CD++

User's Guide

Daniel A. Rodríguez
Gabriel A. Wainer

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Argentina

1999

Contents

1 Invoking to the Simulator 8
1.1 Standalone Mode 8
1.2 Simulation Server 10
2 Definition of the Models 11
2.1 Coupled Models 11
2.2 Atomic Models 12
2.3 Cellular Models 13
3 Incorporating New Atomic Models 17
3.1 Example. Construction of a Queue 18
4 Rules Specification Language 20
4.1 Language's Grammar 20
4.2 Precedence Order and Associativity of Operators 22
4.3 Functions and Constants used by the language 23
4.3.1 Use of Boolean Values 23
4.3.1.1 Boolean Constant of the Trivalent Logic 23
4.3.1.2 Boolean Operators 23
4.3.1.2.1 Operator AND 23
4.3.1.2.2 Operator *OR* 23
4.3.1.2.3 Operator *NOT* 24
4.3.1.2.4 Operator *XOR* 24
4.3.1.2.5 Operator *IMP* 24
4.3.1.2.6 Operator *EQV* 24
4.3.2 Functions and Operations on Real Numbers 25
4.3.2.1 Relational Operators 25
4.3.2.1.1 Operator = ... 25
4.3.2.1.2 Operator != 25
4.3.2.1.3 Operator > 25
4.3.2.1.4 Operator < 25
4.3.2.1.5 Operator <= 26
4.3.2.1.6 Operator >= 26
4.3.2.2 Arithmetic Operators 26
4.3.2.3 Functions on Real Numbers 27
4.3.2.3.1 Functions to Verify Properties of Real Numbers 27
Function Even 27
Function Odd 27
Function isInt 27
Function isPrime 27
Function isUndefined 27
4.3.2.3.2 Mathematical Functions 28
4.3.2.3.2.1 Trigonometric Functions 28
Function tan 28
Function sin 28
Function cos 28
Function sec 28
Function cotan 28
Function cosec 28

Function atan 29
Function asin 29
Function acos 29
Function asec 29
Function acotan 29
Function sinh 29
Function cosh 29
Function tanh 29
Function sech 30
Function cosech 30
Function atanh 30
Function asinh 30
Function acosh 30
Function asech 30
Function acosech 30
Function acotanh 30
Function hip 31
4.3.2.3.2.2 Functions to calculate Roots, Powers and Logarithms. 31
Function sqrt 31
Function exp 31
Function ln 31
Function log 31
Function logn 32
Function power 32
Function root 32
4.3.2.3.2.3 Functions to calculate GCD, LCM and the Rest of the Numeric Division 32
Function LCM 32
Function GCD 32
Function remainder 32
4.3.2.3.3 Functions to Convert Real Values to Integers Values 33
Function round 33
Function trunc 33
Function truncUpper 33
Function fractional 33
4.3.2.3.4 Functions to manipulate the Sign of numerical values 34
Function abs 34
Function sign 34
Function randomSign 34
4.3.2.3.5 Functions to manipulate Prime numbers 34
Function isPrime 34
Function nextPrime 34
Function nth_Prime 34
4.3.2.3.6 Functions to calculate Minimum and Maximums 35
Function min 35
Function max 35
4.3.2.3.7 Conditional Functions 35
Function if 35
Function ifu 35
4.3.2.3.8 Probabilistic Functions 36

Function randomSign....36
Function random....36
Function chi....36
Function beta....36
Function exponential....36
Function f....36
Function gamma....36
Function normal....36
Function uniform....37
Function binomial....37
Function poisson....37
Function randInt....37
4.3.2.3.9 Functions to calculate Factorials and Combinatories....37
Function fact....37
Function comb....37
4.3.2.4 Functions for the Cells and his Neighborhood....38
Function stateCount....38
Function trueCount....38
Function falseCount....38
Function undefCount....38
Function cellPos....38
4.3.2.5 Functions to Get the Simulation Time....39
Function Time....39
4.3.2.6 Functions to Convert Values between different units....39
4.3.2.6.1 Functions to Convert Degrees to Radians....39
Function radToDeg....39
Function degToRad....39
4.3.2.6.2 Functions to Convert Rectangular to Polar Coordinates....39
Function rectToPolar_r....39
Function rectToPolar_angle....39
Function polarToRect_x....39
Function polarToRect_y....39
4.3.2.6.3 Functions to Covert Temperatures between different units....40
Function CtoF....40
Function CtoK....40
Function KtoC....40
Function KtoF....40
Function FtoC....40
Function FtoK....40
4.3.2.7 Functions to manipulate the Values on the Input and Output Ports....40
Function portValue....40
Function send....41
4.3.3 Predefined Constants....42
Constant Pi....42
Constant e....42
Constant INF....42
Constant electron_mass....43
Constant proton_mass....43
Constant neutron_mass....43

Constant Catalan 43
Constant Rydberg 43
Constant grav 43
Constant bohr_radius 43
Constant bohr_magneton 43
Constant Boltzmann 43
Constant accel 43
Constant light 43
Constant electron_charge 43
Constant Planck 43
Constant Avogadro 43
Constant amu 44
Constant pem 44
Constant ideal_gas 44
Constant Faraday 44
Constant Stefan_boltzmann 44
Constant golden 44
Constant euler_gamma 44
4.4 Techniques to Avoid the Rewriting of Rules 44
4.4.1 Clause *Else* 44
4.4.2 Preprocessor – Using Macros 45
5 File for the Definition of the Initial Values of the Model 47
6 File of Map of Initial Values 48
7 File for the definition of External Events 49
8 Format of the Events generated as output 49
9 Format of the Log File 50
10 Output generated by the Parser Debug Mode 50
11 Output of the Debug Mode for the Evaluation of Rules 51
12 Viewing the Results – *DrawLog* 53
12.1 Representing bidimensional cellular models with DrawLog 55
12.2 Representing three–dimensional cellular models with DrawLog 55
12.3 Representing cellular models with 4 or more dimensions 56
13 Random Initial States – *MakeRand* 57
14 Converting .VAL files to Map of Values – *ToMap* 58
15 Converting .VAL files to use with *CD++* – *ToCDPP* 59
16 Appendix A – Examples 60
16.1 Game of Life 60
16.2 Simulation of the Rebound of an Object 61
16.3 Classification of Substances 62
16.4 Game of Life – 3D 64
16.5 Use of Macros 65
17 Appendix B – The Preprocessor and the Temporary Files 67

