Extending CD++ Specification Language for Cell-DEVS Model Definition

April, 2003

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

Director: Gabriel Wainer
Author: Alejandro López

INDEX

Abstract

This work describes two new improvements made to CD++, a tool used to study, model and simulate cellular models. The tool is an incomplete implementation of the Timed Cell-DEVS formalism. The modifications described in this work remove some limitation introduced in the previous implementation. These modifications allow the cells to use multiple state variables and to use multiple ports for inter-cell communications. The cellular model specification language has been extended to cover these cases. Thus, CD++ becomes a more powerful tool while getting closer to the implementation of the whole Timed Cell-DEVS formalism.

Abstracto

Este trabajo describe dos nuevas mejoras realizadas a CD++, una herramienta utilizada para estudiar, modelar y simular modelos celulares. La herramienta es una implementación incompleta del formalismo Timed Cell-DEVS. Las modificaciones descriptas en este trabajo eliminan algunas de las limitaciones de la implementación anterior. Estas modificaciones permiten a las celdas utilizar múltiples variables de estado y múltiples puertos para la comunicación con otras celdas. El lenguaje de especificación de modelos celulares fue extendido para cubrir estos casos. Así, CD++ deviene una herramienta más poderosa mientras se acerca a una implementación completa de formalismo Timed Cell-DEVS.

1 Introduction

Simulation is a powerful tool for studying complex systems, with quite a range of uses, from new system testing to physical phenomena understanding. The simulation process starts with a problem to solve or understand. It might be the case of a train company trying to develop a new strategy for cargo storage and railway tracks usage or a chemist trying to understand a complex process of physical diffusion taking place inside a narrow tube [Tro01]. The simulation process starts from the observation of a **real system.** Entities are identified, and an abstract representation, a **model**, is constructed. Once the model is constructed, it needs to be executed. This is done by a **simulator**, which consists of a computer system that executes the model's instructions to generate its behavior. To complete the cycle, the results obtained are compared to those of the real system for model validation. It is often the case that a modeler is only interested in a few aspects of the real system. In such a case, an **experimental frame** captures the modeler's objectives and defines the scope of the model.

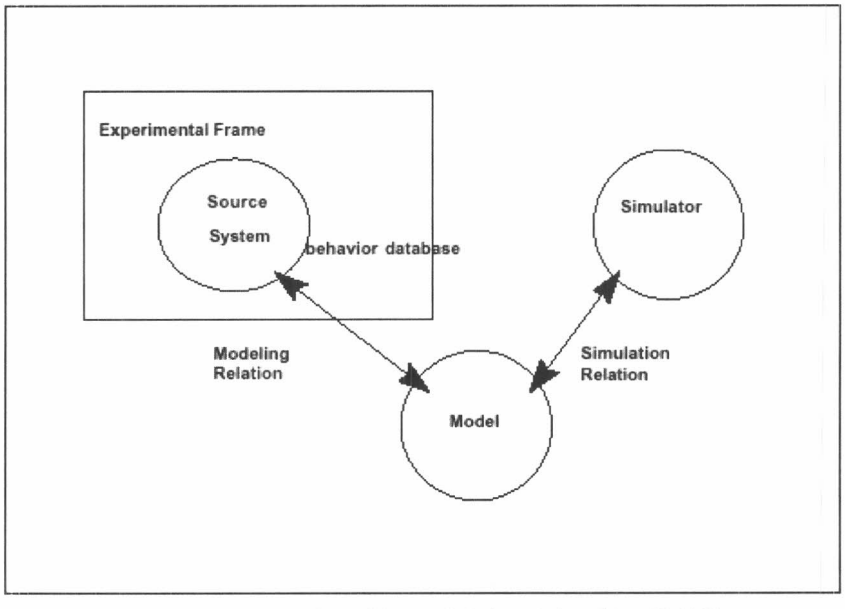


Figure 1: The basic entities and their relationships [Zei00]

The basic entities are linked by two relations [Zei00]:

- *modeling relation.* Links the real system and model, defining how well the model represents the system or entity being modeled. In general terms a model can be considered valid if the data generated by the model agrees with the data produced by the real system in an experimental frame of interest.

- *simulation relation.* Links the model and simulator. It represents how faithfully the simulator is able to carry out the instructions of the model.

There exist at present quite a number of simulation techniques and paradigms. Among these, the **DEVS** formalism provides a framework for the construction of hierarchical

models in a modular manner, allowing for model reuse and reducing development time and testing. In DEVS a model is specified as a black box with a state and duration for that state. When the duration time for the state expires, an output event is sent, an internal transition takes place and the model changes its current state. A change of state can also occur when an external event is received. Then, a complete model is defined by describing the set of states a model goes through, the internal and external transition functions, the output function and the state duration function. DEVS models can be put together by linking the outputs of a model to inputs of other models to form coupled models. Models made out of only one component are called atomic.

DEVS not only proposes a framework for model construction, but also defines an **abstract simulation** mechanism that is independent of the model itself. This mechanism is high level description of how the simulation of DEVS models should be executed by a **simulator**. Two kinds of **simulators** are defined, one for atomic and another one for coupled models, this latter known as a **coordinator**. These simulators progress through the simulation by exchanging messages as described by the abstract simulation mechanism.

Timed Cell-DEVS [Wai01] is a formalism based on DEVS for the simulation of cellular models. A cellular automaton is a lattice of cells, each of which has a value and a local rule that defines how to obtain a new value based on the current state of the cell and the values of neighboring cells. Cells are updated synchronously all at the same time. Timed Cell-DEVS defines a cell as a DEVS model and a cellular automaton as a coupled model, and introduces a new way of defining the timing of each cell which is more flexible than the existing synchronous approach. In Timed Cell-DEVS each cell defines its own update delay.

CD++ is a tool for the simulation of DEVS and Cell-DEVS models which has been used to simulate a variety of models including: traffic, forest fires, ants and watershed simulation. Simple models were easily handled by the tool, but lack of state variables and the inability to create a number of neighbor ports showed up to be a problem when writing complex models. As the workarounds used by the modelers required extra work from their side and were time-consuming during the simulation, it was proposed to add these two capabilities to CD++.

The aim of this work is to extend CD++ to allow the modeler to declare and use state variables to store values inside the cell, and to declare and use multiple neighbor (inter-cell) ports to communicate extra values to the neighbor cells. This modification will permit the modelers to remove the workarounds, reducing the simulation times, and to reduce the writing time for new complex models.

This work is organized as follows. Chapter 2 presents the DEVS and Timed Cell-DEVS formalisms. In chapter 3, the previous implementation of CD++ is introduced. Chapter 4 presents the new structure of CD++ produced by the extension, and chapter 5 shows some examples of the utilization of the new features. Finally the conclusions and future work.

2 The formalisms

2.1 The DEVS formalisms

Systems whose variables are discrete and the time advance is continuous are known as **DEDS – Discrete Events Dynamic Systems**, as opposed to **CVDS – Continuous Variable Dynamic Systems** [Wai96]. A simulation mechanism for DEDS systems assumes that the system will only change its state at discrete time points upon the occurrence of an event. An **event** is formally defined as a change of state that takes place at a specific point t_i in time, $t_i \in \boldsymbol{R}$.

DEVS is a formalism for modeling and simulation of DEDS systems. It defines a way of specifying systems whose states change upon the reception of an input event or the expiration of a time delay. It also allows for hierarchical decomposition of the model by defining a way to couple existing DEVS models.

The original DEVS model is a structure:

$$DEVS = < X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta >$$

where

X	is the set of *external* events
Y	is the set of *output* events
S	is the set of *sequential* states;
$\delta_{ext}: Q \times X \rightarrow S$	is the *external state transition function*;

where $Q := \{ (s, e) \mid s \in S, 0 \leq e \leq ta(s) \}$ and e is the elapsed time since the last state transition.

$\delta_{int}: S \rightarrow S$	is the *internal state transition function*;
$\lambda : S \rightarrow Y$	is the *output function;*
$ta : S \rightarrow R_0^+ \cup \infty$	is the *time advance function*;

The semantics for this definition are as follows. At any given time, a DEVS model is in a state $s \in S$ and in the absence of external events, it will remain in that state for a period of time as defined by $ta(s)$. The $ta(s)$ function can take any real value between 0 and ∞. A state for which $ta(s) = 0$ is called a **transient state**. On the other hand, if $ta(s) = \infty$, the system will stay in that state forever unless an external event is received. In such a case, s is called a **passive state**. Transitions that occur due to the expiration of $ta(s)$ are called **internal transitions**. When an internal transition takes place, the system outputs the value $\lambda(s)$, and changes to state $\delta_{int}(s)$. A state transition can also happen when an external event occurs. In this case, the new state is given by δ_{ext} based on the input value, the current state and the

elapsed time. Figure 2 illustrates this definition by specifying a model of a computer processor using DEVS.

A computer processor can be specified as a DEVS model. A processor would have two states: busy and available. So

$S = \{ busy, available \}$

Jobs will constitute the set of input events and output events. A job arriving on an input port will change the processor state to busy. Once the job has been processed it will be sent as an output event. Jobs will be identified with natural numbers, hence

$X = N$

$Y = N$

Assuming no job arrives while the processor is busy and that the model keeps an internal variable with the id of the job it is processing, then the external transition function is defined as follows:

$\delta_{ext}(x, e)$
{
$s = busy$
$jobId = x$
}

A job will occupy the processor during a random time with a given Poisson distribution, so the time advance function is

$ta(busy) = Poisson()$
$ta(available) = \infty$

If the processor is available, then it will remain in that state until an external event arrives.

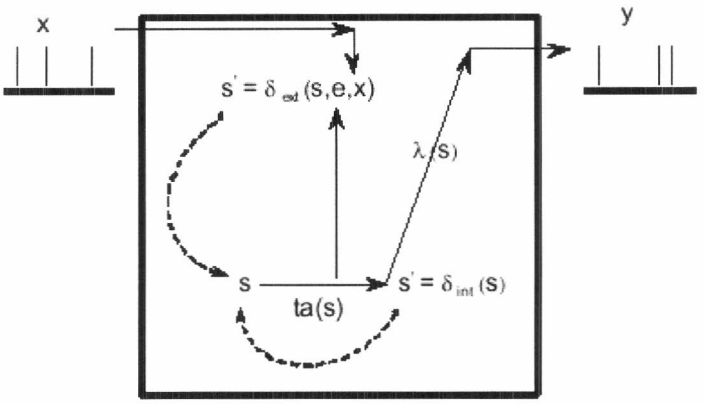
When the processing time has expired, a state transition will take place. At this time, the output function is called followed by the internal transition function. Continuing with our description,

$\lambda(busy) = jobId$

$\delta_{ext}(busy) = available$

An internal transition from the available to busy state will never happen because available is a passive state.

(a)



(b)

Figure 2: (a) Specification of a computer processor using DEVS
(b) DEVS semantics

A *coupled model* is a structure:

$$DN = < X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select)$$

where

D is a set of components.

for each $i \in D$,

M_i is a component with the constraint that
$M_i = < X_i, Y_i, S_i, \delta_{i\ ext}, \delta_{i\ int}, \lambda_i, ta_i)$ is a DEVS model

for each $i \in D \cup \{ self \}$,

I_i is the set of influences of i.

for each $j \in I_i$

$Z_{i,j}$ is a function, the *i - to -j* output-input translation

select is a tie-breaker function.

I_i is a subset of $D \cup \{ self \}$, i is not in I_i,

$Z_{self,j}$: $X_{self} \rightarrow X_j$

$Z_{i,self}$: $Y_i \rightarrow Y_{self}$

$Z_{i,j}$: $Y_i \rightarrow X_j$

select : subset of D $\rightarrow$ D

such that for any non-empty subset E,

$select(E) \in E$

A coupled model groups several DEVS models together into a compound model that can be regarded, due to the closure property, as another DEVS model. This allows for hierarchical model construction. A DEVS model that is not constructed as a coupled model is known as an atomic model.

A coupled model can have its own input and output events, as defined by the X_{self} and Y_{self} sets. Upon receiving an external event, the coupled model has to redirect the input to one or more of its components. In addition, when a component produces an output, it has to be mapped as another component's input or as an output of the coupled model itself. All these input-output mappings are defined by the Z function.

When models are coupled together, ambiguity arises when there are more than one component scheduled for an internal transition at the same time. The first model to make its internal transition will produce an output that may be translated to an external event being received by another component model that is already scheduled for an internal transition at that time. But then, should this second model process the external transition first with $e = ta(s)$? or is it the internal transition that should be executed first and then the external transition with $e = 0$? The way the DEVS formalism solves this problem is by the use of the *select* function. Only one model of the group of imminent models will be allowed to have $e = 0$. The other imminent models will be divided into two groups: those that do receive the external output from this model, and those that do not. The first group will execute their external transitions functions with $e = ta(s)$ and the second group will be among the group of imminent models for the next simulation cycle, which may require again the use of the select function to decide which model will execute first.

This tie-breaking approach is a potential source of errors since the serialization may not reflect the correct system's behavior upon the occurrence of simultaneous events. In addition, the serialization reduces the possibility of a speed up in a parallel environment. For these reasons, the parallel DEVS formalism was revised giving place to the Parallel DEVS formalism [Wai00] [Tro01].

2.2 Cellular Automata

Cellular Automata are used to describe real systems that can be represented as a cell space. A cellular automaton is an infinite regular n-dimensional lattice whose cells can take one finite value. The states in the lattice are updated according to a local rule in a simultaneous and synchronous way. The cell states change in discrete time steps as dictated by a local transition function using the present cell state and a finite set of nearby cells (called the neighborhood of the cell).

