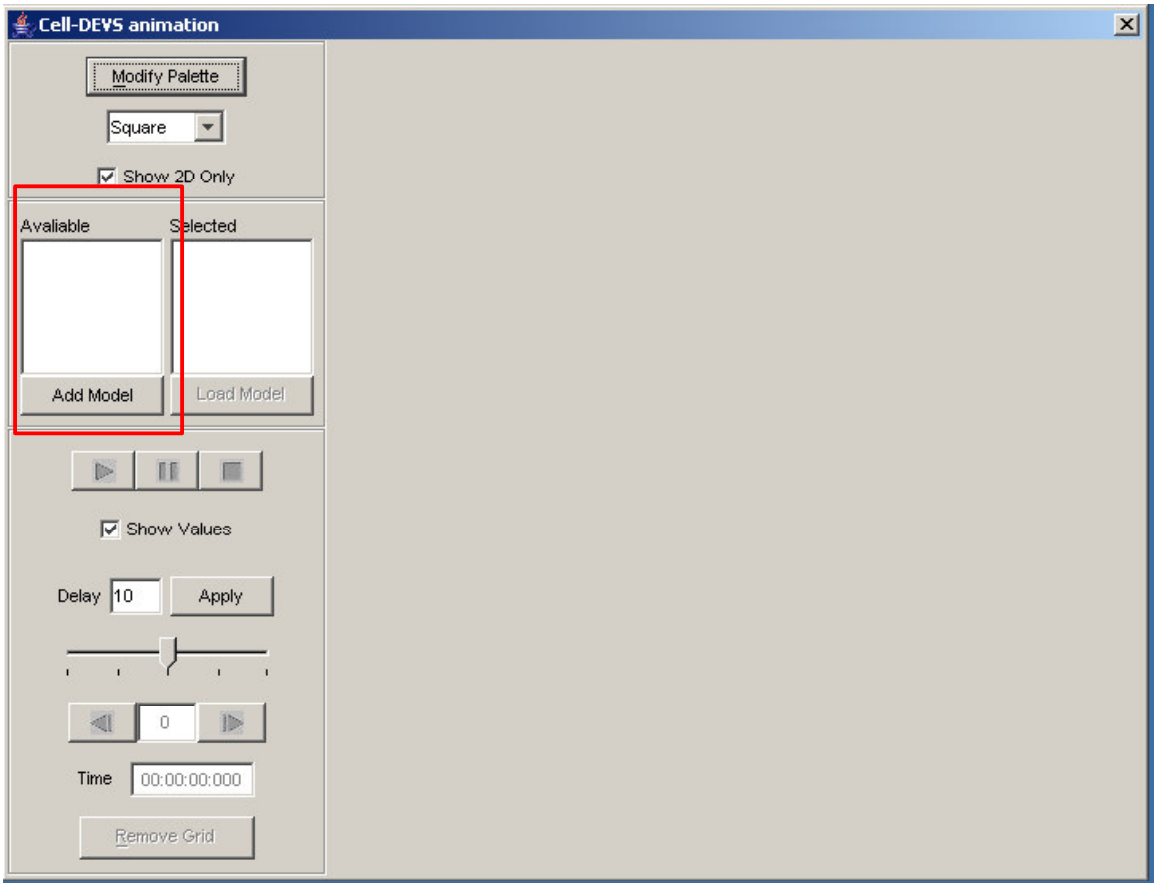


Figure 120 Add Model button and Available List

Add Model button is used to load *.drw, *.ma and *.log files. After these files are loaded, the corresponding model names will be shown as a list in the Available list.

When click “Add Model” button, the following dialog appears. With this dialog, you can choose a model (.ma), drw or any other type file. If the file extension is drw, CDModeler treats the file as a DRW file, otherwise, CDModeler tries to load the file as a model file, if failed, then consider it as a DRW file.

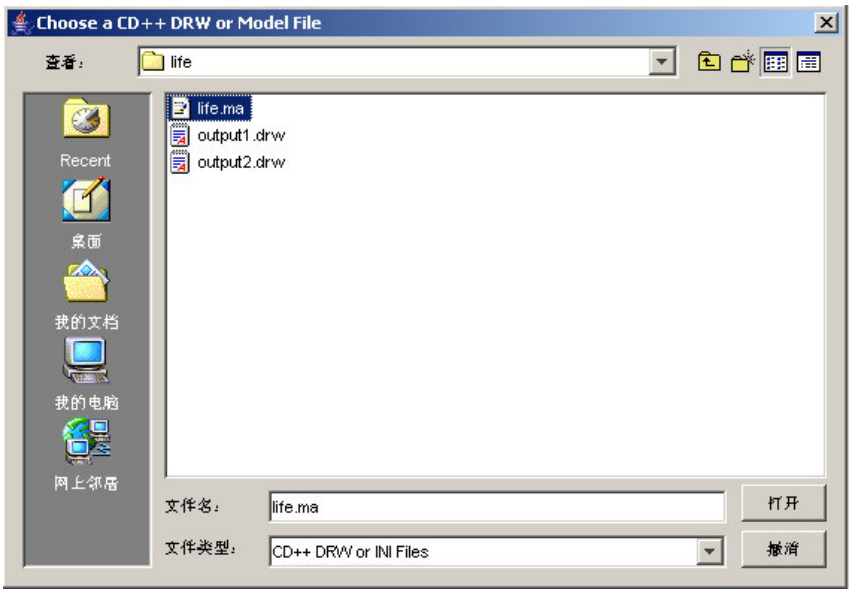


Figure 121 Open Ini/Drw file dialog box

When CDModeler parses the file and think it is a legal mode file (include [top]\n components : …), it automatically opens the following dialog to let user indicate which log file he/she wants to open for getting data.