Index of Figures

Figure 1 – Help showed by the simulator 8
Figure 2 – Example for the definition of a DEVS coupled model 12
Figure 3 – Setting values to a DEVS atomic model 12
Figure 4 – Example of setting parameters to DEVS atomic models 13
Figure 5 – Structure of a Queue 18
Figure 6 – Method to initialize the *Queue* 18
Figure 7 – Method for the External Transition Function of the *Queue* 19
Figure 8 – Methods for the Output Function and the Internal Transition of the *Queue* 19
Figure 9 – Grammar used for the definition of the rules on *CD++* 22
Figure 10 – Precedence Order and Associativity used in N–CD++ 23
Figure 11 – Behavior of the boolean operator *AND* 23
Figure 12 - Behavior of the boolean operator *OR* 23
Figure 13 – Behavior of the boolean operator *NOT* 24
Figure 14 – Behavior of the boolean operator *XOR* 24
Figure 15 – Behavior of the boolean operator *IMP* 24
Figure 16 – Behavior of the boolean operator *EQV* 24
Figure 17 – Behavior of the Relational Operator = ... 25
Figure 18 – Behavior of the Relational Operator != 25
Figure 19 – Behavior of the Relational Operator > 25
Figure 20 – Behavior of the Relational Operator < 26
Figure 21 – Behavior of the Relational Operator <= 26
Figure 22 – Behavior of the Relational Operator >= 26
Figure 23 – Arithmetic Operators 26
Figure 24 – Example of use of the function *portValue* 41
Figure 25 – Example of use of the function portValue with thisPort 41
Figure 26 – Example of use of the clause *Else* 45
Figure 27 – Example of a circular reference produced by a bad use of the clause *Else* 45
Figure 28 – Example of a circular reference detected by the simulator 45
Figure 29 – Format used to define a Macro 46
Figure 30 – Example of using Comments 47
Figure 31 – Format of the file used to define the initial values of a cellular model 47
Figure 32 – Example of a file for the definition of the initial values for a Cellular Model 48
Figure 33 – Format of the file of Map of values for a Cellular Model 48
Figure 34 – Example of a file for the definition of the External Events 49
Figure 35 – Example of an Output file 49
Figure 36 – Fragment of a Log File 50
Figure 37 – Output generated in the Parser Debug Mode for the *Game of Life* 51
Figure 38 – Fragment of the output generated by the debug mode for the Evaluation or Rules 53
Figure 39 – Help shown by *DrawLog* 53
Figure 40 – Examples for the invocation to *DrawLog* 54
Figure 41 – Fragment of the output generated for a bidimensional cellular model 55
Figure 42 – Fragment of the output generated for a three–dimensional cellular model 56
Figure 43 – Fragment of the output generated for a model with dimension 4 57
Figure 44 – Help shown by *MakeRand* 57
Figure 45 – Help shown by *ToMap* 59
Figure 46 – Help shown by *ToCDPP* 59

Figure 47 – Implementation of the Game of Life 61
Figure 48 – Implementation of the Rebound of an Object 62
Figure 49 – Coupling structure for the Classification of Substances 63
Figure 50 – Implementation of the Model to Classify Substances 64
Figure 51 – Implementation of the Game of Life – 3D 65
Figure 52 – Initial values for the cells of the Game of Life – 3D 65
Figure 53 – Implementation of the *Game of Life* with 4 dimensions and using macros 66
Figure 54 – File *life.val* that contains the initial values for the *Game of Life* in 4D 66
Figure 55 – File *life.inc* that contains some macros used in the *Game of Life* 4D 67
Figure 56 – File *life–1.inc* that contains the remaining macro for the *Game of Life* 4D 67

CD++

User's Guide

1 Invoking to the Simulator

It exists two forms to invoke to the simulator:

- *Standalone Mode.*
- *Simulation Server* (using network connection).

1.1 Standalone Mode

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

–h: shows this help:

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

Figure 1 – Help showed by the simulator

–**e**: External events filename. If this parameter is omitted, the simulator will not use external events.
The format used to describe the external events is showed in the section 6.

–**l**: Log filename. This file is used to store the messages received and emitted by each model within the simulation. If this parameter is omitted, the simulator will not generate activity log. If you wish to get the log on standard output, you should write **–l**).
The format used by the log is described in the section 9.

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

–**o**: output filename. This parameter indicates the name of the file that will be used to store the output generated by the simulator. If this parameter is omitted, the simulator will not generate any output. If you wish to get the results on standard output, simply write **–o**.
The format of this output is showed in the section 8.

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

HH: hours

MM: minutes (0 to 59)

SS: seconds (0 to 59)

MS: thousandth of second (0 to 999)

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

–p: Shows additional information on parsing the cell model's rules. The parameter must be accompanied with the filename that will be used to store the detail. This mode is useful when a syntax error occurs on complex rules.
The format used to store the output is showed in the section 10.

–v: Enable the debug mode on the evaluation of all cell model's rules. For each rule to be evaluated it will be showed the results of the evaluation of each function and operator that they compose it. In addition, this mode evaluates the rules in complete form, that is, it doesn't use the rule's optimization. The parameter must be accompanied with the filename that will be used to store the rule's evaluation.
The format of the output generated when this mode is enabled is showed in the section 11.

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

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

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

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

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

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

1.2 Simulation Server

The invocation of the simulation without parameters indicates that it must run in simulation server mode. In this implementation, the communication with the server will use the TCP/IP services and will be only available under any version of Unix. The simulator will wait on a port for the specification, it will simulate it, and it will return the results through the same port.

The specification is composed by three parts separated by a line with a point at the first position. The order for the specification is:

Description of the model.

List of external events.

Maximum simulation time.

2 Definition of the Models

The file that allows to define the model is composed by groups of definitions for the coupled models and, optionally, configuration of atomic models. Each definition indicates the name of the model (between [ ]) and its attributes. The group **[top]** is obligatory and defines the coupled model at the top level.

In the section 16, there are some examples that show how the models are defined.

2.1 Coupled Models

For this models exists four properties to configure: components (using the clause "*components*"), output ports (clause "*out*"), input ports (clause "*in*") and links between models (clause "*link"*). The syntax is:

Components: Describe the models that compound the coupled model. If this clause is not specified, an error will occur. The syntax is:

model_name@class_nombre

The order in which the models are specified establish the priority used to send the messages. This represents the **select** function of the formalism.

The model's name is necessary because is possible to construct a coupled model with more than one instance of the same atomic model. For example, a coupled model that has two queues called *queue1* and *queue2*.

The class' name can reference to either atomic or coupled models. These last ones should be defined in the same configuration file as a new group.

Out: Enumerate the name of the output ports. This clause is optional because a model cannot have ports of this kind.
Example: *Out* port1 port2 port3

In: Enumerate the name of the input ports. This clause is optional because a model cannot have ports of this kind.
Example: *In* port1 port2 port3

Link: Describe the internal and external coupled schema. The syntax is:

source_port[*@model*] *destination_port*[*@model*]

The name of the model is optional since if it is not indicated the coupled model being defined will be used.

Example:

```
[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 2 – Example for the definition of a DEVS coupled model

2.2 Atomic Models

If the configuration for the atomic models is not specified, the default values assumed by the class's developer will be used (see section 13).
The configuration is specified as showed in the next figure:

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

Figure 3 – Setting values to a DEVS atomic model

The name of the variables is defined by the class's developer and must be documented together with the source code.
Each instance of an atomic model can be configured independently of another instances of the same kind.

In the next example two instances of the class *Processor* (derived from *Atomic*) with different configuration is showed:

```
[top]
```

```
components : Queue@queue Processor1@processor Processor2@processor
.
.


[processor]
distribution : exponential


[processor2]
distribution : poisson


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

4 – Example of setting parameters to DEVS atomic models

Cellular Models

is used with the aggregate of certain inherent parameter characteristics of them. These parameters are:

Type : CELL |]

Indicate if the cellular model is flat or not. If it is not specified, it will be assumed that *CELL*).

:

It allows to define the quantity of columns for unidimensional and bidimensional cellular models.

Width clause necessarily implies the use of the clause to complete the definition of the dimension.

Width is used, the invocation of the clause in the same model's definition it will produce an error.

:

It allows to define the quantity of rows only for bidimensional cellular models.

