CD++

(Parallel version)

User's Guide

Gabriel A. Wainer

Dept. of Systems and Computer Engineering
Carleton University
Ottawa, Canada

Alejandro Troccoli

Daniel A. Rodriguez, Amir Barylko, Jorge Beyoglonian

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Argentina

2001

Contents

CD++

CD++ is a tool for the simulation of Parallel DEVS and Parallel Cell-DEVS models. It runs either in standalone (1 machine) or in parallel mode over a network of machines. This is CD++ User's Guide. A complete understanding of the Parallel DEVS and Cell-DEVS models is assumed. Please, refer to the CD++ scientific report if necessary.

1 Installation

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.

1.1 System requirements

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

Compiling for UNIX / Linux

- 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

Compiling for Windows

To compile CD++ in Windows the CYGWIN tools are required.

- Cygwin: latest version available from http:\\www.cygwin.com. When downloading Cygwin, select the packages that are listed in *Compiling for UNIX / Linux*. You will need to get makedepend also (it is not included in the standard Cygwin distribution)

1.2 MPI

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
   ./configure -prefix= /usr/local/mpich-1.2.0. (or your preferred location)

4. Compile mpich by running
   make >& make.log

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

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

6. (Optional) Install mpich on a shared location

   make install

Troubleshooting

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.

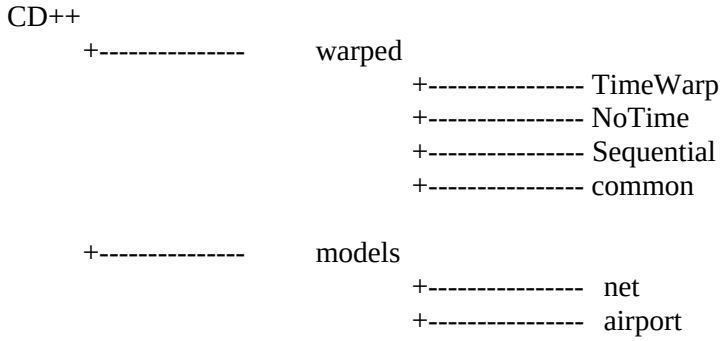
1.3 CD++

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



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 1: Makefile.common – Setting the source location

2. 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 2: Makefile.common – Choosing the Warped kernel

3. 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 3: 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
```

2 Starting the simulator

Previous versions of CD++ provided two different startup modes: a server mode and a workstation mode. When running in server mode, the program is started and opens a TCP port through which it will receive a model's specification. Instead, when the workstation startup is chosen, all settings are read from files specified in the command line options.

CD++ currently supports the workstation mode only. The server mode option is being developed.

2.1 Workstation Mode

To run CD++, type

./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 4: 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 5.3.

–**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 6.2.

–**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 :	Initialisation 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 5.4.

–**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 6.1.

–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 6.4.

–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 6.5.

–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.

3 Model definition

The simulator requires a model to run. A model is defined using a file (usally a .ma file), which is a plain text file which details the model components. This section will explain how the structure of such .ma file.

3.1 Structure of .ma file

A model file is used to define coupled and Cell-DEVS models. Atomic models are added to the tool at compile time, and if new atomic models need to be defined, they must be code as detailed in section 4. A model file consists of a set of groups and definition clauses within the groups. A group is identified by writing its name between square brackets. All lines following a group declaration are taken to be parameters for that group and are of the form

Id : value

As an example, mygroup is defined below:

```
[mygroup]
mygroupparameter : value
mygroupparameter2 : value
```

Figure 5: Defining groups and group parameters

All model files must have a **top** group identifying the top level coupled model. A small model example will be now shown, but Section 8 defines more complex models.

3.1.1 Coupled Models

A coupled model is defined in a group that has the model's name. For a couple model, four different parameters exist:

Components:

components : model_name1[@atomicclass1] [model_name2[@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 6 : Example for the definition of a DEVS coupled model

3.1.2 Atomic models

As it was mentioned before, 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_name n : value n
```

Figure 7: 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 8: Example of setting parameters to DEVS atomic models

3.1.3 Cell DEVS models

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 an unidimensional 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 $\Re \cup \{?\}$. 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 5.1 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 5.2 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 9.3 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.

4 Coding new atomic models

This section will describe how to code new atomic models into CD++. Knowledge of C++ is required. Users not intending to code new models can skip this section.

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.

4.1 Defining the state of a model

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 9: The AtomicState class.

To access the current state the 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.