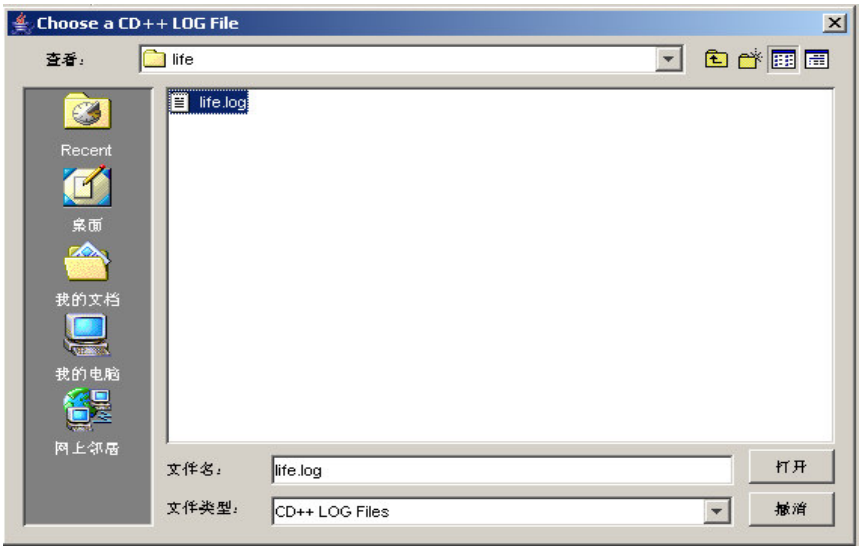


Figure 122 Open Log file dialog box

After that, the new model(s) are added to the end of Available Models list. You can add as many models as you want. To remove one from available models list, select it and press “Delete” key. The names of models have following format. Note there may have 0, 1 or more cell models defined in one model file.

File Type	Name in Available model list	Examples
DRW file	<drw_filename>	output, traffic
Model file	<cell_modelname>@<log_filename>	life@life, segment1a@traffic, segment1b@traffic

		segment1b@traffic

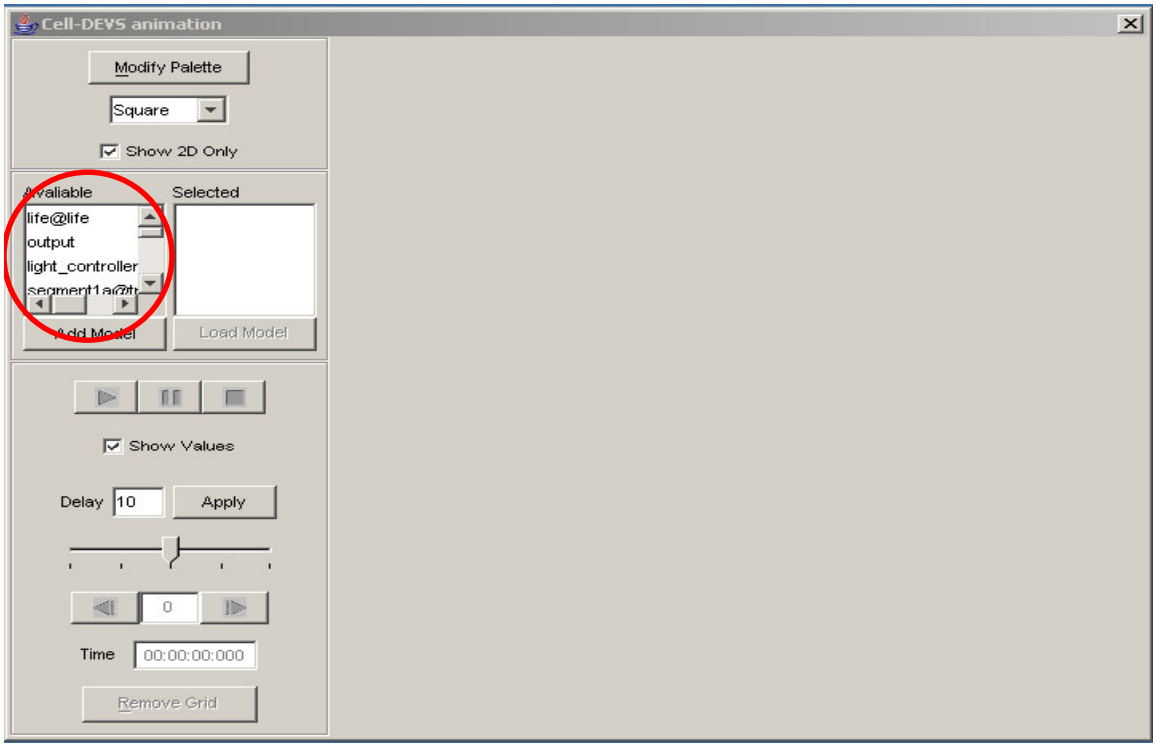


Figure 123 Add Multiple Models From Different Sources

12.5.1 Load Model button and Selected list

Even through you have a list of available models; they really have not been loaded in memory for display. To make a model visible, you have to double-click it in available list to append it to the end of "Selected Models" list, then press the "Load Model" button. To remove one from them selected list, double-click it or press delete key after choose it.