If you wish to define an unidimensional cellular model, you must assign the value 1 to *Height* clause.

Height clause necessarily implies the use of the clause to complete the definition of the dimension.

Height is used, the invocation of the clause in the same model's definition it will produce an error.

Dim : (x_0, x_1, ..., x_n)

It allows to define the dimension of any cellular model.
All the x_i values must be integers.
If the clause *Dim* is used, the invocation of the clauses *Width* or *Height* in the same model's definition it will produce an error.
The tupla that defines the dimension of the cellular model must have two or more elements. This imply that if you wish to create an unidimensional cellular model, you must define a dimension (x_0, 1).

All the references to cells will have the format:

(y_0, y_1, ..., y_n) where: $0 \le y_i < x_i$ $\forall$ i = 0, .., n
with y_i an integer value

Select : cellName ($x_{1,1}$, $x_{2,1}$,...,$x_{n,1}$)... cellName ($x_{1,m}$, $y_{2,m}$,...,$k_{n.m}$)

With: $0 \le x_{1,i} < dim_1 \vee 0 \le x_{1,i} < Width$ $\forall$ i = 1, ..., m
$0 \le x_{2,i} < dim_2 \vee 0 \le x_{2,i} < Height$ $\forall$ i = 1, ..., m
$0 \le x_{k,i} < dim_k$ $\forall$ i = 1, ..., m ; $\forall$ k = 3, ..., n

It represents the function ***select*** described in the formalism, which indicates the cells that have priority on the rest. The not specified cells possess the priority dictated by the order of pairs according to their position.

In : The same as in the coupled models. This clause can be not defined, since a cell cannot have input ports connected with external models.

Out : The same as in the coupled models. This clause can be not defined, since a cell cannot have output ports connected with external models.

Link : The same as in the coupled models but to make reference to a cell it should be used the name of the coupled model together with (x_1,x_2,...,x_n) without leaving spaces.
Examples: *Link outputPort inputPort@*cellName (x_1,x_2,...,x_n)
*Link outputPort@*cellName (x_1,x_2,...,x_n) inputPort

Border : [**WRAPPED** | **NOWRAPPED**]

It indicates if the model is toroidal or not. By default, it takes the value NOWRAPPED.
If a non-toroidal border is used, a reference to a cell outside the cellular space will return the undefined value (**?**).

Delay : **TRASPORT** |]

It specifies the delay type used in every cell of the model. By default, the value

DefaultDelayTime integer

Defines the delay used by default for the external events (measured in milliseconds).

cellName (x , x ,...,$x_{n,1}$ $x_{1,m}$ $x_{2,m}$ $_{n.m}$)

$x_{1,i}$ $x_{2,i}$ $_{n.i}$)

CD++ does not impose restrictions on the neighborhood's creation, allowing that the

It is possible to use more than a sentence **neighbors**
cellular model.

Initialvalue : *Real* |]

It represents the initial value for the cell's space. The symbol represents the undefined value.

row$_i$ $_1$...value

With 0 ≤ $_i$ < (where *Height*
with **Dim** **Height**).

bidimensional cellular model. The value defined in the position *j*
establish the initial state of the cell (*i j*) of the cellular model.

the set {**?**
If this clause is used for the description of a model with more than 2 dimensions, an error will occur.

16.1.

row$_i$ $_1$ ... value

With 0 ≤ $_i$ < (where *Height*
with **Dim** **Height**).

bidimensional cellular model. The value defined in the position *j*
establish the initial state of the cell (*i j*) of the cellular model.

The values are indicated separated by a blank space. Contrary to **InitialRowValue**, by means of this clause it is possible to use any value belonging to the set $\Re \cup \{?\}$.
If this clause is used for the description of a model with more than 2 dimensions, an error will occur.

InitialCellsValue : *fileName*

It specifies the name of the file that contains the initial values for the cells of a cellular model. The format of this file is defined in the section 5.
InitialCellsValue can be used with any type of cellular models, even with bidimensional models. On the other hand *InitialRowValue* and *InitialRow* will not be able to be used when the dimension of the model is greater than 2. If the dimension is 2, anyone of them can be used, and even a combination of them, but in this case the read values of the file specified in *InitialCellsValue* will replace to the values of the same cells defined by *InitialRowValue* or *InitialRow*.

InitialMapValue : *fileName*

It specifies the name of the file that it contains a map of values that will be used as initial state for a cellular model. The format of this file is defined in the section 6.

LocalTransition : *transitionFunctionName*

It indicates the name of the group that contains the rules that define the local computing function for all the cells.

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

It allows to define an alternative behavior when an external message arrives to the specified input port of the cell (x_1, x_2,...,x_n) of the cellular model.
If this clause is not used for a cell that has an input port, when arriving an external message through this port, the value of this message will be assigned to the cell using the delay specified by defect in the definition of the model.
In the section 16.3 the use of this clause is exemplified.

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

It allows to define an alternative behavior for the group of cells understood inside the specified range. Each range is defined as (x_1,x_2,...,x_n) describing a unique cell, (x_1,x_2,...,x_n)..(y_1,y_2,y_n) describing an area of cells, or a list that can combine both of them (separating to each element with a blank space).
For example: *zone* : pothole { (10,10).. (13, 13) (1,3) }

In the moment to calculate the new state for a cell, if the cell is inside of any zone, the defined function it will be used for such, else the function defined with **LocalTransition** will be used.
In the section 16.2 is showed an example that use zones.

3 Incorporating New Atomic Models

This section describes the mechanism to define and to incorporate new atomic models to the tool. However, these models won't be able to be used to create a cellular coupled model, but it can be used to interact directly with other models or to be part of a DEVS coupled model. This chapter this guided to users with knowledge of programming in C++ language and their content cannot be useful for those people that it only interests to use the tool with the purpose of creating cellular coupled models and/or use models already defined within *CD++*.

To generate a new atomic model, it should be design a new class that is derived of the class *Atomic* and it should be added the new type of atomic model to the method *MainSimulator.registerNewAtomics*(). Then it should be overloaded the following methods:

- ***initFunction***: this method is invoked by the simulator at the beginning the simulation. The objective is to allow the initialization that the model considers necessary. Before invoking to the method, the value of *sigma* is infinite and the state is *passive*.

- ***externalFunction***: this method is invoked when an external event arrives from a port of the model.

- ***internalFunction***: before invoking to this method, the value of *sigma* is zero, since the interval has been completed for the internal transition.

- ***outputFunction***: before invoking to the method the value of *sigma* is zero, since the interval has been completed for the internal transition.

- ***className***: returns the name of the class.

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

- ***holdIn***(state, time): indicates to the simulator that the model should stay in the same state during a time, and after that it will generate an internal transition.

- ***passivate()***:indicates to the simulator that the model enters in passive mode and that it will only be reactivated when an external event arrives.

- ***sendOutput***(time, port, value): sends an output message through the port.

- ***nextChange***(): this method allows to obtain the remaining time for its next state change (*sigma*).

- ***lastChange***(): this method allows to obtain the time in that the last state change took place.

- ***state***(): this method obtain the actual phase of the model.

- ***getParameter***(modelName, parameterName): this method allows to access to the parameters that configure the class.

To initialize and use an atomic model see the section 2.2.

3.1 Example. Construction of a Queue

A queue is a device of temporary storage that uses a FIFO (First In First Out) mechanism. To implement it in *CD++* as a new class it should be created (that will call Queue) that extends to the class *Atomic*.