4.2

When creating a new atomic model, a new class derived from atomic has to be created. Atomic is an abstract class that declares a model's API and defines some service functions the user can use to

```
class Atomic : public Model
{
public:
                        // Destructor


protected:


  virtual Model &initFunction() = 0;
  virtual Model &externalFunction ( const
  virtual Model &externalFunction( const ExternalMessage & );
  virtual Model &internalFunction( const InternalMessage & ) = 0 ;

  virtual Model &confluentFunction ( const InternalMessage &, const MessageBag & );

  virtual string className() const

  //Kernel services

  Vtime nextChange();
  void lastChange(Vtime);


  Model &holdIn( const AtomicState::State &, const VTime & ) ;

  Model &sendOutput(const VTime &time, const Port & port , Value value)
  Model &passivate();




  virtual ModelState* getCurrentState() const;


  //State shortcuts
  Model &state( const AtomicState::State &s )


  const AtomicState::State &state() const
  {return ((AtomicState *)getCurren

};       // class Atomic
```

10: The Atomic Class

services are functions that allow the model to tell the simulator the current state and duration. These are:

holdIn(state, VTime)

state for a period of V It corresponds to the ta(s) function of the DEVS formalism.

passivate()

external event is received.

- **sendOutput(VTime, port, BasicMsgValue*):**

  Sends an output message through the port. The time should be set to the current time. The user can define any structure for the messages values, as described further on. The simulator will delete the pointer received.

- **sendOutput(VTime, port, Value):**

  This function is provided for backward compatibility. It send a real value through the given port. Again, the time should be set to the current time. If only real values will be used, then this function will do.

- **nextChange():**

  Returns the remaining time for the next internal transition (*sigma*).

- **lastChange():**

  Returns the time the model last changed, either because an external event was received or an internal transition took place.

- **state():**

  Returns the current model's phase.

- **getParameter(modelName, parameterName)**

  Returns the parameters the user defined in the .ma file. *ModelName* is the model's instance name, and *parameterName* is the name of the parameter to be returned. If the parameter has not been specified, an empty string is returned.

The new class should override the following functions:

- **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.

- **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 recommend 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.

```
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 11 : 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;

}
```

virtual string className()

4.3

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

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

```
class BasicMsgValue
{

  BasicMsgValue();
  virtual ~BasicMsgValue();
    rtual int valueSize() const;
  virtual string asString() const;


  BasicMsgValue(const BasicMsgValue& );

};


{
public:

  RealMsgValue( const Value& val);

  Valu
  int valueSize() const;
  string asString() const ;

  RealMsgValue(const RealMsgValue& );
};
```

12: The BasicMsgValue and RealMsgValue classes

•

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.

4.4 Example. A queue model.

A queue is a device of temporary storage that uses a FIFO (First In First Out) mechanism. 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 13 shows the structure of our model of a queue.

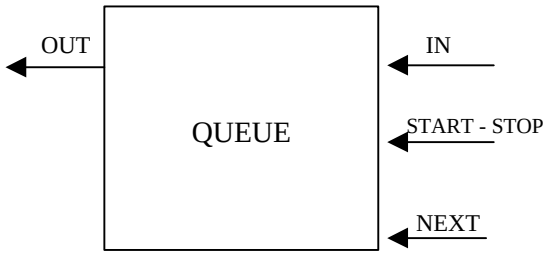


Figure 13: Structure of a Queue

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. 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 14 lists the Queue state class declaration and definition.

Once the state class has been defined, we are ready to implement the model itself. The Queue class declaration is shown in Figure 15.

```
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 14 : QueueState class

The *Queue* model overloads the initialization methods, internal function, external transition and output function. In addition, it 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 15: The Queue class declaration

The initFunction has to set the initial state for the queue, as shown in Figure 16. The elements of the list will be erased and the enabled will be set to true.

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

Figure 16: 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 17. 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 17 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 17: 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 18: 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.

Figure 19 lists the definition of the IPPacket class. The only restriction that needs to be placed in classes derived from BasicMsgValue is that they do not contain any pointers.

```
class IPPacket : public BasicMsgValue
{
public:

  char OriginIP[15];
  char DestinationID[15];
  int  Port;
  int  SequenceNumber;
  int  PayloadSize;

  IPPacket();
  virtual ~IPPacket();

  virtual int valueSize() const
  { return sizeof( IPPacket ); }

  virtual string asString() const;
  virtual BasicMsgValue* clone() const;

  IPPacket(const IPPacket& );

};
```

Figure 19: IP Packet Definition

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 20.

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

Figure 20 : 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 21 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 21 : 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 22.

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

Figure 22 : .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 23 : 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 24 : 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 25 : 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 26 : 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 27 : 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 28 : 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 29 : 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 31 shows a fragment of a partition debug file generated when running the model described in Figure 27 with the partition shown next.

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

Figure 30 : Partition for the heat diffusion model of Figure 27