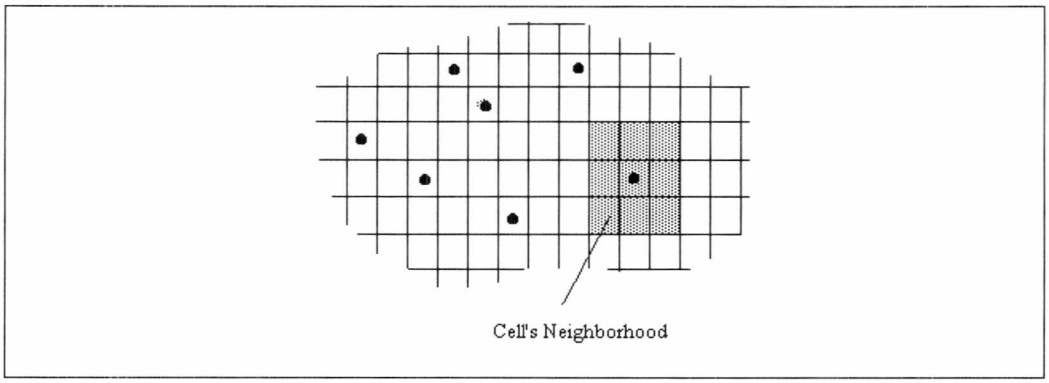


Figure 3: Sketch of a Cellular Automaton [Wai96]

When cellular automata are used to simulate complex systems, large amounts of computation time are required, and the use of a fixed interval discrete time base poses restrictions in the precision of the model. The Timed Cell-DEVS formalism [Wai01] tries to solve these problems by using the DEVS paradigm to define a cell space where each cell is defined as a DEVS atomic model. The goal is to build discrete event cell spaces, improving their definition by making the timing specification more expressive.

2.3 The Timed Cell-DEVS formalism

Cell-DEVS defines a cells as DEVS atomic models. A Cell-DEVS atomic model is defined by [Wai01]:

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

where

X	is a set of external input events;
Y	is a set of external output events;
I	represents the model's modular interface;
S	is the set of sequential states for the cell;
θ	is the set of the cell's state variables;
E	is the set of states for the input events;
delay	is the type of delay: transport or inertial;
d	is the transport delay for the cell;
δ_{int}	is the internal transition function;

δ_{ext}	is the external transition function;
τ	is the local computation function;
λ	is the output function; and
D	is the state's duration function.

A cell uses a set of input values E to compute its future state, which is obtained by applying the local computation function τ. A delay function is associated with each cell, deferring the output of the new state to the neighbor cells. There are two types of delays: inertial and transport delays. When a transport delay is used, the future value will be added to a queue sorted by output time. Therefore, all previous values that were scheduled for output but that have not yet been sent, will be kept. On the contrary, inertial delays use a preemptive policy: any previous scheduled output value, unless the same as the new computed one, will be deleted and the new one will be scheduled. This activation of the local computation is carried by the δ_{ext} function.

θ is defined [Wai02] as {s, phase, σqueue,σ}

where:
$s \in S$,
phase$\in$ {active, passive},
$\sigma\text{queue} = \{(v_1,\sigma_1),...,(v_m,\sigma_m) / m \in N, m < \infty \wedge \forall (i \in N, i \in [1,m]), v_i \in S \wedge \sigma_i \in R_0^+ \cup \infty\}$,
$\sigma \in R_0^+ \cup \infty$

After the basic behavior for a cell is defined, the complete cell space will be constructed by building a coupled Cell-DEVS model:

$$GCC = < Xlist, Ylist, I, X, Y, n, \{t_1,...,t_n\}, N, C, B, Z, select >$$

where

Xlist	is the input coupling list;
Ylist	is the output coupling list;
$I=< \eta, \mu^x, \mu^y, \boldsymbol{p}^x, \boldsymbol{p}^y >$	represents the definition of the interface for the modular model whose size is $\eta \in N, \eta < \infty$; $\boldsymbol{p}^x$ is the set of all input ports (η neighbor ports + μ^x external ports) and $\boldsymbol{p}^y$ is the set of all output ports (η neighbor ports + μ^y external ports);
X	is the set of external input events;
Y	is the set of external output events;
n	is the dimension of the cell space;

$\{t_1,...,t_n\}$	is the number of cells in each of the dimensions;
N	is the neighborhood set;
C	is the cell space;
B	is the set of border cells;
Z	is the translation function; and
select	is the tie-breaking function for simultaneous events.

This specification defines a coupled model composed of an array of atomic cells. Each cell is connected to the cells defined in the neighborhood, but as the cell space is finite, either the borders are provided with a different neighborhood than the rest of the space, or they are "wrapped", meaning that cells in one border are connected with those in the opposite one. Finally, the Z function defines the internal and external coupling of cells in the model. This function translates the outputs of m-th output port in cell C_{ij} into values for the m-th input port of cell C_{kl}. Each output port will correspond to one neighbor and each input port will be associated with one cell in the inverse neighborhood.

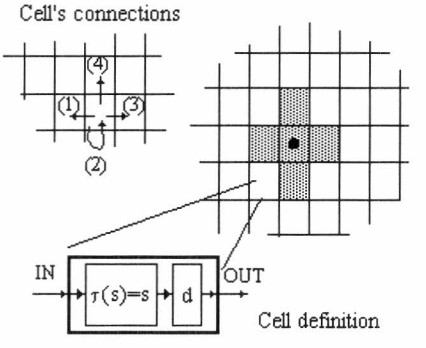


Figure 4: Informal definition of a Cell-DEVS model [Wai98]

The select function serves the same purpose as in the original DEVS models: to tie-break between imminent components.

The use of the select function introduces similar problems to those described for coupled DEVS models: lack of parallelism exploitation and a probable inconsistency with the real system. In addition, the timed Cell-DEVS was restricted to one input from each input port. Such restriction do not allow [Wai00]:

- zero-delay transitions
- external DEVS models sending two simultaneous events to the same cell.

Forbidding zero-delay transitions is too restrictive, and so is allowing only one event per external model, specially after the Parallel DEVS formalism allowed a basic model to send

more than one event at a time. These were enough reasons to revise Cell-DEVS and the Parallel Cell-DEVS formalism was proposed. This latter formalism will not be described here because it does not directly affect this work. Please refer to [Wai00] and [Tro01] for further information.

3 Previous CD++ Architecture

The CD++ architecture is based on the proposition made on [Wai97]. Even if the first versions were somehow limited [Bar98], it continued evolving [Rod99c] [Tro01].

The architecture is briefly described in this section. However, not all the components are described. The description will be provided only for the most important components or those affected by this work. For more detailed information refer to [Rod99c].

3.1 Model hierarchy

Models describe the wanted behavior in a simulation. *Atomic models* are the basis for *coupled models* which interconnect atomic models creating a larger model. All these models have common characteristics.
A few classes are used to create all these objects during a simulation [Rod99c]. The abstract class **Model** is the root of this tree. Being an abstract class, it cannot be used to instantiate objects.

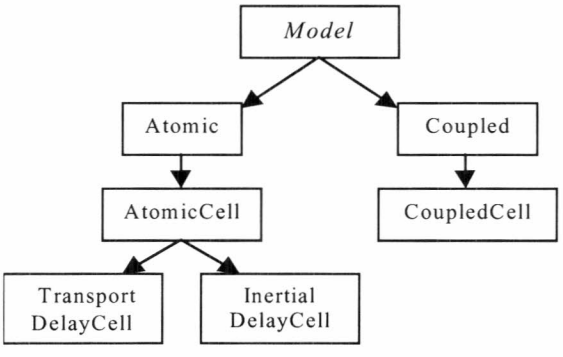


Figure 5: Model hierarchy [Rod99c]

Note: This version of CD++ does not support flat coupled cell models. This fact made this type of models become uninteresting for this work, and thus all reference to them has been skipped.

3.1.1 Model

This is the basic abstract class, from which all the models are subclasses. It is responsible for:

- managing all the input and output ports,
- knowing when the next event is scheduled,
- knowing its identifier and its parent model.

3.1.2 Atomic

This abstract specialization of the *model* class represents the interface of an atomic model. In addition to all the responsibilities inherited from *model*, it also provides the interfaces for:

- the initialization function,
- the internal and external transition functions,
- the output function,
- changing the model's state.

3.1.3 AtomicCell

A new abstract class is a specialization of the *Atomic* class. It provides the interfaces for the cells of a cellular model. Its responsibilities are:

- knowing the local computation function,
- the cell's neighborhood,
- the available ports,
- the cell's value.

When an instance of a non-abstract subclass is created, this class will take care of notifying the neighbor cells the cell's initial value. The *neighborChange* input port and *out* output port are created. The rest of the input and output ports are created dynamically as needed. These dynamic ports are stored in two lists named *in* and *output*. A local computation function is associated to each input port, in order to allow the cell to have a different behavior when a value arrives through a port.

3.1.4 TransportDelayCell

This is a non-abstract subclass of *AtomicCell*. It represents the cells that use a transport delay, by redefining the behavior internal transition, external transition and output functions. The transport delay applies a FIFO policy to the events. Any new event will be queued waiting to be executed latter.

3.1.5 InertialDelayCell

Another non-abstract subclass of *AtomicCell*. It represents the cells that use an inertial delay, by redefining the behavior internal transition, external transition and output functions. When an event arrives, the cell evaluates its local computation function, getting a value and a delay. If the remaining time for the next scheduled event is greater than 0, the value is removed and the next event is re-scheduled with the new delay.

3.1.6 Coupled

This specialization of *model* represents a coupled model. It groups other models, which in turn can be atomic or coupled, and creates a larger model by assembling the basic models. Its is responsible for:

- having a list of basic models,
- providing the means to manage that list.

3.1.7 CoupledCell

This class is a specialization of the *Coupled* class, and represents a particular type of coupled model: *cellular coupled models*. Its responsibilities are to know:

- the cell lattice, its dimension and size,
- the type of delay and the default delay,

- the type of border,
- the initial value for each cell,
- the default local computation function, and
- the zones with alternate behavior.

It is also responsible for the creation of the cell lattice and linking the cells with each other.

3.2 Processor hierarchy

Processors' main responsibility is to provide the needed simulation mechanisms for models to execute their behavior. The abstract root class **Processor** receives messages and executes the corresponding actions.

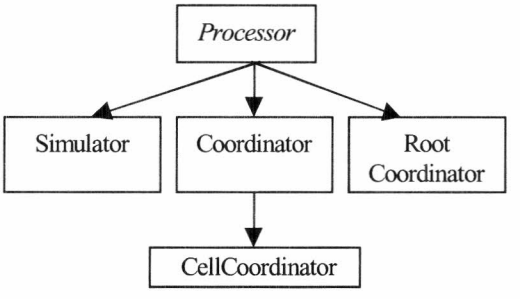


Figure 6: Processor hierarchy [Rod99c]

Note: As already stated, this version of CD++ does not support flat coupled cell models. This fact made this type of models become uninteresting for this work, and thus all reference to them has been skipped.

The class **RootCoordinator** is the root of the component tree in a simulation. **Simulator** and **Coordinator** are the processors responsible for executing atomic and coupled models respectively. **CellCoordinator** is a special coordinator dedicated to cell coupled models.

During the simulation, there is a relation 1–to–1 between models and processors: each model has a unique processor exclusively dedicated. The relation between models and processors is given by the pairs *Atomic-Simulator*, *Coupled-Coordinator* and *CellCoupled-CellCoordinator*.

3.2.1 Processor

This class is the basic abstract class for processors and represents the simulation mechanism used. It is responsible for:

- receiving messages of any type,
- knowing the associated model,
- knowing its parent processor, and
- sending the output messages to its parent processor.

3.2.2 Simulator

This subclass is a specialization of the *Processor* class. It is capable of executing only atomic models. It is charged of:

- forwarding the input messages to the atomic model, and
- sending to its parent processor the scheduled time for its associated atomic model's next event.

3.2.3 Coordinator

Another specialization of *Processor*, this subclass is represents the simulation mechanism for coupled models. It is responsible for:

- forwarding the initialization messages to all its children processors,
- forwarding the external and output messages to the corresponding influenced processor, and
- forwarding the internal messages to the imminent processor.

3.2.4 RootCoordinator

This subclass is another specialization of *Processor*. It represents the root of the processor tree from which the simulation starts. Only one instance can exist. It is the only processor which has no associated model, and it has special responsibilities:

- starting and stopping the simulation,
- managing the external events,
- output the received output messages, and
- increment the time during the simulation.

3.2.5 CellCoordinator

As a specialization of the *Coordinator* class, this class coordinates cellular models. It is charged of:

- selecting the imminent model,
- the management of the output messages to avoid duplicated messages being send to the cells (because they only indicate the need of state recalculation), but
- not filtering an external value, as in any other model.

3.3 Model and Processor Administration

Models and processors used during the simulation are created when the model description file is read, and destroyed when the simulation finishes. For models and processor to be able to reference other models and processors, there must exist a mechanism capable of getting a reference out of their name. This is the function of the **administrators**.

3.3.1 ProcessorAdmin

This class administers all the processors participating to the simulation. Because only one instance of this class must exist, it must be know of all the components of the simulation. This only instance is named *SingleProcessorAdmin* and it is responsible for:

- The creation of all the processors (root coordinator, simulators, coordinators and cell coordinators), and
- It is capable of retrieving a processor from its identification.

3.3.2 ModelAdmin

Similar to *ProcessorAdmin*, this class administers the models used in the simulation. Because all the models have an associated processor, their creation happens simultaneously. Only one instance of this class must exist. It is publicly known and is named *SingleModelAdm*.