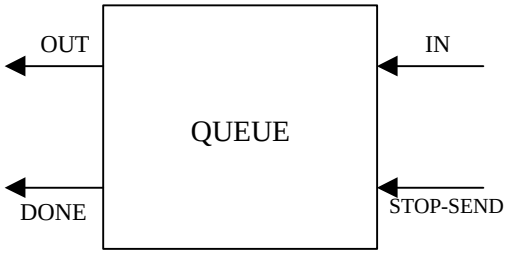

Figure 5 – Structure of a Queue

A queue should have an input port that allows to the rest of the models insert elements to be stored by the queue, and an output port to returns the stored values. The time of delay between the arrival of the element and their exit it is configurable by the user. To fulfil these requirements the queue defines two ports, a port *Done* that it indicates the reception of the element sent by the output port and a port regulator of flow called *stop-send*.

The *Queue* model overloads the initialization methods, internal function, external transition and output function. In the initialization, the variables of the model take the initial value and all the values of the queue are eliminated.

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

Figure 6 – Method to initialize the *Queue*

When an external event comes from an input port, the value is inserted in the internal queue and then it is verified if the state of the queue allows to be programmed to carry out a new shipment for the output port. If the message arrived for the port *Done* the last sent element can be eliminated of the internal queue and it prepares the next (if it exists). If the message comes from the port *Stop* the content it should be analyzed to interpret the order as "to stop" or "to continue" the expedition of data. If it stops, then registers the remaining time to conclude the iteration to be considered when renewing the tasks.

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

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

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

   return *this;
}
```

Figure 7 – Method for the External Transition Function of the *Queue*

The output function indicates that the time of preparation for the first element of the queue has concluded and this is sent by the port *Out*. Then the internal transition function is executed indicating that it has finished sending the value, therefore the model changes its phase to *passive*. The cycle will continue with the next external message.

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

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

Figure 8 – Methods for the Output Function and the Internal Transition of the *Queue*

4 Rules Specification Language

The definition of the rules that describe a certain behavior is made in independent form to the cellular models that use it. This allows that more than a cellular model uses the same specification and that several areas inside a cellular space use it without necessity of redefining it.

The language is defined as a new group inside the specification, where each component of the group is a rule with the following syntax:

rule : result delay { condition }

Each rule is composed of three elements: a *condition*, a *delay* and a *result*. To calculate the new value for the cell's state, the simulator takes each rule (in the order in that they were defined) and if the condition of it is evaluated to true, then its result and its delay are evaluated, and these values will be assigned to the cell. If the evaluation of the condition of the rule is false, then it takes the following rule. If all the rules are evaluated without having found some valid one, then the simulation will be aborted. If it exists more than a valid rule, it takes the first of them.

If when evaluating the delay it is obtained the undefined value, then the simulation will be automatically cancelled.

4.1 Language's Grammar

The syntax of the language used by *CD++* for the specification of the behavior of the atomic cellular models can be defined with the BNF shown in the Figure 4, where the words written with lowercase and in boldface represents terminals symbols, while those written in uppercase represent non terminals symbols.

```
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 9 – Grammar used for the definition of the rules on *CD++*

In the definition of a rule, the second value, that correspond to the delay of the cell, can be a real number, either in direct form or as a result of the evaluation of an expression. However, if it is not an integer number, it will be automatically truncated. On the other hand, if their value is undefined (**?**) then an error will occur, aborting the simulation.

4.2 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 habitual operations on real numbers; and *A*, *B* and *C* are real numbers. In this case, as * has high precedence that + then *B* * *A* will be first solved; therefore, it will be equivalent to solve *C* + (*B* * *A*).

The associativity indicates which function will be solved before two operations of same precedence. For example: the left associativity of the operators *AND* and *OR* indicate that if one interprets the line:

C **and** *B* **or** *D*

as *AND* and *OR* have the same precedence, it is appealed to the associativity to choose some of them. As they are associative to left, it is chosen to solve the *AND* first.

The operations that don't have associativity is because they cannot be combined in simultaneous form without using another operator of different precedence. For example, two real numbers don't have associativity, since it cannot be in the form *REAL REAL*, but rather it should have an operation that links them, as an arithmetic operator.

The table of precedence and associativities used for the language interpretation are shown in the following figure:

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 10 – Precedence Order and Associativity used in N–CD++

4.3 Functions and Constants used by the language

4.3.1 Use of Boolean Values

This section describes the constants that represent the boolean values of the trivalent logic used by *CD++* and show the operators applicable on it.

4.3.1.1 Boolean Constant of the 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*; this last one allows to represent a state whose value cannot be determined.

4.3.1.2 Boolean Operators

4.3.1.2.1 *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 11 – Behavior of the boolean operator *AND*

4.3.1.2.2 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 12 - Behavior of the boolean operator *OR*

4.3.1.2.3 Operator *NOT*

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

NOT	
T	F
F	T
?	?

Figure 13 – Behavior of the boolean operator *NOT*

4.3.1.2.4 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 14 – Behavior of the boolean operator *XOR*

4.3.1.2.5 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 15 – Behavior of the boolean operator *IMP*

4.3.1.2.6 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 16 – Behavior of the boolean operator *EQV*

4.3.2 Functions and Operations on Real Numbers

4.3.2.1 Relational Operators

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

Considering the definitions of the behavior of these operators, we can see that doesn't exists a total order on the elements which conform the real numbers, because in all the cases, the value **?** isn't comparable with any traditional real number.

4.3.2.1.1 Operator =

The operator **=** is used to test if two real numbers are equal. Its behavior is defined as follow:

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

Figure 17 – Behavior of the Relational Operator =

4.3.2.1.2 Operator !=

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

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

Figure 18 – Behavior of the Relational Operator !=

4.3.2.1.3 Operator >

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

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

Figure 19 – Behavior of the Relational Operator >

4.3.2.1.4 Operator <

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

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

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

Figure 20 – Behavior of the Relational Operator <

4.3.2.1.5 Operator <=

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

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

Figure 21 – Behavior of the Relational Operator <=

4.3.2.1.6 Operator >=

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

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

Figure 22 – Behavior of the Relational Operator >=

4.3.2.2 Arithmetic Operators

The language has operators to carry out the most usual operations on real numbers. If any of the operands has the undefined value, then the result of this operation will be undefined. This is also valid when any kind of function is used, and some of its parameters are undefined.

The operators used 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 23 – Arithmetic Operators

If a division by zero takes place, the undefined value will be returned.

4.3.2.3 Functions on Real Numbers

4.3.2.3.1 Functions to Verify Properties of Real Numbers

In this section the functions detailed allows to check if a real number has certain properties, as being an integer number, the undefined value, an even or odd number, or a prime number.

Function Even

Signature: **even** : *Real* → *Bool*
Description: Returns *True* if the value is integer and even. If the value is undefined returns *Undefined*. In another case 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 another case 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. In another 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. In another 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

4.3.2.3.2 Mathematical Functions

This section describes different kinds of functions used commonly in trigonometry, as well as for the calculation of roots, powers and logarithms. In addition, functions to obtain the rest and the module of the division of integer numbers are included.

4.3.2.3.2.1 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 **?**.