The display sequence of selected models is in the order of their names in selected list. At any time after you have made changes to the selected list, press load model button to re-display the new selected models. In case of the selected list is empty, the load model button is automatically disabled.

When all selected models have been loaded, CDModeler tries to load PAL file according to the name of the first entry in selected list as following. If such pal file does not exist or values are not defined in PAL file, CDModel now use white (previous use BLACK) as default.

Model name	**Searched PAL file**

<modelname>	<modelname>.pal
<modelname>@<log_filename>	<log_filename>.pal

Currently, CDModeler does not allow loading both time-unsupported and time-supported models at same time. An error message is generated when that happens.

Model Type	**Model Generated from**
Time-unsupported model	Drawlog with –f argument
Time-supported model	Drawlog without –f argument
	Load directly from log file

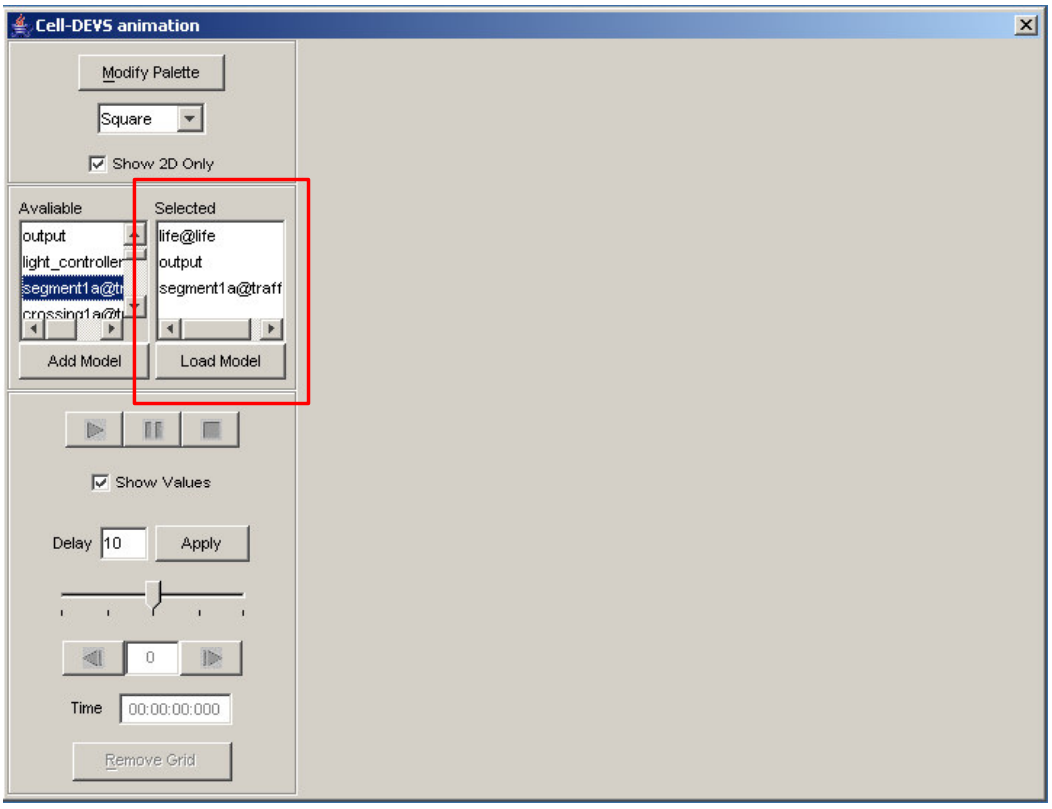


Figure 124 Load Model button and Selected List

12.5.2 The "Show 2D Only" checkbox

This checkbox is used to tell CDModeler how 3-dimensions models are displayed. For a 3-dimension model, if this checkbox is selected, only cells in its first plane are displayed, i.e. only cells with coordinates (x, y, 0) are displayed. Otherwise, all planes are displayed from left to right, i.e. in sequence of (x, y, 0), (x, y, 1), (x, y, 2) … etc.

For models loaded directly form log file, this checkbox also affect for N-Dimension model. If this checkbox is checked, only (x, y, 0,0, ...0) plane is displayed. Otherwise, from left to right, the sequence of displayed planes is (x, y, 0, 0, …0), (x, y, 1,0, ...0), … (x, y, D2, 0, ...0), (x, y, 0, 1,…0) … (x, y, D2, D3, …DN).

For 2-dimension only modes, the state of this checkbox is ignored.

12.5.3 Display simulation result for one model

*Users can use extended CD++ Modeler to load *.ma and *.log files and show the simulation result directly.*

Step 1. In the Cell-DEVS animation dialog box, click on the Add Model button, the Open Ini/Drw file dialog box appears.

Step 2. In the Open Ini/Drw file dialog box, travel to where the desired *.ma file is placed. In this example, we choose life.ma.