3.4 Message Passing

Message passing between processors is the basis for the simulation mechanism. There must be as many different massage types as event types in the formalism. Also, each message could carry information specific to the type of event it represents.

As the message passing mechanism is encapsulated, the message distribution policy can be changed without impacting the rest of the modules. The currently chosen policy is FIFO, as the sending of a message will happen only when the model finished processing the previous one. This policy produces a sequential simulation.

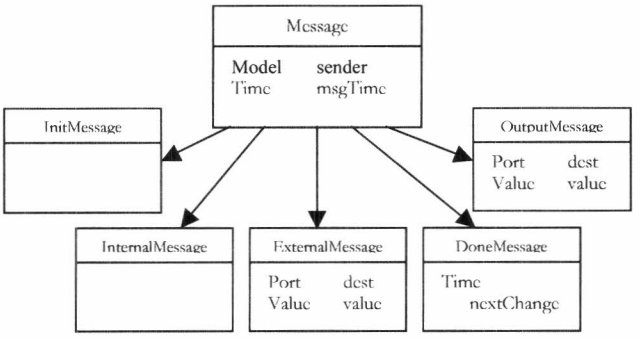


Figure 7: Message hierarchy [Rod99c]

3.4.1 Message

This is the root abstract class for all messages. It is responsible for knowing the time of the message and its sender.

3.4.2 InitMessage

This subclass of Message represents the message that the processors receive when the simulation begins. It has no extra information.

3.4.3 InternalMessage

As a specialization of *Message*, this class indicates to the destination processor that the time for an internal event has arrived. It corresponds to the * message in the DEVS formalism.

3.4.4 ExternalMessage

A subclass of *Message* that represents the arrival of an external event. It corresponds to the **X** message in the DEVS formalism. In addition to the information provided by *Message*, this class includes:

- the port of arrival, and
- the value.

3.4.5 DoneMessage

This specialization of *Message* represents the message that a processor receives from one of its child processors indicating the time for the child's next state change. It corresponds to the **Done** message in the DEVS formalism.

3.4.6 OutputMessage

Another subclass of *Message*. This one represents the output messages. It corresponds to the **Y** messages in the DEVS formalism. In addition to the information provided by its superclass, it includes:

- the output port, and
- the value.

3.4.7 MessageAdm

This class does not represent a message, but the encapsulation of the message passing mechanism. It manages the requests for sending messages between processors. Only one instance exists, which is publicly known, and is named *SingleMessageAdm*. It works by queuing the messages that processors want to send to other processors, until it is said to send the messages.

3.5 Rule Evaluation Hierarchy

3.5.1 SyntaxNode

This abstract class describes a node of a rule's evaluation tree. Type checking is done in tree itself because each construction knows the required type of its parameters. The tree is constructed by the function **yyparse**, which is generated by *yacc* from the grammar specification.

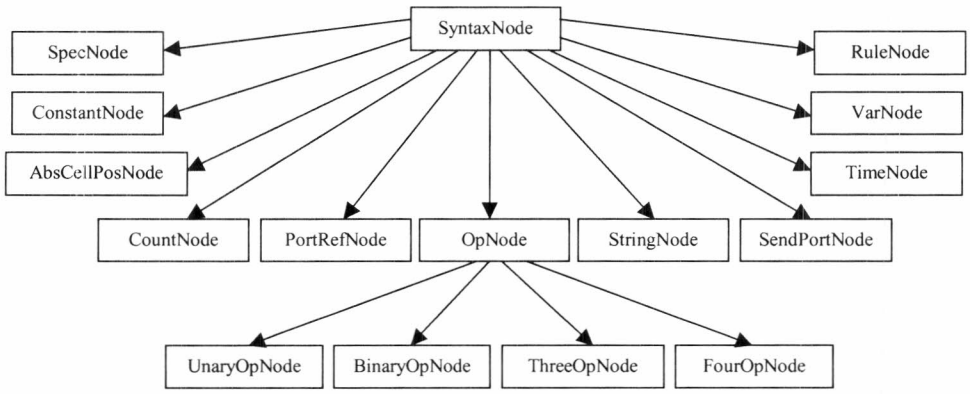


Figure 8: Class hierarchy for the tree nodes representing the rules

This class includes the methods:

- **evaluate**: returns the result of the evaluation of the node,
- **checkType**: checks the types of the node,
- **name**: returns the name of the node,
- **print**: show information about the node in the specified stream.

3.5.2 SpecNode

This class is a specialization of *SyntaxNode* and represents a list of rules and includes methods to add new rules and to look for a valid rule. A node of this class implements the local computation function and the method **evaluate** executes it.

3.5.3 RuleNode

This subclass of *SyntaxNode* represents a rule. It includes the expressions for the three parts of the rule: *value*, *delay* and *condition*. The method **evaluate** evaluates the expression which is the condition of the function; the method **value** returns the result of the evaluation of the expression in the *value* part of the rule; and the method **delay** evaluates the subtree for the delay part.

3.5.4 VarNode

This class a specialization of *SyntaxNode* is represents a reference to a neighbor cell, defined as an offset from the center of the neighborhood (the cell whose local transition function is being computed).

Note: The name of this class is out-dated. In the beginning the only available variables in a rule's specification were the references to other cells. With N-CD++ [Rod99a] and this work, other variables were introduced. Those new variables are not represented by this class, making its name incorrect, but it was decided to keep the original name.

3.5.5 ConstantNode

This subclass of *SyntaxNode* represents a constant value in a rule. The accepted types are real, integer and three-state Booleans, although all of them are mapped to the class *Real*. When a reference to a symbolic constant (*t*, *f*, *?*, *pi*, *light*, etc) is found in a rule, it will be converted to its numeric value before creating the *ConstantNode* object.

3.5.6 PortRefNode

This specialization of *SyntaxNode* represents a reference to an input port. When evaluated it returns the last value arrived through that port. The port must be one of the dynamic ports introduced by [Rod99c], those linked to the coupled model's input ports.

3.5.7 StringNode

This class is a specialization of *SyntaxNode* that stores a character string and represents a port name.

3.5.8 SendPortNode

This subclass of *SyntaxNode* is used to send a value through an output port. The port must be one of the dynamic ports introduced by [Rod99c], those linked to the coupled model's output ports.

3.5.9 TimeNode

This is a specialization of *SyntaxNode* that represents the function **time**. Its evaluation provides the simulation time.

3.5.10 AbsCellPosNode

A subclass of *SyntaxNode* that represents the function **cellpos** which returns the part of the position of the cell evaluating its local computation function.

3.5.11 CountNode

Another specialization of *SyntaxNode*, this one permits to count the number of neighbor cells that have a particular value. It is used by **TrueCount**, **FalseCount**, **UndefCount** and **StateCount**.

3.5.12 OpNode

This is a template of an abstract subclass of *SyntaxNode*. It represents the functions that take one or more arguments.

3.5.13 UnaryOpNode

This is a template abstract subclass of *OpNode* that represents a function with exactly one argument. Its template parameters are the operation, the type of the returned value and the type of the argument; and it has one member which is a pointer to the syntax node that represents the argument.

This class is used to instantiate specialized subclasses by instantiating the template parameters, creating in this way the classes for each unary function available in the language, such as **not**, **even**, **odd**, etc.

3.5.14 BinaryOpNode

This is a template abstract subclass of *OpNode* that represents a function with exactly two arguments. Its template parameters are the operation, the type of the returned value and the type of the arguments (all of them of the same type); and it has two members which are pointers to the syntax node that represent each argument.

This class is used to instantiate specialized subclasses by instantiating the template parameters, creating in this way the classes for each binary function available in the language, such as **and**, **+**, **max**, etc.

3.5.15 ThreeOpNode

This is a template abstract subclass of *OpNode* that represents a function with exactly three arguments. Its template parameters are the operation, the type of the returned value and the type of the arguments (all of them of the same type); and it has three members which are pointers to the syntax node that represent each argument.

This class is used to instantiate specialized subclasses by instantiating the template parameters, creating in this way the class for the only function taking three arguments available in the language: **if**.

3.5.16 FourOpNode

This is a template abstract subclass of *OpNode* that represents a function with exactly four arguments. Its template parameters are the operation, the type of the returned value and the type of the arguments (all of them of the same type); and it has four members which are pointers to the syntax node that represent each argument.

This class is used to instantiate specialized subclasses by instantiating the template parameters, creating in this way the class for the only function taking four arguments available in the language: **ifu**.

3.5.17 Classes For Operations And Functions

These are the non-abstract classes derived from *UnaryOpNode*, *BinaryOpNode*, *ThreeOpNode* and *FourOpNode*. They are created by instantiating the template parameters of the mentioned super-classes, and represent each operation or function available in the language, such as **not**, **and**, **if**, **ifu**, etc.

3.5.18 Parser

This class is completely independent from the *SyntaxNode* hierarchy, but it is the class that implements the parser. This is the reasons for studying it in this section.

To evaluate the rules their specification must be previously translated into an evaluation. This class encapsulates the creation of the evaluation tree for the rules. This tree is created by the code automatically generated by **yacc** and is composed syntax nodes (instances of subclasses of *SyntaxNode*).

This class has only one instance named **SingleParser**.

3.5.19 LocalTransAdmin

Similarly to *Parser*, this class is not a subclass of *SyntaxNode* but is very closely related to the evaluation of the rules. That is the reason to study it here.

To evaluate the rules their specification must be previously translated into an evaluation tree by the *SingleParser* object. As most of the cells have the same local computation function, the generated tree is associated an identifier, and each cells stores the corresponding identifier. This class provides the means of executing a function provided the identifier.

There is only one instance of this class named **SingleLocalTransAdmin**.

3.6 Starting the simulation

3.6.1 ParallelMainSimulator

This class is the one that manages the simulator initialization. It is responsible for:

- creating the model tree (includes loading the cells),
- creating the processor tree,
- linking the models,
- providing the external events to the *RootCoordinator*,
- determination the finish simulation time, and
- starting the simulation using *RootCorrdinator*'s *run* method.

There is only one instance of this class. In this case the single instance is managed differently from the other cases. This class has a static method names **Instance** that returns a reference to the only object of the class.

4 New CD++ Architecture

4.1 Overview

The previous implementation of CD++ had two main limitations: it had no support for state variable, and the set of neighbor ports was fixed.

CD++ had no support for multiple **state variables**. To work around this problem, modelers needed to use extra planes in their cell space, and create as many new layers as state variables they needed. For instance, when one state variable was needed in a planar cell space, the solution was to create a three-dimensional cell space with two planar layers. The layer 0 (x, y, 0) was used for the cell value, and the other layer (x, y, 1) to store in the upper cell's value, the lower cell's state variable's value [Ame00]. An example of this technique can be seen in 5.2.1. One of the extensions presented in this work removes this restriction. The user can still define multidimensional models representing different phenomena, but each of the planes can include cells with multiple state variables, permitting to define more complex phenomena.

The second limitation mentioned is the fixed set of neighbor ports. The **neighbor ports** are the ports (for input and output) used to send values from one cell to another, in a coupled cell model. The previous implementation of CD++ had only one port for input and one for output. Both were automatically created together with the cell. These ports were referenced as *neighborChange* and *out*, for input and output respectively. It is important to differentiate these ports from those used to receive from, or send to the exterior of the cell coupled model. These latter ports are created automatically in only those cells affected by the arrival of external messages or by those supposed to send output messages.

Back to section 2.3, it can be said that the previous implementation had the following limitations:

$$\eta = 1$$
$$p_1^x = neighborChange$$
$$p_1^y = out$$

After this modification, the user can define which neighbor ports will be used in the model and is no longer limited to only one.

4.2 Language Extensions

The first step in the way to add state variables and ports to CD++ is to be able to declare them and later reference them in the rules. The specification language has been extended to support this.

4.2.1 State Variables

In this section are described the extensions to the language required to declare and use the state variables.

4.2.1.1 Declaring State Variables

To be able to declare and initialize the state variables, three new keywords have been added to the language: **StateVariables**, **StateValues** and **InitialVariablesValue**. The first one declares the list of state variable existing in every cell. The second one is the set of default initial values for the states variables. And the last one, provides the name of a file where the initial values for some particular cells are stored.

```
StateVariables: pend temp vol
StateValues: 3.4 22 -5.2
InitialVariablesValue: initial.var
```

In this example three state variables are declared: *pend*, *temp* and *vol*. Except that in the file *initial.var* other values are specified for a subset of cells, the state variables of every cell will be initialized with the values 3.4, 22 y -5.2 respectively.

The format for the initial values file is quite simple. Each line references a unique cell, followed by an "equal" sign (=) and the list of initial values for every state variable in the cell. The initial values must be separated by, at least, one space character. The values will be assigned to the state variables following the order in which they are listed in the sentence *StateVariables*.

```
(0,0,1) = 2.8 21.5 -6.7
(2,3,7) = 6 20.1 8
```

The first line will assign to the variable *pend* of the cell (0,0,1) the value 2.8; to the variable *temp* of the same cell, the value 21.5; and the value -6.7 to the variable *vol* of the same cell. The second line will assign respectively the values 6, 20.1 and 8, to the variables *pend*, *temp* and *vol* in the cell (2,3,7).

4.2.1.2 Referencing state variables

The state variables can only be referenced from within the rules that define the cells' behavior (*local computation function*). A variable is referenced by its name, as it was declared in the *StateVariables* sentence, preceded by a **dollar sign** (**$**), from any part of a rule.

```
rule: { (0,0,0) + $pend }  10  { (0,0,0) > 4.5 and $vol < 22.3 }
```

4.2.1.3 Assigning values to the state variables

The identifier '**:=**' is used to assign values to a state variable. Any expression returning a numeric value can be placed on the right side of the assignation, but on the left side, there can only be a reference to a state variable.