4.3.2.3.2.2 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*

4.3.2.3.2.3 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 rest 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 $\forall$ y $\neq$ **?**
remainder(x, 0) = x $\forall$ x
remainder(1.25, 0.3) = 0.05
remainder(1.25, 0.25) = 0
remainder(?, 3) = ?
remainder(5, ?) = ?

4.3.2.3.3 Functions to Convert Real Values to Integers Values

In this section, functions to convert real values to integers using the rounding and truncation techniques are detailed. In addition, it's showed functions to obtain the fractional part of a real value.

Function round

Signature: **round** : *Real a* $\rightarrow$ *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* $\rightarrow$ *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* $\rightarrow$ *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* $\rightarrow$ *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

4.3.2.3.4 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 4.3.2.3.8.

4.3.2.3.5 Functions to manipulate Prime numbers

Although the language allows the handling of prime numbers, all these instructions are very complex, and can increase the time of simulation considerably.

Function isPrime

See the section 4.3.2.3.1.

Function nextPrime

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

Function nth_Prime

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

4.3.2.3.6 Functions to calculate Minimum and Maximums

Function min

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

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

Function max

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

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

4.3.2.3.7 Conditional Functions

The functions described in this section allow returning certain real values depending on the evaluation of a specified logical condition.

Function if

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

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

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

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

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

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

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

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

Function ifu

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

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

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

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

4.3.2.3.8 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*
Descriptin: 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*))

4.3.2.3.9 Functions to calculate Factorials and Combinatories

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.

4.3.2.4 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 errorr 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.

4.3.2.5 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.

4.3.2.6 Functions to Convert Values between different units

4.3.2.6.1 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 **?**.

4.3.2.6.2 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 **?**.

4.3.2.6.3 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 **?**.

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

Function portValue

Signature: **portValue** : String *p* → *Real*
Description: Returns the last value arrived through the input port *p* of the cell that is evaluating. This function will only be able to be used when they are defined transition functions in the clause *PortInTransition* (see section 2.3) which allows to give behavior to the cell when a message arrives from an input port. If it is used in a function of non defined transition with *PortInTransition* an error will be generated in the interpretation of the rule.

If at the time to evaluate the function *portValue*, a message not arrived for the port *p* since the beginning of the simulation, the function will return the undefined value (**?**). Once a message has arrived, when being consulted the value for the port, the last input value will be returned.

When using the string "***thisPort***" as parameter of *portValue*, is possible to indicate to the simulator that the value of the port that is wanted is the value from the port for which the message arrived. For example:

Suppose that a cell has associate the input port *A*, and another cell has associate the port *B*. Then it is possible to define functions to calculate the value from the cell when arriving a message. In this case, we have:

```
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 24 – Example of use of the function *portValue*

In the example, was created a function for each port. The behavior of both functions is the same, but as the names of the ports are different, it is not possible to unify both functions. A possible solution is to make that the ports of the cells have the same names, for example *portN*, and to reference to the value of the port then use *portValue(portN)*. Another solution is to work with ***thisPort*** as is shown in the Figure 25.

```
PortInTransition: portA@cell(0,0)      functionA
PortInTransition: portB@cell(1,1)      functionA

[functionA]
rule: 10     100    { portValue(thisPort) > 10 }
rule: 0      100    { t }
```

Figure 25 – Example of use of the function portValue with thisPort

Thus, the behavior is unified, avoiding the rewriting of a function.

In the section 16.3, an example of use of the function *portValue* is showed to implement a model to classify substances.

Function send

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

Description: Sends the value *x* through the output port *p*.
If the cells have not associated the port p then an error will be produced and the simulation will be aborted.

Every time that a change takes place in a cell, *N–CD++* sends this value through the port *Out* of the cell. However, in certain the cases it is desirable to send certain value (that should not necessarily be the state of the cell) to some cell or DEVS model. For these cases, the function *send* is used.

It is recommended to use the function in the following way:

{ new_value + **send**(P, V) } delay {condition}

In that case, if the condition evaluates to true, then the new value of the cell will be the specified and the value *V* will be sent through the port *P*.

The function ***send*** always returns the value 0, because it was created with the idea of sending a value for a port without altering the value of the cell, as it is exemplified in the following case:

{ (0,0) + **send**(port1, 15 * log(10)) } 100 { (0,0) > 10 }

Note: **Send** is a function of the language, and then it can be used in any place where it is possible, for example in the definition of a *condition*. However, this is not desirable because a condition can be evaluated and the evaluation can not be *True*, and therefore the command *send* will be executed sending a value to the port. In this case, use the function *send* in the expressions that represent the new value of the cell or that defines the value of the delay, because they only will be evaluated when the condition is valid.

4.3.3 Predefined Constants

The language used by *N–CD++* allows to use predefined constants frequently used in the domains of the physics and the chemistry.

The constants can be see as functions that don't receive parameters and that always return the same real value.

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 x 10^{-28} grams.

Constant proton_mass
Returns the mass of a proton, which is 1.6726231 x 10^{-24} grams.

Constant neutron_mass
Returns the mass of a neutron, which is 1.6749286 x 10^{-24} grams.

Constant Catalan
Returns the Catalan's constant, which is defined as $\sum_{k=0}^{\infty}(-1)^k.(2^k+1)^{-2}$, that is approximately 0.9159655941772.

Constant Rydberg
Returns the Rydberg's constant, which is defined as 10.973.731,534 / m.

Constant grav
Returns the gravitational constant, defined as 6,67259 x 10^{-11} m^3 / (kg . s^2)

Constant bohr_radius
Returns the Bohr's radius, defined as 0,529177249 x 10^{-10} m.

Constant bohr_magneton
Returns the value of the Bohr's magneton, defined as 9,2740154 x 10^{-24} joule / tesla.

Constant Boltzmann
Returns the value of the Boltzmann's constant, defined as 1,380658 x 10^{-23} joule / °K.

Constant accel
Returns the standard acceleration constant, defined as 9,80665 m / sec^2.

Constant light
Returns the constant that represents the light speed in a vacuum, defined as 299.792.458 m / sec.

Constant electron_charge
Returns the value of the electron charge, defined as 1,60217733 x 10^{-19} coulomb.

Constant Planck
Returns the Planck's constant, defined as 6,6260755 x 10^{-34} joule . sec.

Constant Avogadro
Returns the Avogadro's constant, 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.

4.4 Techniques to Avoid the Rewriting of Rules

This section describes the different techniques that allows to avoid the rewriting of rules, allowing to reuse them in other models, and facilitating the reading and maintenance of the model.

4.4.1 Clause *Else*

When the clause **portInTransition** is used (see section 2.3) for the description of the function to use when an external event arrives through an input port, it is possible to use the clause **else** to give an alternative behavior in case that none of the rules evaluates to true, and to avoid to rewriting code.

In the Figure 26 is shown an example of use of the clause *Else*. The cells of this model use the function *default_rule* to calculate their new state, and the cell (13,13) uses the function *another_rule* when an external event arrives for the port *In*. This function is compound of a series of rules. If when evaluating the conditions of all these rules none of them is valid, the clause *else* determines that unction *default_rule* will be used for the calculation of the state of the cell.