Step 3. The Open Log file dialog box automatically appears. Choose life.log and click on Open button.

Step 4. The model life@life appears in the Available list. Double-click on the model, the model will be selected and listed in the Selected list.

Step 5. Click on the Load Model button, the result is shown in Figure 125.

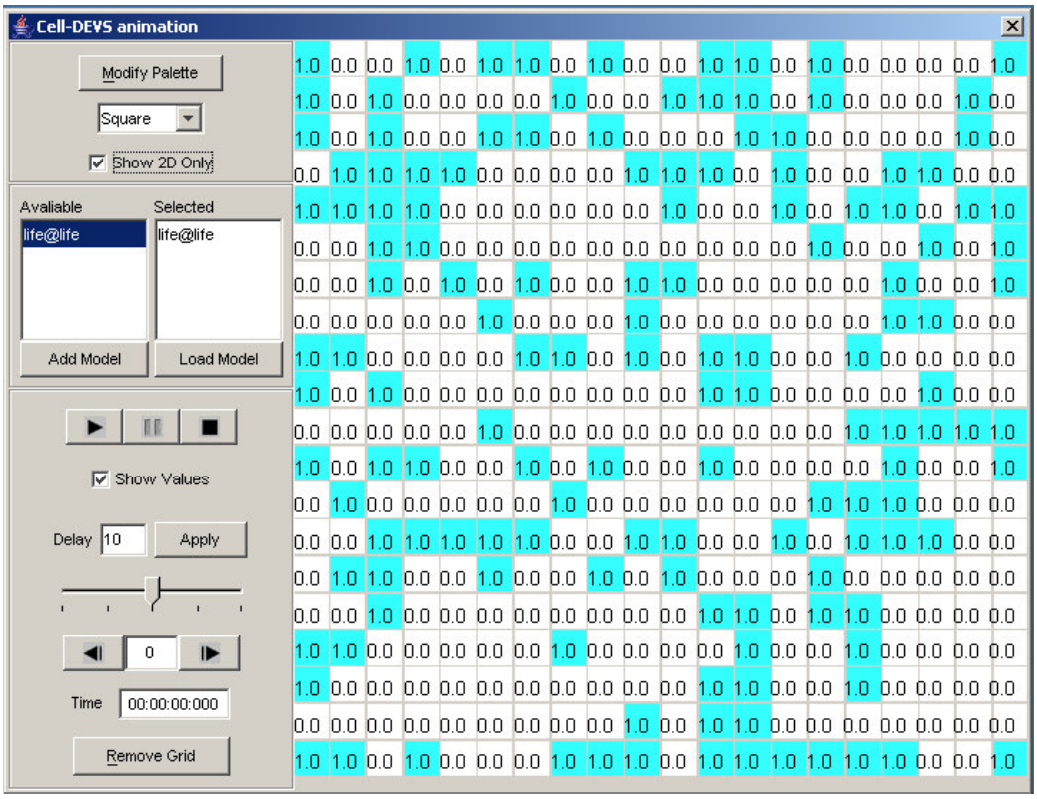


Figure 125 Life Simulation

12.5.4 Display simulation results for multiple models

12.5.5 1. Multiple Models from one Ini file and log file

Step 1. Following the step 1-3 described in Section *Display simulation result for one model*, load traffic model and its log file (available in our assignment 2).

Step 2. By double-clicking on the models, select the models you want to see. Please choose segment1a, crossing1a, light_controller1 , segment2a, crossing2a, light_controller2;then clock on the Load Model button to load the selected models, as shown in Figure 126.