Contrarily to what happens with the references, the assignations can only be placed in a new part of the rules specifically created for this purpose.

```
<value> [ { <assignations> } ] <delay> <condition>
```

The new part is optional, and if present, it must be enclosed between curly brackets. The contents is a list of assignations, each one followed by a **semi-colon**.

```
rule: { (0,0,0) + 1 } { $temp := $vol / 2; $pend := (0,1,0); }
                      10 { (0,1,0) > 5.5 }
```

In the example, if the condition happens to be true, the variable *temp* will be assigned half of *vol*'s value, *pend* will be assigned the value of the neighbor cell (0,1,0), and *vol*'s value will remain unmodified. This assignations are executed immediately, which means that they *are not delayed*, as happens to the output value.

It is important to notice that the assignations are done from left to right. Because of this, the two following rules **are not equivalent**.

```
rule: 5 { $vol := 1; $temp := $vol; } 10 { t }
rule: 5 { $temp := $vol; $vol := 1; } 10 { t }
```

After executing the first rule, both *vol* and *temp* will have the value 1. On the contrary, when the second rule is executed, *vol* will have the value 1, but *temp* will have *vol*'s previous value.

4.2.2 Neighbor Ports

This section describes the extensions done to the language in order to support the use of multiple neighbor ports. In addition to how to declare and reference them, this section include some extra modification needed to keep the language coherency.

4.2.2.1 Port Declaration

Only one keyword was added to the language: **NeighborPorts**. This keyword takes as its argument a list of neighbor port names. Notice that only one keyword was added, but there are two lists of ports (p^x and p^y). The input and output neighbor ports share the names, making possible to calculate automatically the influences: *an output port from a cell will influence exclusively the input port with the same name in every cell in its neighborhood.*

```
NeighborPorts: alarm weight number
```

In this example three ports are declared and their names are *alarm*, *weight* and *number*. All the cells will have three input neighbor ports with these names and three more neighbor ports with the same names but dedicated to output values. When a cell outputs a value through one of these ports, it will be received by all its neighbor cells through their input ports with the same name.

If this keyword is not present in the model description, then the simulator will work in compatibility mode, behaving as the previous implementation.

4.2.2.2 Reading Values Arrived Through The Input Ports

A cell can read the value sent by one of its neighbors. To do so, it will be necessary to specify which port the value must have arrived through.

Before this modification was done, because the only available input neighbor port was *neighborChange*, there was no need to name it, it was enough to name the cell in which the modeler was interested. Currently the modeler must name both the cell and port through which the value must have arrived. The way to do it is to reference the cell in the same way as before, but now the input port reference must follow, separated by a tilde (~):

```
rule : 1 100 { (0,1)~alarm != 0 }
```

4.2.2.3 Sending Values Through The Output Ports

An important modification has been done here. So far, a rule looked like this:

```
<value> [ <assignations> ] <delay> <condition>
```

(Notice the assignations part introduced in 4.2.1.3).

The <*value*> part used to be an expression that evaluated to a single value which was sent through the only output port. As there are currently many possible ports through which the value can go out, the desired port will have to be specified. The way to express this output functions is by "assigning" the values to the output neighbor ports. The new format for the rules is now:

```
<port_assignations> [ <assignations> ] <delay> <condition>
```

Similarly to the assignations part, the <port_assignations> part of the rule must be enclosed between curly brackets. In this case, the port is referenced preceded by the tilde (~), but the cell reference should no be included.

As it can be necessary to output values through many ports at the same time, the "assignation" can be used as many times as needed. Each one must be followed by a **semi-colon**.

```
rule: { ~alarm := 1; ~weight := (0,-1)~weight; } 100
                                        { (0,1)~number > 50 }
```

Keep in mind that from now on, the <port_assignations> part no longer evaluates to a single value, but is a sequence of output operations.

The preexisting function **send()** is still available. This function sends values out of the coupled model and not just out of the cell to its neighbors. Something similar happens with the function **portref()** which reads values from outside the coupled model. Both these functions can still be used and even mixed with the new ones.

```
rule: { ~alarm := 0; send(alert, 1); } 100
                          { portref(alert) = 0 and ~alarm != 0 }
```

4.2.2.4 Collateral effects

As a result of this modification, some extra changes to the language were needed to keep it coherent.

The most important change happened to the **StateCount** function. This function used to take a value as its argument and returned the number of neighbor cells that had exported that value through their *out* port. In other words, it counted the number of times the argument arrived through the input *neighbogChange* from different neighbor cells. Because of the many possible ports that a cell can now have, the function will now take a second

argument which is the *name of the port*. If the second argument is not present, the function will execute in compatibility mode and will consider that the port is *out*.

```
rule: { ~alarm := 1; } 100 { statecount(1, ~alarm) >= 4 }
```

However, some functions kept their semantics intact. This is the case of the functions **TrueCount**, **FalseCount** and **UndefCount**. The rational for these functions not accepting a port name is that the changes needed implied a reorganization on the parser and the dictionary internally used to recognize them, and because their behavior can be simply emulated using the StateCount function. The following example show how to replace the function UndefCount.

```
rule: { ~alarm := 1; } 100 { statecount(?, ~alarm) >= 7 }
```

Another important change is that a cell's **initial value**, however it is introduced in the model, will affect all the cell's neighbor ports.

4.3 Modifications to the Simulation Mechanism

Even if both modification have a small common part, they are mainly separate. That is why they are analyzed separately in this work, except for the common part which is described first.

4.3.1 Common Modifications

A few modifications are used by both features. This sections describes these common modifications. Please notice that this section does not cover the case of classes that were affected by both improvements, but whose changes have nothing in common. These cases are treated separately.

4.3.1.1 New Classes

4.3.1.1.1 ListNode

This class represents a node of the evaluation tree which is a list of syntax nodes. In particular this node represents the sequence of assignations that the modeler can now express. The syntax nodes in this list can be assignations to state variables or assignations to neighbor ports. Since the class has no restriction on the type of nodes it includes, as long as they are *SyntaxNode*s, it is up to the user to check the types and how they are used.

```
class ListNode: public SyntaxNode
{
public:
   ListNode();
   ~ListNode();

   const string name();
   Real evaluate();
   SyntaxNode *clone();
   const TypeValue &type() const;
   ostream &print(ostream &os);
```

```
    bool checkType() const;

    void add(SyntaxNode *sn);
    list<PortValue> getPortValues();

private:
    typedef list<SyntaxNode *> SynNodeList;

    ListNode(SynNodeList &snl);

    SynNodeList NodeList;
};
```

In addition to all the methods required by the abstract class *SyntaxNode*, from which this is a subclass, two new methods are included. One of these methods is **add**, which is used to add a new node to the front of the list. Note that this method describes the policy deciding the order in which the assignation are done. If instead of adding the node in front of the list, it would add it at the back, the assignations would be done from right to left.

The other new method is **getPortValues**, which is specially present for the port assignations. Its functions is to provide the caller with a list of pairs port-value. **PortValue** is a type defined specially for the ports extension and is described in 4.3.3.2.1. Briefly, it includes a value and the name of the port through which it must be exported. The reason for this method to exist is that the method **evaluate** is designed to return one value, and not a list of assignations. In other classes, when there was need to execute some action, the action was executed inside this method and simply returned 0. This trick could not be used here because the actions were executed immediately, but the port assignations (output of the values) must be delayed. This new method provides the means for getting the results early enough to use them internally for checking, but using them after the delay.

4.3.1.2 Modified Classes

4.3.1.2.1 Parser

```
#define VAR_PREFIX      '$'
#define PORT_PREFIX     '~'

class Parser
{
private:
    int analizeToken(string &token, bool &consumed,
                     string &text);
    int nextToken();
};

void printToken(string &t, int tipo);
```

This class is responsible for the lexical analysis of the model description file. The method **analizeToken** is responsible for identifying tokens from a lexeme. It has been modified to recognize, is addition to all the already recognized token, four new tokens:

- a state variable,
- a port name,
- the assignation operation, and
- the end of an operation on a sequence.

The lexeme for the state variable token (**STVAR_NAME**) is a text string prefixed by the character specified by the macro **VAR_PREFIX**. Similarly, the **PORT_NAME** is a string prefixed by the **PORT_PREFIX**. However, there is a remark to make at this point: port names were already identified, but in very particular situations. For instances, in the previous implementation, a text string was considered to be a port name only if it was the first parameter of one of the two functions that used ports: **send** and **portref**. Currently, both cases are identified and recognized as the same token. Finally, the lexeme "**:=**" is recognized as the **OP_ASSIGN** token and the lexeme "**;**" is recognized as the token **;**.

Other methods and functions needed to be updated to deal with the new tokens. They are the method **nextToken**, charged of reading the next lexeme; and the function **printToken**, charged of printing the debug information about the recognized tokens.

4.3.1.2.2 RuleNode

```
class RuleNode: public SyntaxNode
{
public:
   RuleNode(ListNode *v = NULL, ListNode *a = NULL,
            SyntaxNode *d = NULL, SyntaxNode *be = NULL);
   ~RuleNode();

   Real assign();
   list<PortValue> value();

private:
   ListNode *val, *asgn;
   SyntaxNode *dly, *boolExp;
};
```

This class represents a rule in the description language. One member was added and one member was modified. The **val** member, which represents the part value of the rule, was modified to be a pointer to a list of nodes *NodeList*, instead of a pointer to a *SyntaxNode* as it used to be. Even if there was no real need to change the type (because *NodeList* is a subclass of *SyntaxNode*), it was done to help the compiler detect possible errors.

The new member in the class is named **asgn**, and represents the assignation part of the rule. It is also a *NodeList*.

The **constructor** and **destructor** have been modified to cope with these two changes. The method **value** was adapted to return the list of portname-value, instead of just a *Real*, and the new method **assign** executes the assignations.

4.3.1.2.3 SpecNode

```
class SpecNode: public SyntaxNode
```

```
{
public:
   list<PortValue> value() const;
   Real evaluate();

private:
   list<PortValue> lastValue;
};
```

The modified method **value** and member **lastValue** existed already in this class. They used to deal with *Real* values, but now they return a list of portname-value.

The most important change happened to the **evaluate** method. This method is the one that executes the rules. It now executes the assignations just before evaluating the <port_assignations> part of the executed rule, but after having calculated the delay. The only modification related with the ports management is that now the **lastValue** member is emptied instead of being reset to 0. The rest of the work is done by the supporting classes.

4.3.1.3 Modifications to the grammar

After having modified the lexical analyzer, it was the turn for the grammar. The changes are described using top-down approach. The whole grammar can be found in the Appendix A.

4.3.1.3.1 Rule

```
Rule :
   AssignResult Resultado '{' BoolExp '}'
   | AssignResult '{' AssignSet '}' Resultado '{' BoolExp '}' ;
```

The new language rule format is recognized, but in a way that also accepts rules in the old format, assuring the compatibility of the models.

Two changes can be seen:

- The <value> part is now an **AssignResult** and no longer a **Resultado**, and
- the optional **AssignSet** is introduced.

Two grammar rules were needed instead of just one to solve an ambiguity problem.

4.3.1.3.2 AssignResult

```
AssignResult :
   Resultado
   | '{' PortSendSet '}' ;
```

This rule recognizes two sub-trees. The sub-tree **PortSendSet** is a list of output operations through output ports. The sub-tree **Resultado** is only recognized for compatibility reasons.

4.3.1.3.3 PortSendSet

```
PortSendSet :
   /* Empty */
   | PortSend PortSendSet;
```

The **list of outputs** to neighbor ports can be **empty** or an **output operation** followed by a **list of outputs**.

4.3.1.3.4 PortSend

```
PortSend :
   SEND '(' PORTNAME ',' RealExp ')' ';'
   | PORTNAME OP_ASSIGN RealExp ';'       ;
```

An **output operation** can be either a **send** function with a **port name** and a **value** as arguments, or a **neighbor port assignation**. Both cases must be finished by a semi-colon (;).

Except for **RealExp**, all the components of the rule are basic token provided by the lexical analyzer.

4.3.1.3.5 RealExp

```
RealExp :
   IdRef
   ...
```

This rule was not affected itself, but it did one of its components: **IdRef**.

4.3.1.3.6 IdRef

```
IdRef :
   CellRef OptPortName
   ...
```

This rule recognizes the references to identifiers. In particular references to cells (**CellRef**). Now the cells can be followed by a reference to the port (**OptPortName**) whose value we are interested in. Because the compatibility must be assured, this second part of the rule is optional.

4.3.1.3.7 OptPortName

```
OptPortName :
   /* Empty */
   | PORTNAME ;
```

A reference to the port is just a token representing its name. However, as this rule is optional, also an **empty** value is accepted.

4.3.1.3.8 AssignSet

```
AssignSet :
   /* Empty */
   | Assign AssignSet  ;
```

This rule is a component of the rule *Rule* (4.3.1.3.1). It represents a list of assignations to state variables.
It can be **empty** or an **assignation operation** followed by a list of **assignations to state variables**.

4.3.1.3.9 Assign

```
Assign :
   STVAR_NAME OP_ASSIGN RealExp ';' ;
```

This rule recognizes the operation that assigns a value (**RealExp**, 4.3.1.3.5) to a state variable reference. Both **STVAR_NAME** and **OP_ASSIGN** are basic token recognized by the lexical analyzer.

4.3.2 State Variables

4.3.2.1 Overview

From a high level point of view, the modifications done to the simulator for it to be capable of providing state variables are not really complex. A new class representing a set of state variables and their values was created. An instance of this class is a member of the state of each cell. The classes *AtomicCellState* and *AtomicCell* were modified to cope with this new object. The modifications include new methods to get and set the values of the variables, identified by their name.
In addition, the parser was modified to recognize the state variable references, new syntax nodes were created to handle the assignations in the evaluation tree, and some existing syntax nodes were modified to deal with the new nodes.