```
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 31 : 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 32 shows the output generated for the *Game Life* model as implemented in section 9.1.

```
********* 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 32 : 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 33 shows a fragment of the output generated for the Game of the Life model of section 9.1 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 33 : Fragment of the output generated by the debug mode for the Evaluation or Rules

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 34 : 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 35 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 35 : 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 36 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 36 : 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 37 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 37 : 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 38 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 38 : Fragment of the output generated for a model with dimension 4

7.2 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 39 : 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 40 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 40 : Parlog output for a 4 machines partition.

7.3 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]

-bn 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 41 : 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 42 : Running pcd with logbuffer.

7.4 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 43 : 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.

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

The tool *ToMap* allows to creates a .map (section 5.2) file from a .val file (section 5.1).

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 44 : 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.

8 APENDIX A - Local transition functions for cellular 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.

8.1 A grammar for writing the rules

The BNF for the grammar used for the rules is shown in Figure 45. 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_TUPLA

REST_TUPLA     = , INT REST_TUPLA
                |  )

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 45: 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.

8.2 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	**AND OR XOR IMP EQV**	Left
	NOT	Right
	= != > < >= <=	
	+ -	Left
	* /	Left
	FUNCTION	
Higher Precedence	**REAL INT BOOL** COUNT **?** **STRING** CONSTFUNC	
	( )	

Figure 1 – Precedence Order and Associativity used in CD++

8.3 Functions and Constants allowed by the language

8.3.1 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.*

8.3.1.1 Boolean Operators

8.3.1.1.1 *Operator AND*

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

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

Figure 46: operator AND truthtable

8.3.1.1.2 Operator *OR*

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

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

Figure 47: Operator OR truthtable

8.3.1.1.3 Operator *NOT*

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

NOT	
T	F
F	T
?	?

Figure 48: Behavior of the boolean operator NOT

8.3.1.1.4 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 49: Operator XOR truthtable

8.3.1.1.5 Operator *IMP*

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

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

Figure 50: Operator IMP truthtable

8.3.1.1.6 Operator *EQV*

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

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

Figure 51: Operator EQV truthtable

8.3.2 Functions and Operations on Real Numbers

8.3.2.1 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.

8.3.2.1.1 Operator =

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

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

Figure 52: Behavior of the Relational Operator =

8.3.2.1.2 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 53: Behavior of the Relational Operator !=

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

8.3.2.1.3 Operator >

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

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

Figure 54 : Behavior of the Relational Operator >

8.3.2.1.4 Operator <

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

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

Figure 55 : Behavior of the Relational Operator <

8.3.2.1.5 Operator <=

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

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

Figure 56 : Behavior of the Relational Operator <=

8.3.2.1.6 Operator >=

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

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

Figure 57: Behavior of the Relational Operator >=

8.3.2.2 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.

Figure 58: Arithmetic Operators

Division by zero will result to the undefined value.

8.3.2.3 Functions on Real Numbers

8.3.2.3.1 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

8.3.2.3.2 Mathematical Functions

This section describes commonly used mathematical functions.

8.3.2.3.2.1 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 **?**.

8.3.2.3.2.2 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) ∀*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*) ∀*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) ∀*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*) ∀*x*
logn(*x*, 10) is equivalent to **log**(*x*) ∀*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*) ∀*x*

8.3.2.3.2.3 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, ?) = ?

8.3.2.3.3 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

8.3.2.3.4 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 8.3.2.3.8.

8.3.2.3.5 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 8.3.2.3.1.

Function nextPrime

Signature: **nextPrime** : Real $r \rightarrow$ *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 $n \rightarrow$ *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.

8.3.2.3.6 Functions to calculate Minimum and Maximums

Function min

Signature: **min** : *Real a* x *Real b* $\rightarrow$ *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* $\rightarrow$ *Real*
Description: Returns the maximum between *a* and *b*.
If *a* or *b* are undefined then returns **?**.

8.3.2.3.7 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* $\rightarrow$ *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*)

8.3.2.3.8 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* m x *Real* s → *Real*

Description: Returns a random real number with Normal distribution (m, s), where m is the average, and s is the standard error.
If m or s are undefined, or s 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*))

8.3.2.3.9 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 $\left(\frac{a}{b}\right)$

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.

8.3.2.4 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.

8.3.2.5 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.

8.3.2.6 Functions to Convert Values between different units

8.3.2.6.1 Functions to Convert Degrees to Radians

Function radToDeg

Signature: **radToDeg** : Real *r* → *Real*

Description: Converts the value *r* from radians to degrees.
If *r* is undefined then returns **?**.

Function degToRad

Signature: **degToRad** : Real *r* → *Real*

Description: Converts the value *r* from degrees to radians.
If *r* is undefined then returns **?**.

8.3.2.6.2 Functions to Convert Rectangular to Polar Coordinates

Function rectToPolar_r

Signature: **rectToPolar_r** : Real *x* x Real *y* → *Real*

Description: Converts the Cartesian coodinate (*x*, *y*) to the polar form (*r*, *q*), and returns *r*.
If *x* or *y* are undefined then return **?**.

Function rectToPolar_angle

Signature: **rectToPolar_angle** : Real *x* x Real *y* → *Real*

Description: Converts the Cartesian coordinate (*x*, *y*) to the polar form (*r*, *q*), and returns *q*.
If *x* or *y* are undefined then return **?**.

Function polarToRect_x

Signature: **polarToRect_x** : Real *r* x Real *q* → *Real*
Description: Converts the polar coordinate (*r*, *q*) to the Cartesian form (*x*, *y*), and returns *x*.
If *r* or *q* are undefined, or *r* is negative, then returns **?**.

Function polarToRect_y

Signature: **polarToRect_y** : Real *r* x Real *q* → *Real*
Description: Converts the polar coordinate (*r*, *q*) to the Cartesian form (*x*, *y*), and returns *y*.
If *r* or *q* are undefined, or *r* is negative, then returns **?**.

8.3.2.6.3 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 **?**.

8.3.2.7 Functions to manipulate the Values on the Input and Output Ports

Function portValue

Signature: **portValue** : String $p \rightarrow$ *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 section 9.3) . 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 59 : 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 60 : Example of use of the function portValue with thisPort

Section 9.3 shows an example where the *portInTransition* clause is used.

Function send

Signature: **send** : String *p* x Real $x \rightarrow 0$

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.

8.3.3 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 $1.6726231 \times 10^{-24}$ grams.

Constant neutron_mass

Returns the mass of a neutron, which is $1.6749286 \times 10^{-24}$ 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.

8.4 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.

8.4.1 Clause *Else*

When the clause **portInTransition** is used (see section 9.3), it is possible to use the clause **else** to give an alternative rule in case that none of the rules evaluates to true.

Figure 61 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 61 : 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 62 : A circular reference produced by a bad use of the clause Else

CD++ will detect the special case shown in Figure 63, where the *else* clause references the same function being defined.