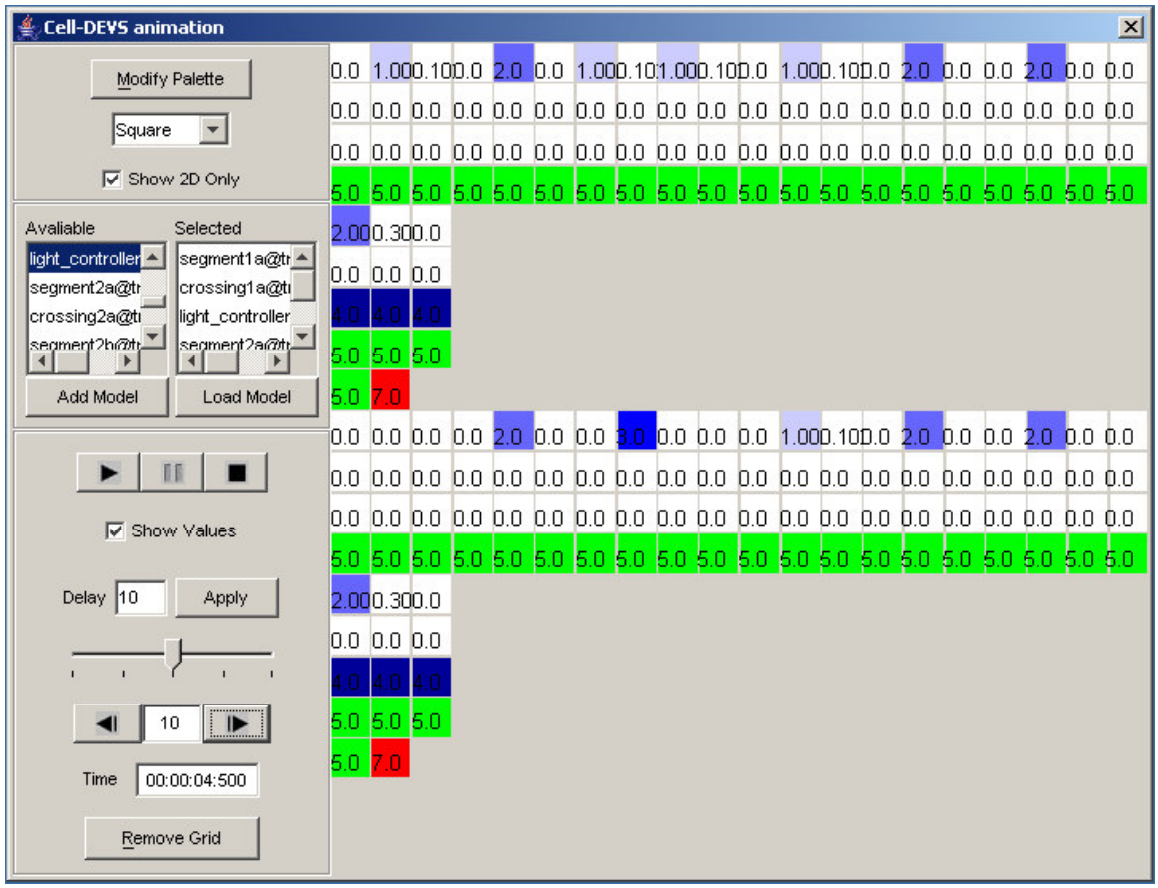


Figure 126 Traffic Model: Multiple Sub-models

12.5.6 2. Multiple Models from Multiple Time-Supported DRW Files

Step 1. In the Cell-DEVS animation dialog box, click on the Add Model button, the Open Ini/Drw file dialog box appears.

Step 2. In the Open Ini/Drw file dialog box, travel to where the desired *.drw files are placed. Open the *.drw file we want.

Step 3. Repeat Step1-2, load segemnt1a.drw, crossing1a.drw, segment2a.drw, crossing2a.drw.

Step 4. Select all the models by double-clicking. Click on the Load Model button. The result is shown in Figure 127.

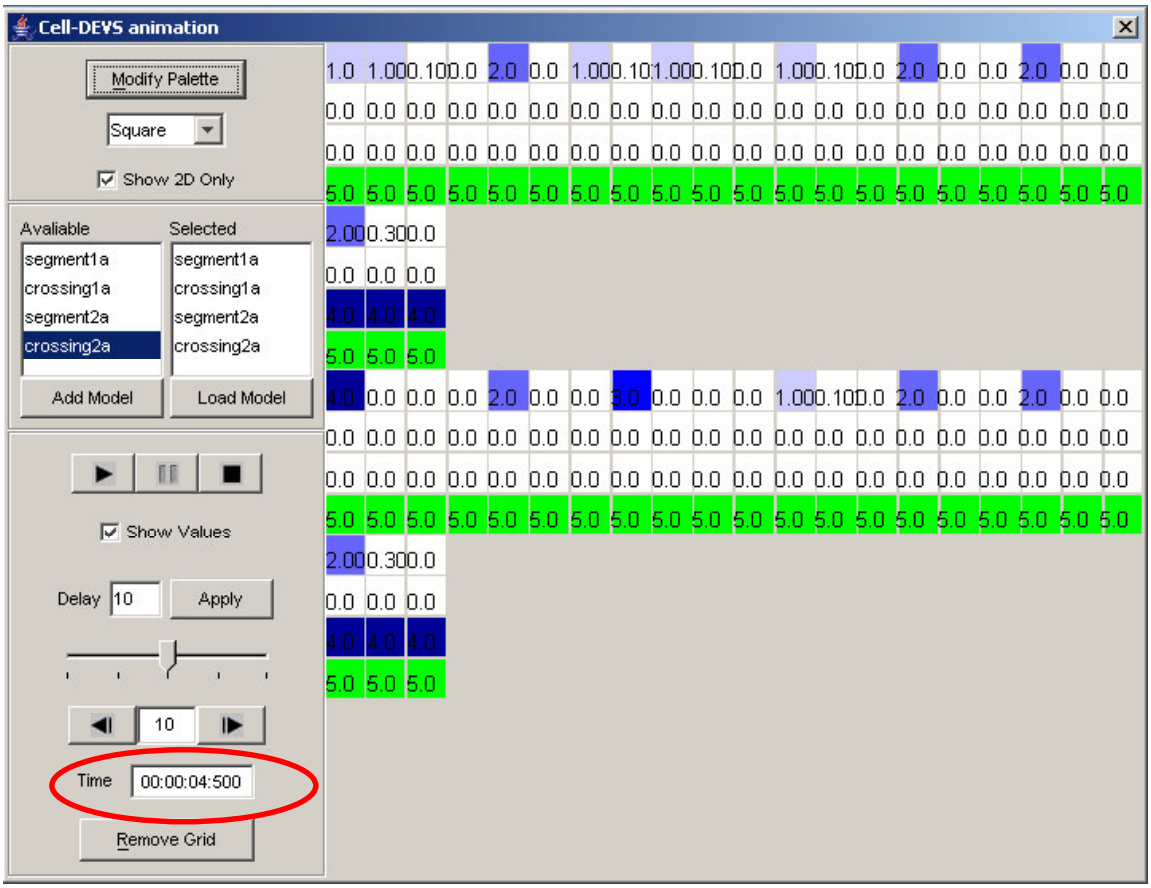


Figure 127 Traffic: from Multiple DRW Files

12.5.7 3. Multiple Models from Multiple Time-Unsupported DRW Files

Step 1. In the Cell-DEVS animation dialog box, click on the Add Model button, the Open Ini/Drw file dialog box appears.