4.3.2.2 New Classes

Some classes were created to add support for state variables to CD++. Those classes are described in detail hereafter:

4.3.2.2.1 StateVars

This class was created to store a set of state variables and their values. An object of this class is created empty, and the state variables are added to the set afterwards together with their initial values. The necessary methods to do this are part of the class.

When the models are being loaded, an empty object of this class is created. Later on, the state variables declared by the modeler are added to the set. At the same moment, the default initial values are assigned to the variables if the modeler provided them. If the modeler did not provide the list of default initial values, the *undefined* value is used.

Once the set is created and initialized, the object is given to the object of the class *CoupledCell*. Subsequently, it will clone the object getting copies that will assign to every cell. From this moment on, each cell will use its own independent object *StateVars*, even

thought that all of them are identical, containing the same state variables and initial values, but the values will evolve independently for each cell.

Before starting the simulation, but still during the model loading, if the modeler defined an initial values file, these values will be assigned to the state variables in the corresponding cells.

Once started the simulation, the state variables' values will be read and modified in each cell according to the rules defining the local computation function.

An object of this class is the set of state variables (and their values) of a cell.

```
class StateVars : protected map<const string, Real>
{
public:
   inline bool exist(const string& name) const;
   bool createVariable(const string& name,
                       Real& value = Real::tundef);
   Real &get(const string &name) const;
   Real &set(const string &name, Real &newValue);
   StateVars& operator=(const StateVars &src);
   StateVars& setValues(const string &values);
   string asString(void) const;
   inline void print(ostream& os) const;
   void clear();

protected:
   inline bool exist(int index) const;
   const string& operator[](int index) const;

private:
   map<int, string> order;
};
```

Because it is a subclass of *map<const string, Real>*, the name of the state variable is used as the key to access the values.

The default constructor will create an empty state variables set. The state variables will be created subsequently by calling the method **createVariable**. In the case that the variable already exists, this method returns *False* and the set is not modified. The method **exist** is used to know whether a state variable exists and the methods **get** and **set** are used to access the variables' values.

The private member *order* exists to keep a relation between the state variables and the order in which they were created. Even thought that the state variables are referenced by their name, there exists a case were they are accessed by the creation order: the method **setValues** assigns a list of values to all the state variables in the object. This method is called only during the simulator initialization when reading the values from the initial values file.

For internal use of the method *setValues*, the method **exist** and the operator **[ ]** are available. Both take an *integer* representing the index (creation order) of a state variables. That is the reason why these functions are *protected.*

Finally, the method **clear** is in charge of emptying the state variables set, and the operator = copies in the self object (the object where the method is called), the object passed as argument.

4.3.2.2.2 StateVarNode

A subclass of *SyntaxNode* that represents the state variables references in the evaluation tree. This class identifies a state variable referenced from a rule, to be able to evaluate it.

```
class StateVarNode: public SyntaxNode
{
public:
   StateVarNode(const string &VarName);
   ~StateVarNode();
   const string name();
   Real evaluate();
   SyntaxNode *clone();
   const TypeValue &type() const;
   ostream &print(ostream &os);
   bool checkType() const;
   const string &getVarName();

private:
   string varName;
};
```

The most interesting methods of this class is **evaluate**, which returns the value of the referenced state available; and **getVarName**, which return the name of the referenced state variable. The rest of the methods are present by requirement of the super-class and have the normal use.

4.3.2.2.3 AssignNode

Another subclass of *SyntaxNode*. This one represents an assignation of a numeric expression to a state variable.

```
class AssignNode: public SyntaxNode
{
public:
   AssignNode(SyntaxNode *var = NULL,
              SyntaxNode *val = NULL);
   ~AssignNode();
   const string name();
   Real evaluate();
   SyntaxNode *clone();
   void var(SyntaxNode *v);
   void val(SyntaxNode *v);
   const TypeValue &type() const;
   ostream &print( ostream &os );
   bool checkType() const;

private:
   SyntaxNode *variable;
   SyntaxNode *value;
};
```

Both the state variable reference and the numeric value must be *SyntaxNodes*. They can be provided at creation time or later using the specific methods **var** and **val**. The method **evaluate** executes the assignation of the value resulting from the evaluation of the value syntax node member, to the state variable specified by the variable syntax node member. As this method returns the assigned value, this allows for chaining and nesting of the assignations.

4.3.2.2.4 AssignSetNode

This is not really a class but an instance of the *ListNode* class described in 4.3.1.1.1. This instance is a list of assignations to state variables (*AssignNode*). Is the equivalent, in the evaluation tree, to the *assignations* part of a rule. *ListNode*'s **evaluate** method evaluates, one by one, the *SyntaxNodes* it includes. Because the included syntax node are, in this case, assignations to state variables, evaluating this list will execute all the assignations.

4.3.2.3 Modified existing classes

This section describes the changes done to existing classes. In addition to the explanation of those changes, an incomplete description of the class is presented. This incomplete description includes only the methods and members affected by the changes.

4.3.2.3.1 ParallelMainSimulator

```
class ParallelMainSimulator
{
protected:
   ParallelMainSimulator &loadCells(CoupledCell &, bool);
   ParallelMainSimulator &loadStateVariables(CoupledCell &);
};
```

The method **loadCells** is responsible for loading the cells of a cellular coupled model. It has been modified to invoke, in addition to its other tasks, the new method **loadStateVariables**.

This new method is charged of interpreting the state-variables-specific part of the model being loaded. It creates an empty set of state variables (*StateVars* object). If the model declares state variables, they will be added to this object, and if general initial values are provided for the state variables, those values will be assigned to the state variables added to the object.

Once the object is initialized (state variables created and initial values set), the object is passed to the coupled model. It is responsible for initializing each cell with this object. Notice that if the model does not use state variables, this object will stay empty, but it will still exist. This is the way in which the backwards compatibility is guarantee without using code specific to that case.

4.3.2.3.2 CoupledCell

```
class CoupledCell : public Coupled
{
public:

   CoupledCell &setVariablesValue(const CellPosition &,
                                  const char *);
   CoupledCell &setVariablesValue(const string &,
                                  const char *);
   const StateVars &initialStateVars() const;
   CoupledCell &initialStateVars(const StateVars &);

protected:
   virtual void afterProcessorInitialize();

private:
   Real      initialValue;
   StateVars initialVars;
};
```

This class has a member that specifies the initial value for each cell (**initialValue**). In a similar way, there is a member specifying the initial values for state variables named **initialVars**. This member is set by calling the new method **initialStateVars**, which is called from the *ParallelMainSimulator*'s **loadCells** method (see 4.3.2.3.1). There is second version of **initialStateVars** which is used to get the object.

The method **afterProcessorInitialize** is called once the associated processor has been initialized. This method initializes every cell and was modified to also create the state variables by copying the **initialStateVars** member. To do so, it invokes *AtomicCell*'s method *variables*. In this way, each cell will have its own state variables set, containing the same variables with the same initial values, but now they become completely independent.

This class also provides the methods to set the values for all the state variable of a particular cell. This is a set of method which accept different argument types, but they all do the same job. These methods are named **setVariablesValue** and are used when special initial values are provided for a particular cell (this are the values provided in the state-variables initial-values file). As normal, they must be called once the state variables have been created and initialized with the default initial values.

4.3.2.3.3 AtomicCell

```
class AtomicCell: public Atomic
{
public:
   Real &variable(const string &name);
   Real &variable(const string &name, Real &value)

protected:
   StateVars &variables();
   AtomicCell &variables(const StateVars &vars)
};
```

Two protected methods called **variables** were added to this class. They are used to set and get the cell's *StateVars* object representing the set of state variables. As the object is really part of the cell's state, and it is stored in another object of class *AtomicCellState*, the appropriated method is used to access the cell's state.

The public methods called **variable** are provided to set and get the value of one state variable. The variables is referenced by its name and the method internally used the services of the above-described method *variables*.

4.3.2.3.4 AtomicCellState

```
class AtomicCellState: public AtomicState
{
public:
   StateVars variables;
};
```

As this class represent the cell's state, it is logical that this is the object that really stores the *StateVars* object. A new member named **variables** was created for it.

4.3.3 Multiple neighbor Ports

4.3.3.1 Overview

Adding multiple neighbor ports was more complex than adding state variables. This part of the modification can be divided into three subtasks.

4.3.3.1.1 Addition of New Ports to the Cells

Before this modification was implemented, each cell had only two neighbor ports. These ports were named *neighborChange* (for input) and *out* (for output). The latter port influencing the former port in all the neighbor cells. This modification replaced these two ports by two lists of ports and the list of influences were extended so that a port in a cell would influence the port with the same name in all the neighbor cells.

Since the *AtomicCell* class in the previous implementation included two ports as members, one for input and one for output, there was no chance to use other ports for neighbor communications. Nowadays, the two ports in the *AtomicCell* class have been replaced by two list of ports. These list are instances of the class *PortList* (this class already existed and was used for the coupled model ports).

Each cell receives from the coupled model (its creator) a list with the names of the ports to be created. From each name, two ports are created, one for input and one for output. As both input and output ports share the same name, the port names are internally prefixed *in_* or *out_* to differentiate them. In addition to these ports, the two default *neighborChange* and *out* ports are also created (which don't need to be prefixed). These two ports are created for compatibility reasons, to support the old models, but nothing prevents the modeler from using them as any other neighbor port.

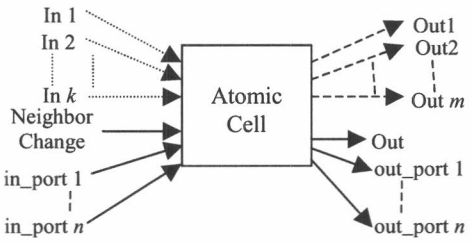


Figure 9: Structure of an atomic cell

Similarly to the historic ports, the new ports will generate **Y** messages and receive **X** messages, and because the messages already included the port name, there is no need to modify the message format.

The creation of the ports is done in *AtomicCells*'s method *createNCPorts()*. The coupled model receives the list of port names from the *ParallelMainSimulator* when the later calls *CoupledCell::createCells()* from its method *loadCells()*.

4.3.3.1.2 Extended Neighborhood Values

In the previous implementation, each cell had an object belonging to the class *NeighborhoodValue*, which was used to store the values arriving from the neighbor cells through the input port *neighborChange* (the only port available at that moment). This class used internally an object of type *NeighborList*, which was a *map* using a cell position as the key to the value. This defined a table of <cell position, value>.

Now that the number of ports is no longer limited to one, only one value for each neighbor cell is not enough. The cell must be capable of storing, for each neighbor cell, as many values as neighbor ports the cells have. To achieve this, it is necessary to extend the table used in the previous implementation to keep, not just one value, but a value for each port. The extended table can now be seen as <cell position, <port name, value>>.

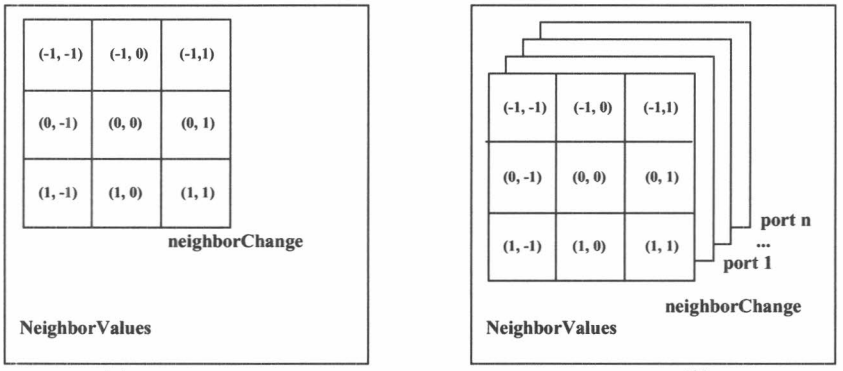


Figure 10: NeighborValues Structure: (a) In the previous implementation (b) after the modification

Because of this, the class *NeighborList* was extended to store a *map* of port names and values, instead of a single value. It was also necessary to update all the methods of this class that accessed the values (either for reading or writing) so that they accept the port name as parameter.

4.3.3.1.3 Extra Messages

The message format and type was not modified. However, it is worth noticing that when the new ports are used, the number of messages circulating between cells increases.

The default ports (*neighborChange* and *out*) kept their historic function: they are still used for administrative communications with the coupled model, such as receiving and sending the **I**, **@**, ***** and **D** messages, exactly in the same way it was done before.

When a cell's **init function** is executed, it used to send the cell's current value to the coupled model in a **Y** message, followed by a **D** message indicating the end of the operation. Now that many ports can be found in a cell, the *init function* will send a **Y** message for each port in the cell, carrying the corresponding value; and concluding with the **D** message to close the operation.

In a similar way, when the **local transition function** is executed, a cell will send one **Y** message for each value being exported. The coupled model will convert those messages into **X** messages and will deliver them to the cell's neighbors.

4.3.3.2 New Classes

Some classes were created to add support for multiple neighbor ports to CD++. Those classes are described in detail hereafter:

4.3.3.2.1 PortValue

```
typedef pair<string, Real> PortValue;
```

This class is used to store a port's name and a value, representing the value that must be sent through the port after the delay. A list of these structures is returned by the execution of a rule. When working in compatibility mode, the list will have only one structure containing the value to be sent through the port (which can be no other than *out*).