```
[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 26 – Example of use of the clause *Else*

The clause *Else* can call to any function that defines the behavior of a cell, even to another function that contains another clause *Else* and that it describes the behavior before the arrival of an event for a port of another cell. However, a wrong use of these could generate a circular reference, which are not detected by the simulator, and that it would cause an infinite cycle that would block to the simulation process, like it is shown in the Figure 27.

```
[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 27 – Example of a circular reference produced by a bad use of the clause *Else*

These circular references can also be given in less direct form that could be implied *n* functions, where the first function references by means of an *else* to the second function, the second reference to the third, ..., the function *n*–1 reference to the n^{th} function, and the n^{th} references to the first.

When the clause *else* references to the same function where is being used, as is shown in the Figure 28, *N–CD++* will detect this situation and it will produce an error during the parsing process.

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

Figure 28 – Example of a circular reference detected by the simulator

4.4.2 Preprocessor – Using Macros

The tool allows to the language to use a preprocessor that acts on the file that contains the definition of models. This preprocessor can be disabled by means of the parameter **–b** in the invocation of the simulator, accelerating the load of the models.

The clause **#include** allows to include the content of a file. Their format is:

#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 file of definition of models is.

The clause *#include* should only be contained in the files of definition of models, and it can exist more than an inclusion of different files inside the definition of models.

The clauses **#BeginMacro** and **#EndMacro** allow to give beginning and finish to the definition of a macro.
A macro definition has the form:

```
#BeginMacro(macroName)
...
...definition of the macro...
...
#EndMacro
```

Figure 29 – Format used to define a Macro

The content of the macro is arbitrary and can have any quantity of lines. The definitions of macros cannot be contained in the same file where they are invoked.

The clause **#Macro** allows the use of a previously defined macro, replacing the text that invoke it for the content of this macro. Their format is:

#Macro(*macroName*)

The file of macros can contain any quantity of macros, no matter how much of these are used in the model.

The text that figures outside of the definition of a macro is ignored, allowing in this way to include comments about the functionality of it.

If a required macro is not found in none of the files included with the clause *#include*, an error will be generated and the tool will abort its execution.

The clause *#include* can be defined in any place of the file, but always before to the clause *#Macro* that uses the macro whose description is contained in the file referenced by the *#include*.

Inside the definition of a macro, it cannot be carried out an invocation to another macro.

The preprocesador also allows the use comments in any part of a **.MA** file. The comments begin with the character '**%** ', and when the preprocesador finds it, ignores the string that are understood among the character '**%** ' until the end of the line is reached.

```
% Here begins the rules
Rule : 1  100  { truecount > 1 or (0,0,1) = 2 }    % Validate the existence
                                                   % of another individual.
```

Figure 30 – Example of using Comments

If a file contains invocation to macros and/or it uses comments, and when executing the simulator is passed the parameter **–b** to disable to the preprocesador, this will generate an incorrect parsing of the models that maybe doesn't generate an error that abort the simulation, but being able that the models can't be correctly and this will generate an incorrect behaviour.

In the section 16.5 a variant of the implementation of the *Game of the Life* that is defined in 4 dimensions, and which uses macros and comments.

For details of where the temporary files are generated by the preprocessor, see the *Appendix B.*

5 File for the Definition of the Initial Values of the Model

To specify the initial values that a model will take you could use the clause *InitialCellValue*, as was commented in the section 2.3. This clause allows to specify the name of a file that will contain the values that will be assigned for some or all the cells of the model before begin the simulation. The format of this file is shown in the Figure 31.

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

Figure 31 – Format of the file used to define the initial values of a cellular model

This file must have a series of lines, where each line has the format:

tupla = real_value

For convention, the extension **.VAL** is used in the name of this kind of files.

The dimension of the tupla should coincide with the defined for the model and should be contained in the space specified by this dimension.

For the definition of the initial values of a cellular model, a single file should be used, and each file won't be able to contain the initial values of two or more models.

It is not necessary that they are defined values for all the cells of the model. Those cells that don't have associate any value inside the file will be initialized with the value designed by the clause **Initialvalue**.

The interpretation of the lines of the file is carried out in sequential order. Then, if is defined a value for a cell and later a new value for the same one, the assigned value will be the most recent.

Example: In the Figure 32 a file that describes the initial values of some cells of a model of 4 dimensions is shown.

```
(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 32 – Example of a file for the definition of the initial values for a Cellular Model

6 File of Map of Initial Values

To indicate the initial values for a model is possible the use the clause *InitialMapValue*, as was commented in the section 2.3. This clause allows to specify the name of a file that will contain a map of values that it will be assigned to the cells of the model before begin the simulation. The format of this file consists of a series of lines, where each one contains a real value, as it is shown in the Figure 33.

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

Figure 33 – Format of the file of Map of values for a Cellular Model

Each value of the defined map will be assigned to a cell of the model according to the order that is shown in the following example:

Suppose that we have a three-dimensional cellular model of size (2, 3, 2). Then, the first value of the map will be assigned to the cell (0, 0, 0), the second value to the cell (0, 0, 1), the third to the cell (0, 1, 0), the fourth to the cell (0, 1, 1), and so on, until all the cells of the pattern have assigned a value.

If the file that contains the map of values doesn't have enough data to be assigned to all the cells of the model, an error will occur and the simulation will be aborted. On the other hand, if a map contains more values of the necessary, the initial values will be assigned until covering the requirements of the model, and the rest will be ignored.

For convention, the extension **.MAP** is used in the name of this kind of files.

Using the tool *ToMap* (see section 14) it is possible the conversion of a file that contains the description of a list of values according to the format described in the section 5 to a file of Map of Values.

7 File for the definition of External Events

The external events are defined in separated form to the description of the models. The file consists of a sequence of lines, where each line describes an event with the following format:

HH:MM:SS:MS PORT VALUE

where:

HH:MM:SS:MS	indicates the time when the event will occur.
Port	indicates the name of the port from which the event will arrive.
Value	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 34 – Example of a file for the definition of the External Events

8 Format of the Events generated as output

The output events generated by the simulator has the format similar to the file of definition of the external events:

HH:MM:SS:MS PORT VALUE

Example:

```
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 35 – Example of an Output file

9 Format of the Log File

The log file registers the flow of messages between the models that participates in the simulation. Each line of the file shows the message type, the time in which occur, the emitter and the destiny. This information is common to all the messages. In addition, if the message is type of ***X*** or ***Y***, then it will include the port and the value. For the messages or type ***D*** it will include the time of the next event, or '…' in case that this time is infinite.
The numbers that figure next to the name of the simulator associated to each model only are for information for the developer.

Example:

```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
Mensaje I / 00:00:00:000 / top(01) para life(02)
Mensaje I / 00:00:00:000 / life(02) para life(0,0,0)(03)
Mensaje I / 00:00:00:000 / life(02) para life(0,0,1)(04)
Mensaje D / 00:00:00:000 / life(0,0,0)(03) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,0,1)(04) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,0,2)(05) / 00:00:00:100 para life(02)
Mensaje D / 00:00:00:000 / life(0,1,0)(06) / ... para life(02)
Mensaje * / 00:00:00:100 / Root(00) para top(01)
Mensaje * / 00:00:00:100 / top(01) para life(02)
Mensaje * / 00:00:00:100 / life(02) para life(0,0,0)(03)
Mensaje * / 00:00:00:100 / life(02) para life(0,0,1)(04)
Mensaje Y / 00:00:00:100 / life(0,0,0)(03) / out /      0.000 para life(02)
Mensaje D / 00:00:00:100 / life(0,0,0)(03) / ... para life(02)
Mensaje Y / 00:00:00:100 / life(0,0,1)(04) / out /     10.500 para life(02)
Mensaje D / 00:00:00:100 / life(0,0,1)(04) / ... para life(02)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,0,0)(03)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,1,0)(06)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,2,0)(09)
Mensaje X / 00:00:00:100 / life(02) / neighborchange / 0.000 para life(0,9,0)(30)
```

Figure 36 – Fragment of a Log File

10 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 which additional information is shown during the interpretation of the rules that define the behavior of the cellular models. The output generated will consist of a sequence of characters showing the content of the buffer, where the rules are located and will be processed by the parser, and next a detailed description of each token that is identified inside the buffer is shown. In this way, if a grammatical error takes place in the writing of a rule, it is possible to identify the location of the error, since the output will show all the tokens interpreted correctly and the first appearance of an unknown or erroneous value will be informed.

In the Figure 37 the output generated is shown for the *Game of the Life* implemented in the section 16.1.