```
[another_rule]
rule: ...
rule: ...
else: another_rule
```

Figure 63 : Example of a circular reference detected by the simulator

8.4.2 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 64 : 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 65 : A .ma file with comments

Section 9.5 shows a model where macros are used.

For special considerations regarding files created by the preprocessor, please see *Appendix B.*

9 Appendix B – Examples

9.1 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 66 : Implementation of the Game of Life

9.2 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     0000000000000000010
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 67: Implementation of the Rebound of an Object

9.3 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.

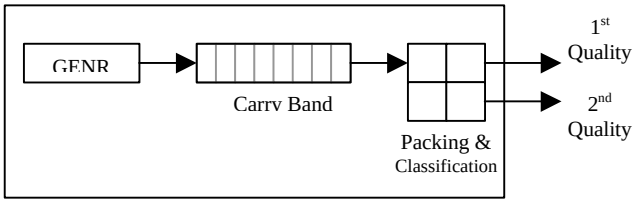


Figure 68: Coupling structure for the Classification of Substances

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 69: 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.

9.4 Life Game – 3D

The next example is an adaptation of the *Game of the Life* to a three dimensional space.

Figure 70 shows the model definition and Figure 71 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 70: Implementation of the Game of Life – 3D

```
(0,0,0) = 1
(0,0,2) = 1
(1,0,0) = 1
(1,0,1) = 1
(1,1,1) = 1
(1,2,0) = 1
(1,2,2) = 1
(1,3,2) = 1
(1,4,2) = 1
(1,5,0) = 1
(1,5,1) = 1
(1,6,0) = 1
(1,6,1) = 1
(2,1,2) = 1
(2,1,0) = 1
(2,3,1) = 1
(2,3,2) = 1
```

```
(2,4,1) = 1
(2,4,2) = 1
(2,5,0) = 1
(2,6,1) = 1
(3,2,1) = 1
(3,5,1) = 1
(3,5,2) = 1
(3,6,1) = 1
(3,6,2) = 1
(4,1,2) = 1
(4,2,0) = 1
(4,2,1) = 1
(4,4,1) = 1
(4,5,0) = 1
(4,5,2) = 1
(4,6,0) = 1
(4,6,2) = 1
```

```
(5,1,2) = 1
(5,2,0) = 1
(5,2,2) = 1
(5,3,0) = 1
(5,3,1) = 1
(5,5,1) = 1
(5,5,2) = 1
(5,6,0) = 1
(6,0,0) = 1
(6,1,1) = 1
(6,1,2) = 1
(6,3,0) = 1
(6,3,2) = 1
(6,4,2) = 1
(6,5,1) = 1
(6,6,0) = 1
(6,6,2) = 1
```

Figure 71: Initial values for the cells of the Game of Life – 3D

9.5 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 72: 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 73: 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 74: 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 75: File life–1.inc that contains the remaining macros for the Game of Life 4D

10 Appendix C – 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.