4.3.3.2.2 SendNCPortNode

```
class SendNCPortNode: public SyntaxNode
{
public:
   SendNCPortNode(SyntaxNode *x = NULL, SyntaxNode *y = NULL);
   ~SendNCPortNode();

   SyntaxNode *clone();
   const string name();
   Real evaluate();
   const TypeValue &type() const;
   bool checkType() const;
   ostream &print(ostream &os);

   PortValue getPortValue();

private:
   string PortName();

   SyntaxNode *portName;
```

```
    SyntaxNode *portValue;
};
```

This is a subclass of *SyntaxNode* representing a port assignation. When it is created, two syntax nodes must be provided: one for the port name and the other for the value. If the node for the port name is NULL, then the default value will be used.

The method **PortName** returns a string with the name of the port. If at creation time no name was provided, the returned name is that of the default port: *out*.

The method **evaluate** exists for compatibility reasons, because the abstract super-class requires its existence. It only shows debug information, checks that the port exists, and returns the value that will be sent through the port. The method that is interesting is **getPortValue**. This method returns a structure *PortValue* (see 4.3.3.2.1) with the port's name and the value to be sent through it.

4.3.3.3 Modified Existing Classes

This section describes the changes done to existing classes. In addition to the explanation of those changes, an incomplete description of the class is presented. This incomplete description includes only the methods and members affected by the changes.

4.3.3.3.1 ParallelMainSimulator

```
class ParallelMainSimulator
{
protected:
    ParallelMainSimulator &loadCells(CoupledCell &, bool);
};
```

The method **loadCells** is responsible for loading the cells of a cellular coupled model. It has been extended to read from the model description file the names of the neighbor ports.

It starts by creating an empty list of names, and inserts in the list every port name in the model. If the model does not use extra ports, as is the case for all the models not using the new extensions, this list will be empty but it will exist. This is the way in which the backwards compatibility is guarantee without using code specific to that case.

This list is then passed to the *CoupledCell* object when calling the method *createCells*.

4.3.3.3.2 CoupledCell

```
class CoupledCell : public Coupled
{
public:
    CoupledCell &createCells(const CellPositionList &neighbors,
                             CellPartition *part,
                             list<string> NCPorts);

    CoupledCell &setCellValue(const CellPosition &,
                              const string &,
                              const Real &);
```

```
    CoupledCell &setCellValue(const ModelId &,
                              const string &,
                              Value);

    CoupledCell &setCellValue(const string &sCellPos,
                              const string &,
                              const Real &);

    CoupledCell &setCellAllValues(const ModelId &, Value);
    CoupledCell &setCellAllValues(const CellPosition &,
                                  const Real &);
    CoupledCell &setCellAllValues(const string &sCellPos,
                                  const Real &);
};
```

The method **createCells** has been extended to take as parameter the list of the names of the neighbor ports. After creating each cell, it will invoke *AtomicCell*'s *createNCPorts* method for the cell to create the ports; and will provide the list of names. Notice that the same list is provided to each cell, this guarantees that every cell in the space will have the same ports.

Once the cells and their neighbor ports are created, *createCells* will establish the influences for each cell:

- the port out will influence all the neighbor's *neighborChange* port, and
- the other output neighbor ports will influence the port will the same name in all the neighbor cell.

All the existing methods named **setCellValue** have been improved to receive the port name as a parameter. A new set of methods was created to be used during the initialization. They are named **setCellAllValues** and set the same initial value for all the cell's ports.

4.3.3.3.3 AtomicCell

```
class AtomicCell: public Atomic
{
public:
    static const string NCInPrefix;
    static const string NCOutPrefix;

    void createNCPorts(list<string> &portNames);
    const Real &value(const string &port) const;

protected:
    PortList &inNCPortList();
    PortList &outNCPortList();
    list<string> &NCPorts();
    bool isInNCPort(const string portName) const;
    bool isOutNCPort(const string portName) const;
    AtomicCell &setAllNCPortsValues(const Real &val);
    string calculateInPort(string &portName);
    string calculateOutPort(string &portName);

private:
    bool addInputNCPort(string portName);
```

```
    bool addOutputNCPort(string portName);
    const Port *getNCPortByName(const PortList &pl,
                                const string portName) const;

    PortList     Xports;
    PortList     Yports;
    list<string> NCPortNames;
};
```

In this class, the ancient members **neighborChange** and **out**, which used to store the only available neighbor ports, were replaced by two list of ports. **Xports** stores the input neighbor ports and **Yports** the output neighbor ports. A third member was added. It is named **NCPortsNames** and keeps the list of names of the ports. This list is provided as an argument to the method **createNCPorts**. This method copies the list it received as argument to the *NCPortsNames* member, and creates two ports for each name in the list, one for input and one for output. To differentiate the input ports from the output ports, their names are internally prefixed with the constants **NCInPrefix** and **NCOutPrefix**. In addition to these ports, the default ports are also created: *neighborChange* and *out*. In this way, if an old model is being simulated, the list of names will be empty and the lists *Xports* and *Yports* will only have these two ports respectively, guaranteeing the compatibility.

Two private methods have been added to create the neighbor ports. Their names are **addInputNCPorts** and **addOutputNCPort**. These are the methods used from *createNCPorts* to create the ports. As the *PortList* object identify the ports by their references and not by their names, another private helper method was created to get a pointer to the port reference from of a *PortList*, provided the port name. This method is named **getNCPortByName**.

A few protected methods were created to manipulate the list of ports. The methods **inNCPortList** and **outNCPortList** provide a reference to the list of ports *Xports* and *Yports*, while a reference to the list of names is given by **NCPorts**. To check whether a port belong to the list of input ports is used the method **isInNCPort**, and **isOutNCPort** is used for the output ports. The function **setAllNCPortsValues** is used during the initialization to set the initial vale for all the ports of the cell. And finally the methods **calculateInPort** and **calculateOutPort** get a port name as argument and provide the name of the port input or output port related to that name.

One of the most important changes is that the method **value** was extended to accept the name of the port whose value is required.

4.3.3.3.4 NeighborhoodValue

```
class NeighborhoodValue
{
public:
    typedef map<string, Real> CellPorts;
    typedef map<CellPosition, CellPorts, less < CellPosition > >
            NeighborList;

    NeighborhoodValue& create(const CoupledCell &coupled,
                              const CellPositionList &neighbors,
                              const CellPosition &center,
```

```
                                const list<string> &ports);
   NeighborhoodValue &set(const string, const Real &);
   NeighborhoodValue &set(const NeighborPosition &n,
                          const string &port,
                          const Real &v);

   const Real &get(const NeighborPosition &n,
                   const string &port) const;
   const Real &get(const string &port) const;

   void print(ostream &os);
};
```

This is probably the most affected class. It used to store a value for each neighbor cell. It now stores *n* values for each neighbor cell, being *n* the number of neighbor ports that the cells have. For further information on the reasons that motivated this change, please see 4.3.3.1.2.

The type **NeighborList** has been redefined store, for each neighbor cell, not just one value, but the set of all their values. This type is a *map* that still uses the neighbor cell's relative position as the key, but now the value is no longer a *Real*, but a *CellPorts* object. This new type, **CellPorts**, is defined to be a *map* using the port names as key and to store a *Real* value for each port. This means that from now, to get a value it will be necessary to specify the neighbor cell and the port.

To deal with these changes, the **constructor** was extended to accept the list of port names. It will create an entry in each cell for the default port *neighborChange* as well as for any port in the list. All of them will se assigned the *Undef Real* value.

In addition, the methods **set** and **get** have all been updated to accept the port name as argument, and the **print** method to show correctly all the new the information.

4.3.3.3.5 TransportDelayCell

```
class TransportDelayCell: public AtomicCell
{
protected:
   Model &initFunction();
   Model &externalFunction(const MessageBag &);
   Model &outputFunction(const CollectMessage &);

Private:
   const Real &firstQueuedValue() const;
   const string &firstQueuedPort() const;
};
```

In this class some if the methods associated to the reception of messages needed to be modified. In general, these three methods were updated to no longer use just one value representing the cell's state, but a list of pairs port-value. **initFunction**'s behavior was extended for it to queue a pair port-value containing every port in the cell and its value. Similarly, the **outputFunction** was updated to send a **Y** message for each pair port-value in the queue. And finally, the method **externalFunction** was adapted in the same way. It no

longer uses one value representing the cell's state to determine whether it should be exported, but now it uses the list of pairs port-value queued by the preceding methods.

The private methods **firstQueuedValue** was adapted to cope with *TDCellState*'s new structure, and the method **firstQueuedPort** was added to complete the set of methods to access the information of the first queued value.

4.3.3.3.6 TDCellState

```
class TDCellState : public AtomicCellState
{
public:
      typedef pair< string, Real >       QueueValue;
      typedef pair< VTime, QueueValue > QueueElement;
};
```

Few changes suffered this class representing the state of a *TransportDelayCell*. The type **QueueElement** used to be a pair including the value and its time to leave, representing an element of the queue of values to be sent out. The value was replaced by a pair specifying the value and its departure port. This latter pair is a new type named **QueueValue**. This means that now a *QueueElement* has three components: the time to leave, the departure port and the value.

4.3.3.3.7 InertialDelayCell

```
class InertialDelayCell: public AtomicCell
{
protected:
   Model &initFunction();
   Model &internalFunction(const InternalMessage &);
   Model &externalFunction(const MessageBag &);
   Model &outputFunction(const CollectMessage &);

private:
   Real futureValue(const string &) const;
   void futureValue(const string &, const Real &);

   Real actualValue(const string &) const;
   void actualValue(const string &, const Real &);
};
```

This class included the means to set and get the current and future values of the cell. These methods were respectively named **actualValue** and **futureValue**. These four functions were extended to accept the name of the port.

This change induced changes in other methods of the class. **initFunction** will set the current and future value for every port in the cell, **outputFunction** will send the value through the corresponding port, and **externalFunction** was adapted to consider the port when setting the cell's future values.

4.3.3.3.8 IDCellState

```
class IDCellState : public AtomicCellState
{
public:
   map<string, Real> futureValue;
   map<string, Real> actualValue;
};
```

This cells represents the state of a *InertialDelayCell*. It had two values, the current and the future value, respectively named **actualValue** and **futureValue**, both of class *Real*. These members were modified to be each one a set of pairs port-value.

4.3.3.3.9 LocalTransAdmin

```
class LocalTransAdmin
{
public:
   list<PortValue> evaluate(const string &,
                            const NeighborhoodValue &,
                            PortValues *,
                            VTime &delay,
                            VTime &actualtime,
                            VirtualPortList *,
                            Model *actualCell,
                            string portSource = "");

   const Real &cellValue(const NeighborPosition &,
                         const string PortName);
};
```

Two modification happened to this class. The method **cellValue** now take the port name as its second argument, and the method **evaluate** no longer returns a real but a list of *PortValue*, representing all the values to be exported and their respective departure ports. This list is obtained by the valuation of the local transition function (*SpecNode*).

4.3.3.3.10 VarNode

```
class VarNode: public SyntaxNode
{
public:
   VarNode( nTupla nt, string pName = "" );

   string &port(void);
   VarNode &port(string pName);
};
```

This class represents a node for a reference to a cell's port. The **constructor** now accepts the name of the port as argument. If it is not provided, it will default to "" (the empty string).

Two methods were also added. Their name is **port** and they are used to set and get the port name after the creation of the node. When the store port name is "", the port used will be *out*. This permits defining the default port in this class without having to care for that kind of details in the grammar description.

4.3.3.3.11 CountNode

```
class CountNode: public SyntaxNode
{
public:
   CountNode(const Real &v, StringNode *p = NULL);
   CountNode(SyntaxNode *s, StringNode *p = NULL);

   Real evaluate();

private:
   StringNode *portName;
};
```

This class represents a node for the function **StateCount** of the language. As it was already said in 4.2.2.4, this function was extended to accept a port name as a parameter; and so was this class.

The **constructors** now accepts as argument a *StringNode* representing the name of the port. If it is not provided, it will default to the *StringNode* with the empty string as value (""). The method **evaluate** has been modified to take into account the new argument. When the port name is "", the port used will be *neighborChange*. This allows for defining the default port in this class without having to care for that kind of details in the grammar description.