```
********* 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 37 – Output generated in the Parser Debug Mode for the *Game of Life*

11 Output of the Debug Mode for the Evaluation of Rules

Using the parameter **–v** in the invocation to the simulator is possible to activate the debug mode for the evaluation of the rules of a cellular model. All rules when being evaluated will show step to step the results of the evaluations of the functions and operators that compose it.

In the Figure 38 a fragment of the output generated for the Game of the Life implemented in the section 16.1 when the debug mode is active is shown. The numbers that are shown at the beginning of each line are not generated, but they have been added especially to refer to certain parts of the text in the following paragraphs.

The output begins with a dividing line and a legend saying "*New Evaluation*" (lines 0 and 1), indicating that a new cell will execute the transition function. Follow this, it is shown in detail the evaluation of each rule until some of them is valid.

In the line 2 begins the evaluation of the first rule for the first cell. Here it can be observed that the value of the cell (0,0) it is 0. In the line 3 the constant 1 is obtained, that later (line 4) will be

compared against the value obtained in the line 2. The legend "*BinaryOp*" indicates that a binary function is evaluating and that it receives as parameters the values 0 and 1, and the name of this function is between parenthesis, in this case the comparison is used (=). After the name of the function, we found the result of the evaluation, in this case 0 (indicating that the comparison gave as a result false). In the generated output, the value *True* is represented with a 1 and the *False* with a 0.
In the line 5 the operation *CountNode* is used with parameter 1, and its evaluation is compared with the constant 3 in the line 7.
In the line 11 the operation *OR* is evaluated among the values 0 and 0 (*False* and *False*). Their result is False.
In the line 13 the final result is indicated for the condition of the rule that is false in this case. Due to this, the following rule is evaluated (see from the line 15). Finally, in the line 24, the evaluation of the rule is valid and finishes the evaluation for the condition. Therefore, the delay of this rule is evaluated (in the line 27). In the line 28, the new value for the cell is calculated, in this case we get the constant 0, but as the delay of the rule, this can be a more complexity expression.
The ellipses of the lines 30 at 33 are not generated for the output, but rather they have been added to indicate that other evaluations exist.

```
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 38 – Fragment of the output generated by the debug mode for the Evaluation or Rules

12 Viewing the Results – *DrawLog*

The tool *DrawLog* allows to represent graphically the activity of the simulator for cellular models at each instant of time, using for it the data registered in the log file. The possible parameters are:

–h: shows the following help:

```
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
```

Figure 39 – Help shown by *DrawLog*

–?: similar to –**h**.

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

–t: Starting time. If it is not defined the tool will begin to show from the time of simulation 00:00:00:000.

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

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

–w: Allows to define the width, in characters, of the numeric values that were shown in the representation. This value should contemplate all the digits of the number, more the point and the sign of the same (in case this it is negative). For example, **–w7** define a fixed size for each value of 7 positions, and in case these values don't cover this space their representation will be completed with blank spaces.
By default, *Drawlog* assumes a value of 10 characters for the width.
For a correct representation it is recommended to use a width that is bigger or similar to the precision (defined with the parameter **–p**) + 3.

–p: Allows to define the precision, in characters, of the numeric values that were shown in the representation. If it is defined **–p0** then all the real values will be truncated to integer values and decimal digits were not shown in their representation. This parameter is generally used in combination with the option **–w**. For example: **–w6 –p2** define that all the values to show have 6 positions, which 2 will be for the fractional part, 1 will be for the decimal point, and the 3 remaining positions will be used for the integer part of the value (including the sign in case this value is negative).
By default, *DrawLog* assumes 3 characters for the precision.

–0: With this option, the numbers whose values are 0 won't be shown in the representation, and in their place blank spaces will be shown. This can be useful to appreciate certain models where great quantity of its cells has the value 0 and its contents don't frequently change.
If this parameter is not used in the invocation of the *DrawLog*, all the values 0 will be shown according to the width and precision established.

Example:

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

or

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

Figure 40 – Examples for the invocation to *DrawLog*

Note: Remember that if a cellular model is executed in *CD++* in *Flat* mode, then the *Drawlog* won't be of utility in this case, because the exchange of messages inside the flat coupled model won't be registered in the log file. For this case, activate the debug mode using the parameter **–f** of *CD++*.

DrawLog has three ways to represent the results at each instant of time for the cellular models depending on its dimension:

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

12.1 Representing bidimensional cellular models with DrawLog

When the model to be represented has dimension 2, *DrawLog* will generate a representation that consists on an schema for the state of the model at each instant of the simulated time.

In the Figure 41 a fragment of the output generated by the DrawLog is shown for a two-dimensional model of dimension (10, 10), where the parameters **–w5 –p1** have been used to format the numeric values.

```
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 41 – Fragment of the output generated for a bidimensional cellular model

12.2 Representing three–dimensional cellular models with DrawLog

When the model to be represented has dimension 3, *DrawLog* will generate a representation that consists on a series of schemas for the state of the model at each instant of the simulated time. The first schema represents the state for all the cells in the slice (*x*, *y*, 0), the second represents the state for the cells in the slice (*x*, *y*, 1), and so on until all the slices are shown.

In the Figure 42 a fragment of the output generated by the *DrawLog* for a three-dimensional model of dimension (5, 5, 4) is shown, where the parameters **–w1 –p0 –0** have been used to format the numeric values. At each instant of time, 4 graphics corresponding to the slices (*x*, *y*, 0), (*x*, *y*, 1), (*x*, *y*, 2) and (*x*, *y*, 3) are shown, where $0 \leq x, y \leq 4$.

```
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 42 – Fragment of the output generated for a three–dimensional cellular model

12.3 Representing cellular models with 4 or more dimensions

When the models to be represented have 4 or more dimensions, *DrawLog* will generate a representation that consists on a detailed listing of the reference of the cell and its respective value for each instant of the simulated time. For this mode the parameters **–w**, **–p** and **–0** of the *DrawLog* don't be used.

In the Figure 43 a fragment of the output generated by the *DrawLog* for a model of dimension 4 with size (2, 10, 3, 4) is shown.

```
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 43 – Fragment of the output generated for a model with dimension 4

13 Random Initial States – *MakeRand*

The tool *MakeRand* allows to create random initial states, which can be used for simulations of different models. The possible parameters are:

–h: show the following help:

```
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 randoms states for the Pinball Model
                 s3-n = Randoms states for the Gas Dispersion Model
```

Figure 44 – Help shown by *MakeRand*

–?: similar to –**h**.