Step 2. In the Open Ini/Drw file dialog box, travel to where the desired *.drw files are placed. Open the *.drw file we want.

Step 3. Repeats Step1-2, load output1.drw, output2.drw and output3.drw.

Step 4. Select all the models by double-clicking. Click on the Load Model button. The result is shown inFigure 128.

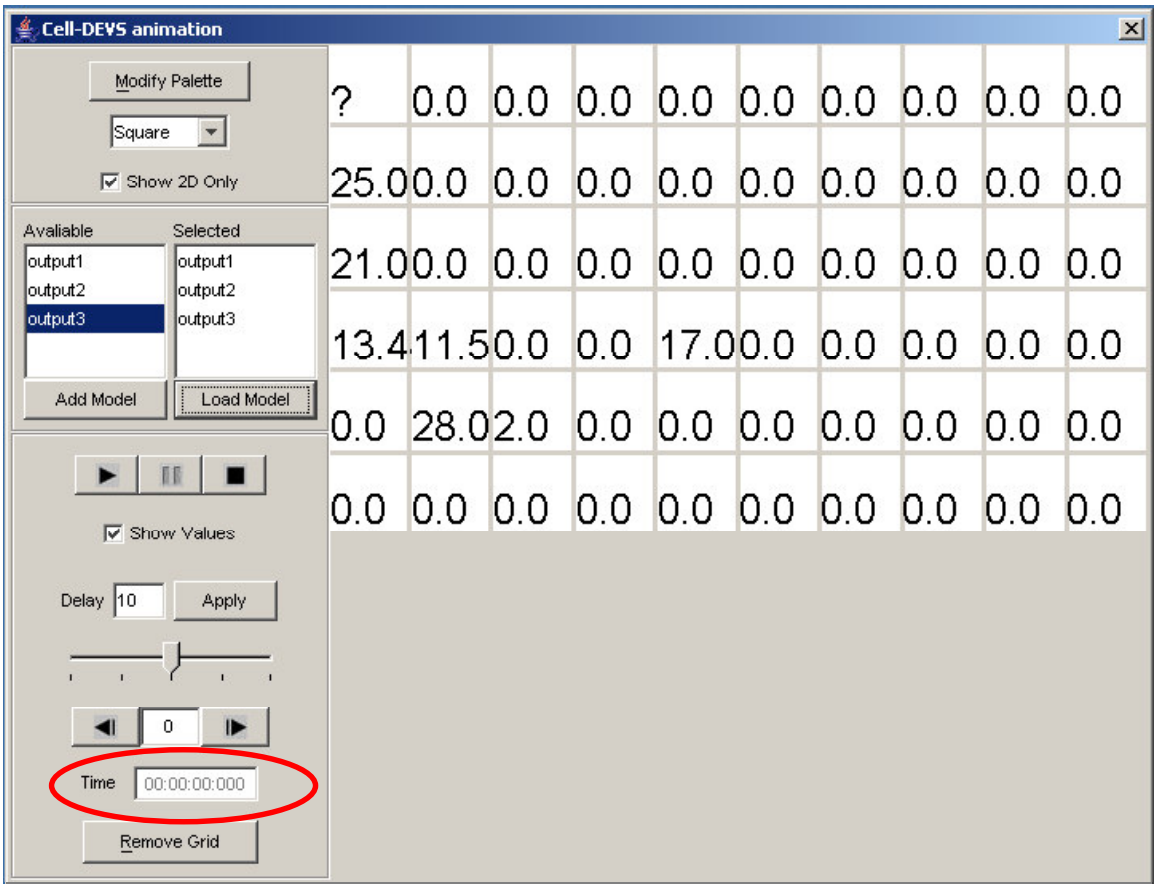


Figure 128 Life3D: from Multiple Time-Unsupported DRW Files

12.5.8 4. Multiple Models from DRW Files and Log Files

This can be used to validate the newly-added functionality.

Step 1. Load segment1a.drw file.

Step 2. Load traffic.ma, traffic.log files.

Step 4. Choose segement1a, crossing1a, segment1a@traffic and crossing1a@traffic models by double-clicking on them. Load the models by clicking on the Load Model button.