4.4 Modifications to drawlog

drawlog is an external application used to generate a visual representation of the simulation. It takes as its input the log files generated by CD++. Because having multiple neighbor ports generates new output messages in the log files, it was necessary to update drawlog, to be able to deal with them. A new optional parameter was added to this tool: **-n<port>**.

```
$ drawlog -h
drawlog -[?hmtclwp0n]

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

If it is not present, drawlog will behave in compatibility mode, drawing the changes exported by the port *out* of the cells. When it is present, it takes as argument the name of the port to draw.

5 Application Examples

This section shows some examples of models that where adapted to use the new capabilities of CD++. The full code of those examples can be found in the appendixes, while in this section there are only the code extracts necessary to understand how those conversion were achieved.

5.1 Generic Comparison – Life Game

For a generic comparison it was decided to use the Life Game, a model that needed very few modifications to use CD++ new capabilities.

Five examples were tested:

- **Original:** the unmodified model using the unmodified simulator,
- **Compatible:** the unmodified model using the new simulator,
- **State Variables:** the model modified to use state variables using the new simulator,
- **Default ports:** the model modified to use explicitly the default neighbor ports, and
- **Non-default ports:** the model modified to use non-default neighbor ports.

All the test cases had cell spaces of 21 x 21 cells and started with the same initial state. All these simulations were executed three times on the same computer with no external load. The model descriptions can be found in Appendix C – Life Game.

What was observed in these tests was the results of the simulation, which were requested to be equivalents; the duration of the simulations; and the percentage of CPU usage. To normalize the results and verify their equivalence, the log files were processed by **drawlog**: *equivalent result should produce the same "visualization."* As expected, all the test produced the same "visualization."

The following table show the results from the test runs. Each cell shows the simulation duration expressed in seconds, and the CPU load expressed as a percentage.

	1st Run		2nd Run		3rd Run		Average	
	Duration	Load	Duration	Load	Duration	Load	Duration	Load
Original	6.70	87	6.73	87	6.68	88	6.70	87.33
Compatible	9.95	86	9.62	89	9.78	87	9.78	87.33
State Vars	9.22	91	9.77	86	9.24	90	9.41	89.00
Def. Ports	9.61	89	9.60	89	9.65	89	9.62	89.00
Non-def Ports	10.00	90	10.08	89	10.35	87	10.14	88.66

A comparison of these values can be seen in the following graphs. The first one shows the simulation durations and the second one shows the percentage of CPU load.

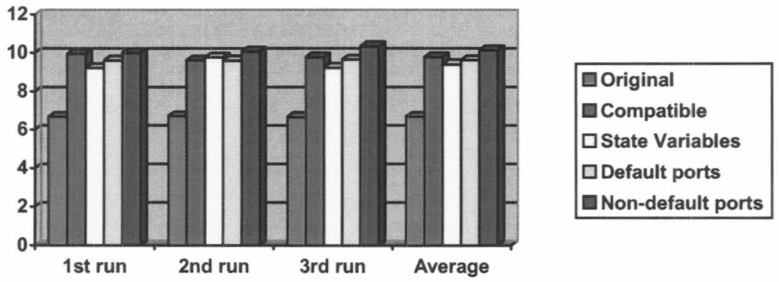


Figure 11: Comparison of the duration times for life game

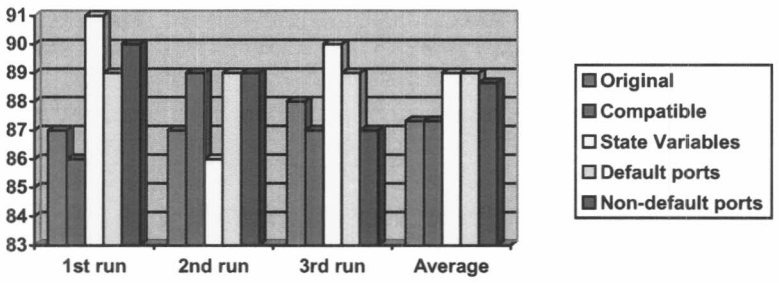


Figure 12: Comparison of the CPU loads for the life game

From these results, it can be concluded that the original simulator performs better than the new one. As strange as it can seem, it is natural: with the new capabilities came an increased overhead. Between the *Original* test and the *Compatible* test, the overhead increased about 45%, but also it must be noticed that the Life Game simulation did not need the new capabilities and their usage was force in an unnatural way.

5.2 State variables – Fire Spread

The current version of the Fire Spread model [Aie01] does not use state variables because they were not available at the moment of its writing. Instead it uses a three-dimensional cell space. Two dimensions are used to represent the field where the fire spreads, whilst the third one is used for technical reasons: the need to keep two values in each cell. As it was impossible to have more than one value in a cell, the modelers stacked cells to store one value in each one, and adapted the model to treat them as if they were just one cell. As of today, the cell space used is of $\mathbf{n} \times \mathbf{m} \times \mathbf{2}$, where $\mathbf{n} \times \mathbf{m}$ is the dimension of the simulated field. The lower layer of cells is used to store the *temperature* and the higher layer to store the *ignition time*. The models can be seen in Appendix B – Fire Spread.

5.2.1 Model Conversion

Using the new simulator, which supports multiple state variables, this trick is not longer needed. The *temperature* is stored as the cell's value and the *ignition time* in a state variable. In the new model this variable is named *ti* because it is the name that the original modelers used to reference this value in the higher layer of cells. The *temperature* is stored in the cell's value because it must be passed to the neighbor cells, while the *ti* value is only used internally to the cell. In this way, now the model has only two dimensions.

The first step was to add the state variable *ti*, to remove the higher layer of cells and to replace all the references to this layer by references to the state variable. In a simpler model this could have been enough, but it was not the case here. A problem appeared when converting the rules "Burning" and "ti". These are the original rules:

```
%Burning
rule : { #macro(burning) } 1 { cellpos(2) = 0 AND ( ( (0,0,0) >
#macro(burning) AND (0,0,0) > 333 ) OR ( #macro(burning) >
(0,0,0) AND (0,0,0) >= 573 ) ) AND (0,0,0) != 209 }

%ti
rule : { time / 100 } 1 { cellpos(2) = 1 AND (0,0,-1) >= 573 AND
(0,0,0) = 1.0 }
```

The "ti" rule is applied only to the higher layer, while the "burning" rule is applied to the lower layer. To make this happen the conditions include a clause specifying to which layer they apply. This clause is **cellpos(2)=*x***, which disappeared when the references to the higher layer were removed:

```
%Burning
rule : { #macro(burning) } 1 { ( ( (0,0) > #macro(burning) AND
(0,0) > 333 ) OR ( #macro(burning) > (0,0) AND (0,0) >= 573 ) )
AND (0,0) != 209 }

%ti
rule : { (0,0) } { $ti := time / 100; } 1 { (0,0) >= 573 AND $ti
= 1.0 }
```

The problem now is that in some cases, both conditions can be true at the same time. For instance, when $ti = 1.0, (0,0) >=573 and #macro(burning) > (0,0).

To solve this problem, the rule "burning" was factorized into two simpler rules, eliminating the OR operation:

```
%Burning
rule : { #macro(burning) } 1 { (0,0) > #macro(burning) AND
                               (0,0) >  333 AND (0,0) != 209 }
rule : { #macro(burning) } 1 { #macro(burning) > (0,0) AND
                               (0,0) >= 573 AND (0,0) != 209 }
```

Now, it is possible to see that in both rules it is requested that (0,0) != 209. But it is also requested that (0,0) is higher than a value which is higher than 209. Then:

$$(0,0) > 333 \Rightarrow (0,0) \neq 209$$
$$(0,0) \geq 573 \Rightarrow (0,0) \neq 209$$

From this can be concluded that (0,0) != 209 is a redundant request, and so it was removed.

```
%Burning
rule : { #macro(burning) } 1 { (0,0) > #macro(burning) AND
```

```
                                    (0,0) > 333 }
rule : { #macro(burning) } 1 { #macro(burning) > (0,0) AND
                                    (0,0) >= 573 }
```

This removal is not mandatory. The model will behave the same if keep this condition is kept, but removing it will simplify the following operations.

Now, the rule "ti" only overlaps with the second part of the rule "burning", so they were merged. This generated three rules that replace the previous two:

```
%Burning and ti
rule : { #macro(burning) } 1 { (0,0) > #macro(burning) AND
                                    (0,0) > 333 }
rule : { #macro(burning) }                              1
       { #macro(burning) > (0,0) AND (0,0) >= 573 AND
         $ti != 1.0 }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { #macro(burning) > (0,0) AND (0,0) >= 573 AND
         $ti = 1.0 }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { #macro(burning) < (0,0) AND (0,0) >= 573 AND
         $ti = 1.0 }
```

The third and fourth part of the rule modify ti's value when(0,0) >= 573 and $ti = 1.0, as requested by the original rule ti, regardless of the value of #macro(burning).

The second and third part set #macro(burning) as the cell's new value when (0,0) >= 573 and #macro(burning) > (0,0), as requested by the original rule "burning".

However, after these modifications, the first and fourth part overlap when $ti = 1.0 because $(0,0) \geq 573 \Rightarrow (0,0) > 333$. This means that the first part's condition must be restricted to prevent this collision to happen:

```
%Burning
rule : { #macro(burning) } 1 { (0,0) > #macro(burning) AND
                                    (0,0) > 333 AND
                                    ((0,0) < 573 OR $ti != 1.0) }
rule : { #macro(burning) }                              1
       { #macro(burning) > (0,0) AND (0,0) >= 573 AND
         $ti != 1.0 }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { #macro(burning) > (0,0) AND (0,0) >= 573 AND
         $ti = 1.0 }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { #macro(burning) < (0,0) AND (0,0) >= 573 AND
         $ti = 1.0 }
```

Now the first rule's condition is true only when *(0,0) < 573 or $ti != 1.0*, which makes the fourth rule's condition false.

However, this new model is far from being "optimal" in its execution. To shorten the execution time, the number of rules can be reduced and the clauses in the rules' condition can be reordered.

To reduce the number of rules, some of them can be merged. For instance the following rules are very similar:

```
rule : { #macro(burning) }                              1
       { #macro(burning) > (0,0) AND (0,0) >= 573 AND
         $ti != 1.0 }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { #macro(burning) > (0,0) AND (0,0) >= 573 AND
         $ti = 1.0 }
```

It can be seen that they only differ in the required value for *$ti*, and in the assignation (or not) of a new value to *$ti*. These two rules can be merged in just one rule that will assign the new value to *$ti* depending on *$ti*'s original value:

```
rule : { #macro(burning) }
                  { $ti := if($ti = 1.0, time / 100, $ti); } 1
                  { (0,0) >= 573 AND #macro(burning) >= (0,0) }
```

For the second step it will be used the fact that CD++ is capable of using short-cut evaluation (in the same style as the C programming language). When the left expression of an **and** operation evaluates to false, the whole operation will evaluate to *false*, so it is useless to evaluate the right expression. Similarly, when the left expression of an **or** operation evaluates to *true*, the whole operation will evaluate to true, and so there is no need to evaluate the right expression of the operation.

By simply reordering the operations and their parameter in the rules' condition, a lot of execution time can be saved. The trick is to make execute first the simplest conditions, while leaving to the end the more complex ones. Moving to the left the simplest operation will make the deal.

```
%Unburned
rule : { #macro(unburned) } 1 { (0,0) != 209 AND (0,0) < 573 AND
                                ( time <= 20 OR
                                  #macro(unburned) > (0,0) ) }
%Burning and ti
rule : { #macro(burning) } 1 { (0,0) > 333 AND
                               ( (0,0) < 573 OR $ti != 1.0 ) AND
                               (0,0) > #macro(burning) }
rule : { #macro(burning) }
                  { $ti := if($ti = 1.0, time / 100, $ti); } 1
                  { (0,0) >= 573 AND #macro(burning) >= (0,0) }
rule : { #macro(burning) } { $ti := time / 100; } 1
                             { $ti = 1.0 AND (0,0) >= 573 AND
                               #macro(burning) < (0,0) }
%Burned
rule : { 209 } 100 { (0,0) != 209 AND (0,0) <= 333 AND
                     (0,0) > #macro(burning) }
```

5.2.2 Comparison

This section compares the three models' performance. All of them were executed with the same initial values and until the simulation finished by itself.

The first great difference is the visualization of the results using the tool *drawlog*. The original model shows the information for both layers:

```
Line : 1 - Time: 00:00:00.000
        0   1   2   3   4   5          0   1   2   3   4   5
    +-----------------------+      +-----------------------+
  0| 300 300 300 300 300 300|    0|   0   0   0   0   0   0|
  1| 300 600 600 300 300 300|    1|   0   1   1   0   0   0|
  2| 300 600 600 300 300 300|    2|   0   1   1   0   0   0|
  3| 300 300 300 300 300 300|    3|   0   0   0   0   0   0|
  4| 300 300 300 300 300 300|    4|   0   0   0   0   0   0|
  5| 300 300 300 300 300 300|    5|   0   0   0   0   0   0|
    +-----------------------+      +-----------------------+
```

While the new models using state variables only show one layer, the only one existing:

```
Line : 1 - Time: 00:00:00.000
        0   1   2   3   4   5
    +-----------------------+
  0| 300 300 300 300 300 300|
  1| 300 600 600 300 300 300|
  2| 300 600 600 300 300 300|
  3| 300 300 300 300 300 300|
  4| 300 300 300 300 300 300|
  5| 300 300 300 300 300 300|
    +-----------------------+
```

For this comparison, four examples were tested:

- **Original:** the unmodified model using the unmodified simulator,
- **Compatible:** the unmodified model using the new simulator,
- **State Variables:** the model modified to use state variables, and
- **Optimized:** the optimized model modified to use state variables.

All the test cases had cell spaces of 6 x 6 cells and started with the same initial state. All these simulations were executed three times on the same computer with no external load. What was observed in these tests was the duration of the simulations and the percentage of CPU usage.

The following table shows the results from the test runs. Each cell shows the simulation duration expressed in seconds, and the CPU load expressed as a percentage.

	1st Run		2nd Run		3rd Run		Average	
	Duration	Load	Duration	Load	Duration	Load	Duration	Load
Original	66.42	74	67.94	72	66.13	75	66.83	73.67
Compatible	86.53	80	86.69	80	87.19	80	86.80	80.00
State Vars	84.03	81	84.78	80	84.88	80	84.56	80.33
Optimized	63.80	75	64.60	74	64.07	75	64.16	74.67

A comparison of these values can be seen in the following graphs. The first one shows the simulation durations and the second one shows the percentage of CPU load.

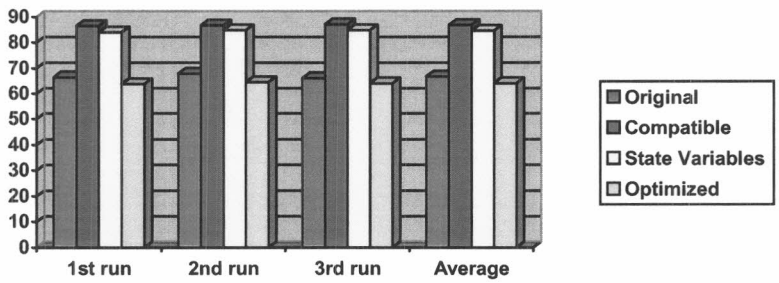


Figure 13: Comparison of the durations for Fire Spread using State Variables

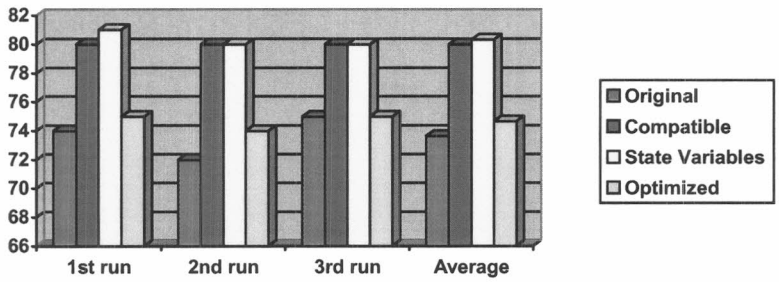


Figure 14: Comparison of the CPU loads for Fire Spread using State Variables

It can be concluded from these results that the original simulator performed better that the new simulator for the compatible model and the un-optimized model using state variables, but in comparison with the *Life Games* results (see 5.1), this time the increase was just 9%.

On the contrary, the optimized model performed slightly better than the original simulator. The reason for this is that, this model has half of the cells than the original model (one layer was removed by using state variables), and because of the optimizations.

In conclusion, models using multiple state variables take as much time to simulate as the previous simulator; but now models are more natural because there is no need for an inexplicable second layer of cells, which increases the ease of writing, reading and maintaining; and use less resources (such as memory and file descriptors), because there are half of the cells to manage.

5.3 Multiple neighbor Ports – Fire Spread

As explained in 5.2, the Fire Spread model stacked cell layers to simulate the storage of multiple values in one cell. Using multiple neighbor ports can also solve this problem.

5.3.1 Model Conversion

The conversion from the original Fire Spread model to the one using neighbor ports is very similar to the conversion to use state variables exposed in 5.2.1. Because of this, the conversion will not be explained step by step, but only the specific differences.

In this case, two ports are declared: *temp* and *ti*.