–m: Specifies the filename that contains the definition of the model for which the random initial state will be created. This parameter is obligatory.

–c: Name of the cellular model. This parameter is obligatory and it will be fundamental to know the dimension of the model for which the random initial state will be created.

–s: Specifies the type of random initial state that will be generated. This parameter is obligatory and its possible options are:

–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 won't be generated.
For example, if we have a model with 40x40 cells and we want that initially 75% of the cells (1200 cells) has the value 1 and the rest 0, the option **–s1–1200** should be written.

–s2–n: Indicates that the random initial state must be for the model of the Pinball. For this model a value between 1 and 8 will be randomly created for the ball that will represent the initial direction of it, which will be located randomly inside the cellular space, and *n* cells of the model chosen randomly will have the value 9, representing walls. The rest of the cells will have the value 0.

–s3–n: Indicates that the initial state create must be to use with the model of gas's dispersion, where *n* gas's particles will be simulated.

Independently of the type of initial state that will be generated (specified by the parameter **–s**), the data created will always be stored in a file as it was described in the section 5 and the name of it will be generated starting from the name of the file that contains the description of the model (indicated by the parameter **–m**) but with the **.VAL** extension.

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

The tool *ToMap* allows to convert files that contains the description of a list of values according to the format described in the section 5 to a file of Map of Values (as was defined in the section 6). The possible parameters are:

–h: shows the following help:

```
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 45 – Help shown by *ToMap*

–?: similar to –**h**.

–m: Specifies the filename that contains the definition of the cellular model. This parameter is obligatory.

–c: Name of the cellular model. This parameter is obligatory and it will be fundamental to know the dimension of the model to be able to create the Map of Values.

–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 of Values.

ToMap create a Map of Values for the cells of the selected model considering that if a cell has a value specified in the .VAL file, this value will be used in the Map. Otherwise, the value of the Map for this cell will have the value specified by the clause *InitialValue* in the definition of the cellular model.

The file with the created Map of Values will have the same name that the file that contains the definition of the cellular model, but with the .MAP extension.

15 Converting .VAL files to use with *CD++ – ToCDPP*

The tool *ToCDPP* allows to modify the file that contains the description of the model (.MA), so that it will includes the values defined by a file according to the format described in the section 5. The objective of this is that the same initial values can be used by *CD++*, whenever the defined values are supported by it (**0**, **1** and **?**). The possible parameters are:

–h: shows the following help:

```
toCDPP -[?hmcio]

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

Figure 46 – Help shown by *ToCDPP*

–?: similar to –**h**.

–m: Specifies the filename that contains the definition of the cellular model and that references to the file of values (.VAL). This parameter is obligatory.

–c: Names of the cellular model. This parameter is obligatory and it will be fundamental to establish the model to set the initial values.

–i: Specifies the name of the .VAL file that contains the list of values to be used.

–o: Specifies the name of the output file (.MA).

ToCDPP takes the file that contains the definition of models (specified by the parameter **–m**), and it generates a file with the same models (specified by **–o**), but replacing the clause *InitialCellsValue* that makes reference to the .VAL file for a sequence of clauses *InitialRowValue*, such that if the models are supported by *CD++*, the generated file can be used by it without the necessity of depending on the .VAL file.

16 Appendix A – Examples

16.1 Game of Life

In the *Game of Life*, the rules are specified as follow:

- 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 another case the cell will pass to inactive.

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 47 – Implementation of the Game of Life

16.2 Simulation of the Rebound of an 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 48 – Implementation of the Rebound of an Object

16.3 Classification of Substances

The objective of this example will be 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 substance that contains 30% of carbon approximately. Also, it has a machine that locates fractions of 100 grams of that substance in a carry band. This stores them temporarily until they are processed by a packager that takes these fractions until reaching the kilogram of weight, and it packs them. Later, the packed substance is classified. If each packet contains 30 ± 1 % of carbon, then it is classified as of first quality; else, it classifies as 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 model, which generates each fractions of the substance. Another cellular model obtains the fractions of the substance from the carry band and it will carry out the packing tasks (grouping in fractions or 10 elements) and selection.

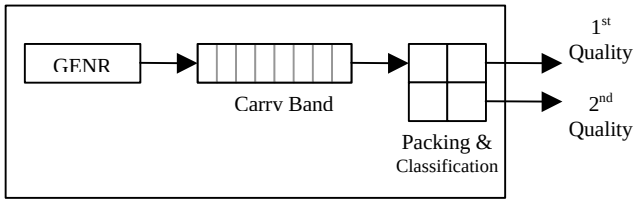

Figure 49 – Coupling structure for the Classification of Substances

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 : cola(0,-1) cola(0,0) cola(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 50 – Implementation of the Model to Classify Substances

In the definition of the model *Queue* that represents to the carry band it can be saw that has a special behavior for the external messages that arrives to the cell (0,0) coming from the generator of substances (clause ***portInTransition***). Also, in the definition of the model *Packing* this clause is used to specify the functions that describe the behaviours for the cells (0,0) and (1,0) when a substance coming from the carry band arrives.

16.4 Game of Life – 3D

The following example is an adaptation of the *Game of the Life* modelled with a cellular model of 3 dimensions. They have been carried out modifications on the rules, and in the neighborhood used, which consists of a cube of size 3x3x3 cells.

In the Figure 51 the description of the model is shown in the language provided by the tool, while in the Figure 52 the file "3d–life.val" that contains the initial values for the model is shown.

```
[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 51 – 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 52 – Initial values for the cells of the Game of Life – 3D

16.5 Use of Macros

The following example shows the use of macros to model a version of the *Game of the Life* in 4 dimensions.

In the Figure 55 the content of the file *LIFE.INC* is shown. This file contains the definition of one of the macros used in this variant of the *Game of the Life*. This type of files can contain several definitions of macros. As it can be appreciated, it is possible the inclusion of comments. For this, write

a text outside of the definition of the macro. All text non-contained between a *#BeginMacro* and a *#EndMacro* is ignored.

```
#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 53 – 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 54 – 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 55 – 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 56 – File *life–1.inc* that contains the remaining macro for the *Game of Life* 4D

17 Appendix B – The Preprocessor and the Temporary Files

The preprocesador will generate a temporary file that will contain the definition of the models where previously all the macro invocations are replaced by it content (if they exist), and all the comments are eliminated. This temporary file is passed to the simulator for its interpretation. Due to this, if the file that contains the definition of models includes invocations to macros or comments, and in the invocation of the simulator use the parameter **–b** to ignore to the preprocessor, the simulator will use directly the file that contains this code without have been make the macro–expansiones and with the comments, which will generate an incorrect interpretation of the models.

The name of the temporary file is the value returned by the instruction *tmpnam* of the *GCC*. For the selection of the directory where the temporary files were located, the following politic is used:

1. When being compiled the *N–CD++*, it is included inside of the executable code a reference to the directory established by the variable *P_tmpdir* located in <*stdio.h*>. If this directory is not the root directory, it will be used to store the temporary file.
   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. In case that the previous step references to the root directory, it proceeds to read the content of the environment variable **TEMP**. If this variable is defined, their value will be considered as the directory to use to store the temporary files.
3. If the environment variable **TEMP** is not defined, it consults the environment variable **TMP**. If this variable is defined, their value will be considered as the directory to use to store the temporary files.
4. If the environment variable **TMP** neither is defined, the directory to be used will be the directory where the executable file of the simulator is.