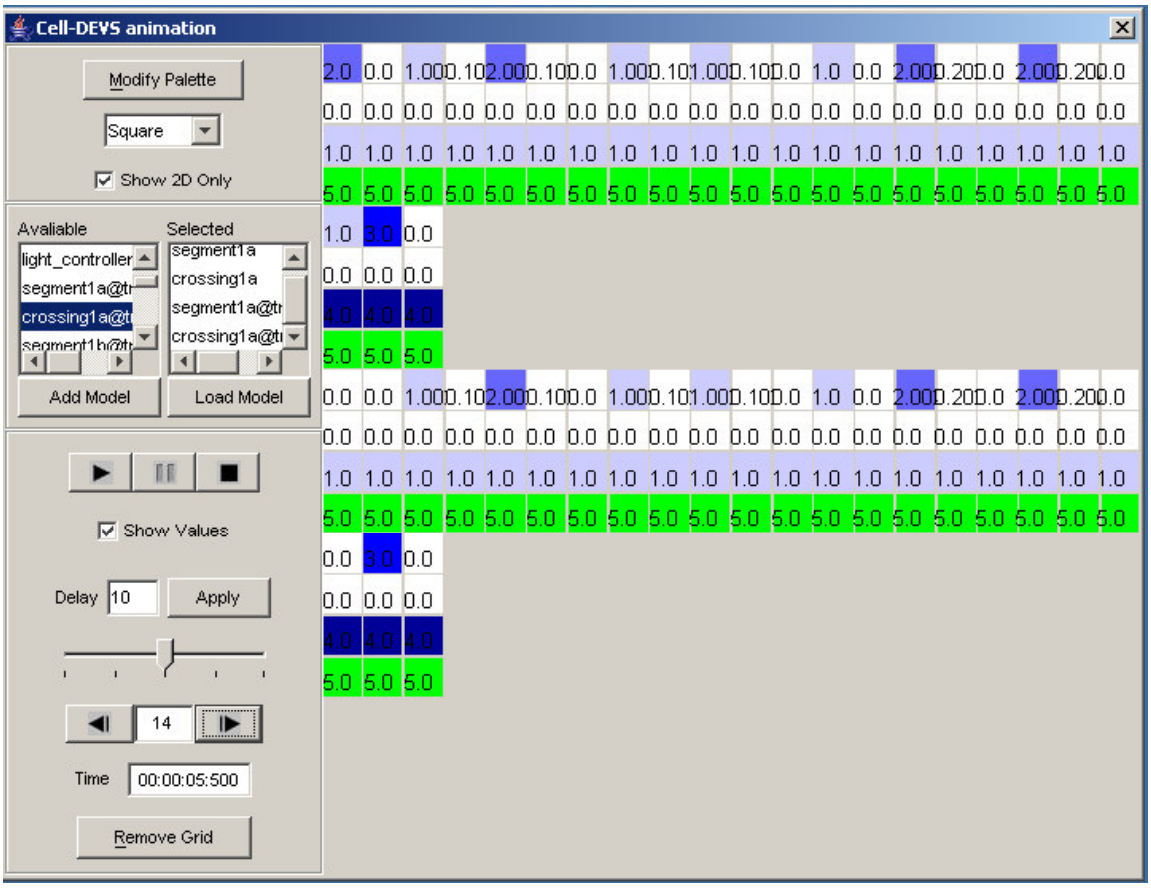


Figure 129 Traffic Models: from DRW and LOG Files

Here, we can clearly see the bug of drawlog program: at this time, the (0,0) cell of both segment1a and crossing1a models send value 2 and 1 out through in_space port, but these values are considered as value of (0, 0) cells. The lower two models loaded from log file are displayed correctly.

12.5.9 5. Show 2D / 3D Models

Step 1. In the Cell-DEVS animation dialog box, click on the Add Model button, the Open Ini/Drw file dialog box appears.

Step 2. In the Open Ini/Drw file dialog box, travel to where the desired *.drw files are placed. Open the *.drw file we want.

Step 3. Repeats Step1-2, load life@life and segment1a.drw.

Step 4. Select all the models by double-clicking. Click on the Load Model button. The result is shown in Figure 130.

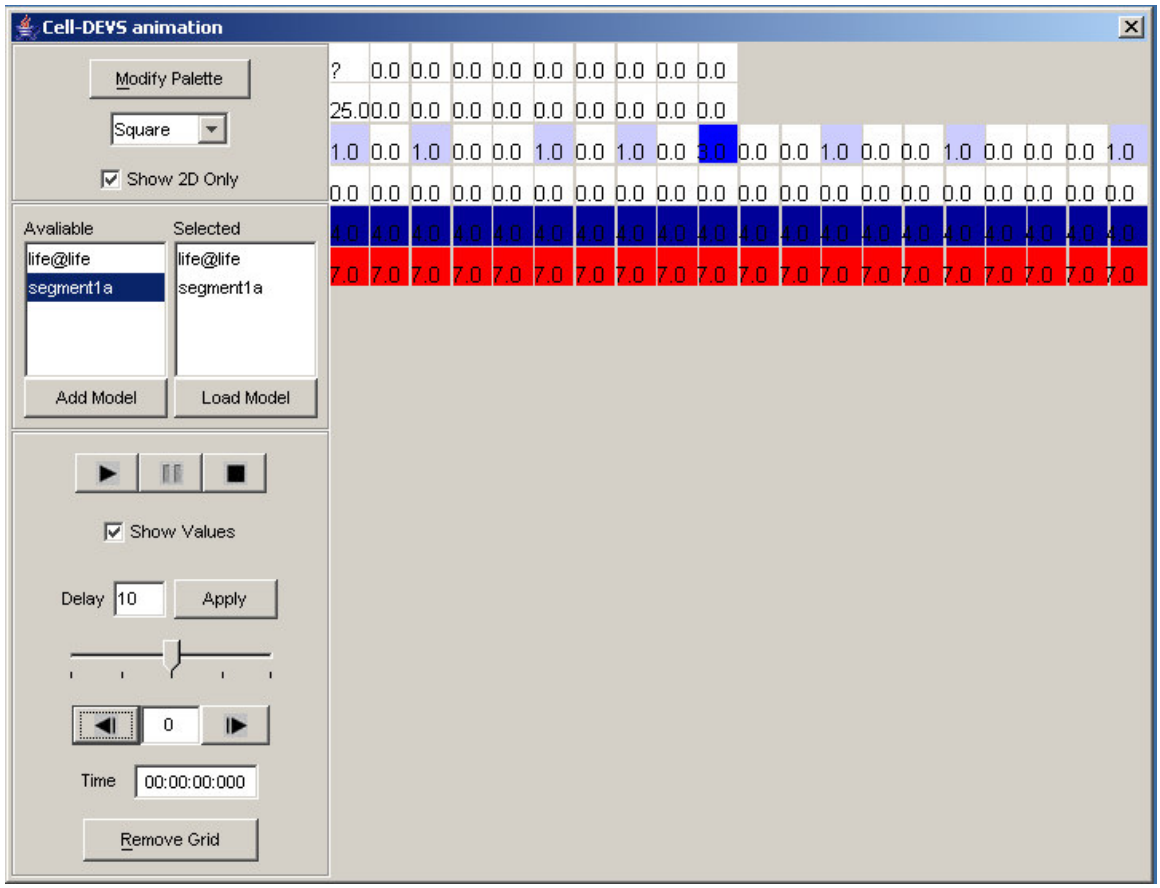


Figure 130 Show 2D model in 2D mode

Step 5. Deselect the "Show 2D Only" checkbox, The result is shown in Figure 131.

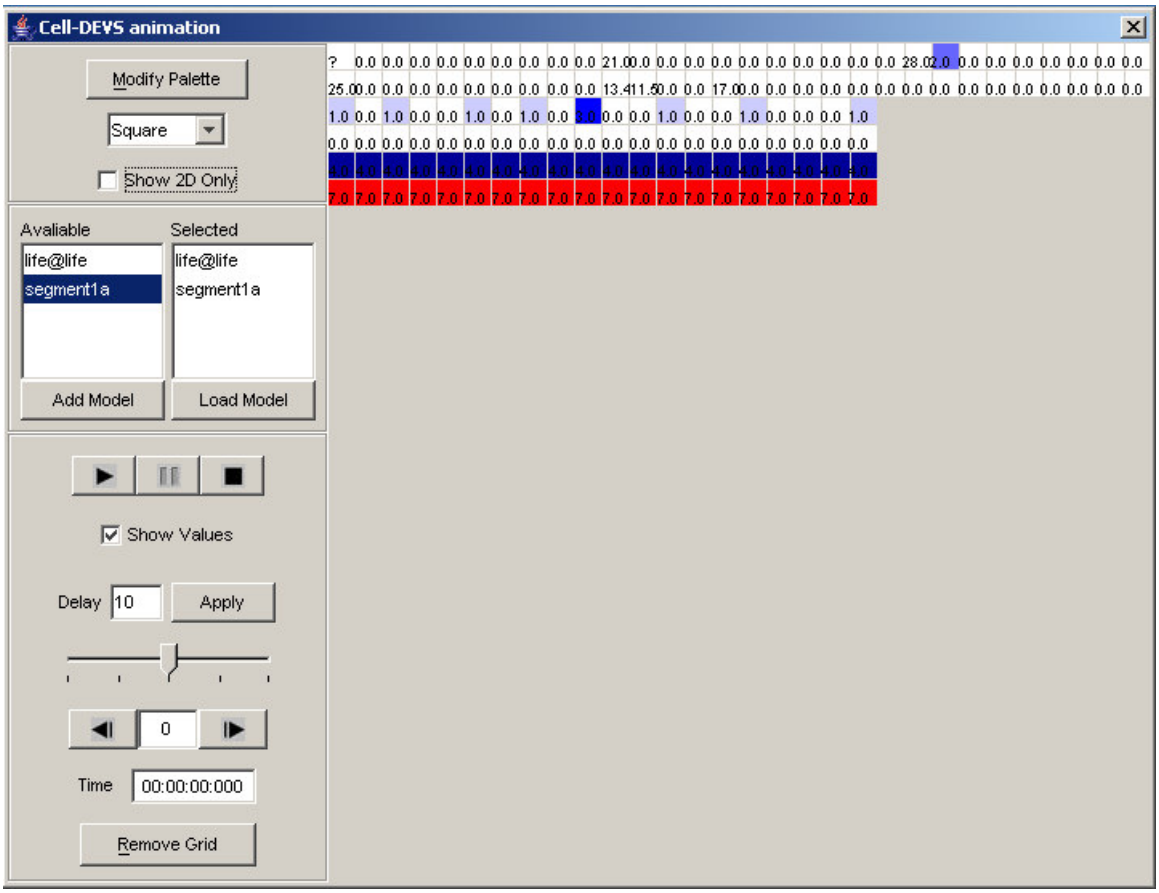


Figure 131 Show 3D Model with 3D mode