```
neighborports: temp ti
```

The port temp exports the cell's *temperature* (the old lower layer), while the port *ti* exports the *ignition time* (the higher layer).

The rules generated are equivalent to those used in the state variables model, but instead they use ports:

```
%Unburned
rule : { ~temp := #macro(unburned); } 1
       { ((#macro(unburned)) > (0,0)~temp OR time <= 20 ) AND
         (0,0)~temp < 573 AND (0,0)~temp != 209 AND
         (0,0)~temp > 0 }

%Burning and ti
rule : { ~temp := #macro(burning); }                    1
       { (0,0)~temp > #macro(burning) AND
         (0,0)~temp > 333 AND
         ((0,0)~temp < 573 OR (0,0)~ti != 1.0) }
rule : { ~temp := #macro(burning); }                    1
       { #macro(burning) >= (0,0)~temp AND
         (0,0)~temp >= 573 AND (0,0)~ti != 1.0 }
rule : { ~temp := #macro(burning); ~ti := time / 100; } 1
       { #macro(burning) >= (0,0)~temp AND
         (0,0)~temp >= 573 AND (0,0)~ti = 1.0 }
rule : { ~temp := #macro(burning); ~ti := time / 100; } 1
       { #macro(burning) < (0,0)~temp AND
         (0,0)~temp >= 573 AND (0,0)~ti = 1.0 }

%Burned
rule : { ~temp := 209; } 100 { (0,0)~temp > #macro(burning) AND
                               (0,0)~temp <= 333 AND
                               (0,0)~temp != 209 AND
                               (0,0)~temp > 0 }
%Stay Burned or constant
rule : {  } 1 { t }
```

As the initial value for both ports is the same and this model needs different values, it was solved by assigning initial value that will never appear during the simulation and adding two rules that generate the real initial state when the cell has this special values.

The special values are **–1** and **–2**. Thus the initial value for the ports is *-1*and the cells that start with a different initial value will have *-2*.

```
initialValue : -1
initialCellsValue : init.val
```

The contents of the file init.val are:

```
(1,1) = -2
(2,1) = -2
(1,2) = -2
(2,2) = -2
```

And the rules to generate the real initial state from this values are:

```
%initialization
rule : { ~temp := 300; ~ti := 0; } 1
       { (0,0)~temp = -1 and (0,0)~ti = -1 }
rule : { ~temp := 600; ~ti := 1; } 1
       { (0,0)~temp = -2 and (0,0)~ti = -2 }
```

These rules are placed in the last places to minimize their interference counting on the fact that CD++ evaluates the rules in order.

This model can be optimized in a way similar to that used for the state variables model. These are the rules after the optimization.

```
%Unburned
rule : { ~temp := #macro(unburned); } 1
       { (0,0)~temp > 0 AND (0,0)~temp != 209 AND
         (0,0)~temp < 573 AND
         (time <= 20 OR (#macro(unburned)) > (0,0)~temp ) }

%Burning and ti
rule : { ~temp := #macro(burning); }                          1
       { (0,0)~temp > 333 AND
         ( (0,0)~temp < 573 OR (0,0)~ti != 1.0 ) AND
         (0,0)~temp > #macro(burning) }
rule : { ~temp := #macro(burning);
         ~ti := if( (0,0)~ti = 1.0, time / 100, (0,0)~ti); } 1
       { (0,0)~temp >= 573 AND #macro(burning) >= (0,0)~temp }
rule : { ~temp := #macro(burning); ~ti := time / 100; }       1
       { (0,0)~ti = 1.0 AND (0,0)~temp >= 573 AND
         #macro(burning) < (0,0)~temp }
%Burned
rule : { ~temp := 209; } 100 { (0,0)~temp != 209 AND
                               (0,0)~temp > 0 AND
                               (0,0)~temp <= 333 AND
                               (0,0)~temp > #macro(burning) }
```

5.3.2 Comparison

This section compares the three models' performance. All of them were executed with the same initial values and until the simulation finished by itself.

For this comparison, four examples were tested:

- **Original:** the unmodified model using the unmodified simulator,
- **Compatible:** the unmodified model using the new simulator,
- **Ports:** the model modified to use neighbor ports, and
- **Optimized:** the optimized model modified to use neighbor ports.

All the test cases had cell spaces of 6 x 6 cells and started with the same initial state. All these simulations were executed three times on the same computer with no external load. What was observed in these tests was the duration of the simulations and the percentage of CPU usage. The Compatible model is the same as for the state variables example. It was re-used here to easy the comparison.

The following table shows the results from the test runs. Each cell shows the simulation duration expressed in seconds, and the CPU load expressed as a percentage.

	1st Run		2nd Run		3rd Run		Average	
	Duration	Load	Duration	Load	Duration	Load	Duration	Load
Original	66.42	74	67.94	72	66.13	75	66.83	73.67
Compatible	86.53	80	86.69	80	87.19	80	86.80	80.00
Ports	81.30	79	81.11	80	80.90	80	80.10	79.67
Optimized	65.41	73	63.04	76	64.92	73	64.46	74.00

A comparison of these values can be seen in the following graphs. The first one shows the simulation durations and the second one shows the percentage of CPU load.

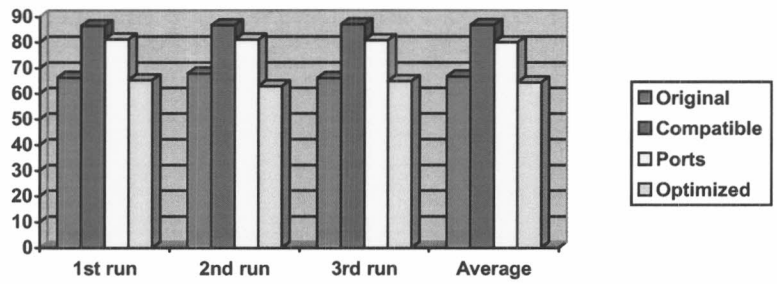


Figure 15: Comparison of the durations for Fire Spread using Neighbor Ports

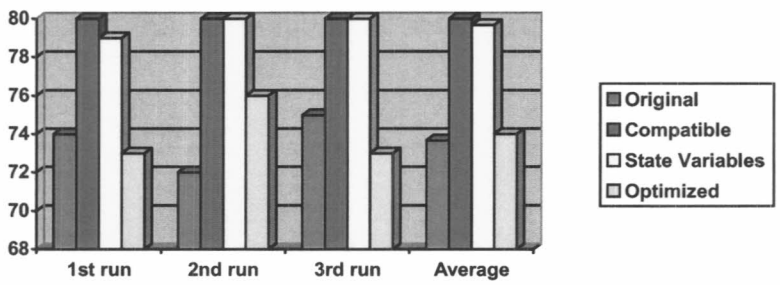


Figure 16: Comparison of the CPU loads for Fire Spread using Neighbor Ports

It can be concluded from these results that, in a similar way to what happened with the state variables, the original simulator performed better that the new simulator for the compatible model and the un-optimized model using neighbor ports. In this case the increase in duration was about 9%.

Again, the optimized model performed slightly better than the original simulator. The reason for this is that, this model has half of the cells than the original model (one layer was removed by using state variables), and because of the optimizations.

As with the state variables, models using multiple neighbor ports take as much time to simulate as the previous simulator; but now models are more natural because there is no need for an inexplicable second layer of cells, which increases the ease of writing, reading and maintaining; and use less resources (such as memory and file descriptors), because there are half of the cells to manage.

6 Conclusions And Future Work

The new implementation of CD++ was presented. This implementation includes two new features that were missing so far. These features are *state variables* and *multiple neighbor ports*. To achieve this, the state of the cells was extended to include a set of state variables and their values, and the only value arriving from neighbor cells was replaced by a set of values arriving through different ports.

These new features add great power to the specification language, and thus to the simulator; simplifying the modeling task. But with every improvement there is always a price to be paid. The price is an increased overhead required for the management of these features, making the simulations longer. It was also shown that when the model is optimized, this overhead can be nullified and even inversed, but this means that modelers will have to pay more attention to model optimization.

Nevertheless, the models can now be written more clearly, without the need of tricks like extra layers of cells; and their simulation consume less memory and file descriptors (than those models with extra cell layers), which allow for larger cell spaces to be simulated.

There are a few topics were CD++ can be improved:

- Allow all the neighbor ports to be initialized with different values. Nowadays, all the neighbor ports of a cell are initialized with the same value. Removing this limitation will simplify the modeler's work.
- After its first version, CD++ suffered many modifications and improvements, most of which the developers could not even dream of. Some of these improvement made obsolete some parts of the code that were good enough for the old versions. For instance, the lexical analyzer (which was designed to recognize a very reduced and strict language) is today imposing too many restrictions to the extensions of the language.
- Go on improving CD++ getting every day closer to the whole formalism.

7 Appendix A - Grammar

```
RuleList = Rule | Rule RuleList

Rule = AssignResult Result { BoolExp }
     | AssignResult { AssignSet } Result { BoolExp }

AssignResult = Result | { PortSendSet }

Result = Constant | UNDEF | { RealExp }

BoolExp = BOOL | ( BoolExp ) | RealRelExp | NOT BoolExp
        | BoolExp BOOL_OP

RealRelExp = RealExp REL_OP RealExp
           | COND_REAL_FUNC ( RealExp )

RealExp = IdRef | ( RealExp ) | RealExp OPER RealExp

IdRef = CellRef OptPortName | Constant | Function
      | UNDEF | PORTREF ( PORTNAME )
      | SEND ( PORTNAME , RealExp )
      | CELLPOS ( RealExp ) | STVAR_NAME

OptPortName = /* Empty */ | ~ PORTNAME

AssignSet = /* Empty */ | Assign AssignSet

Assign = STVAR_NAME ASSIGN_OP RealExp ;

PortSendSet = /* Empty */ | PortSend PortSendSet

PortSend = SEND ( PORTNAME , RealExp ) ;
         | ~ PORTNAME ASSIGN_OP RealExp ;

Constant = INT | REAL | CONSTFUNC

Function = COUNT
         | STATECOUNT ( RealExp OptParamPort )
         | UNARY_FUNC ( RealExp )
         | BINARY_FUNC ( RealExp , RealExp )
         | WITHOUT_PARAM_FUNC_TIME
         | WITHOUT_PARAM_FUNC_RANDOM
         | UNARY_FUNC_RANDOM ( RealExp )
         | BINARY_FUNC_RANDOM ( RealExp , RealExp )
         | COND3_FUNC ( BoolExp , RealExp , RealExp )
         | COND4_FUNC ( BoolExp , RealExp , RealExp , RealExp )

OptParamPort = /* Empty */ | , ~ PORTNAME

CellRef = ( Tuple

Tuple = INT , INT Rest_nTuple
```

```
Rest_nTuple = , INT Rest_nTuple | )

BOOL = t | f | ?

REL_OP = != | = | > | < | >= | <=

BOOL_OP = and | or | xor | imp | eqv

ASSIGN_OP = :=

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

STVAR_NAME = $ 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 | hip |
            | rectToPolar_angle | polarToRect_x | polarToRect_y

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

8 Appendix B – Fire Spread

8.1 Original

```
#include(rules.inc)

[top]
components : ForestFire

[ForestFire]
type : cell
dim : (6,6,2)
delay : transport
defaultDelayTime : 1000
border : nowrapped
neighbors : ForestFire(-1,0,0) ForestFire(0,-1,0)
neighbors : ForestFire(1,0,0)  ForestFire(0,1,0)
neighbors : ForestFire(0,0,0)
neighbors : ForestFire(0,0,-1) ForestFire(0,0,1)
initialValue : 300.0
initialCellsValue : init.val
localTransition : FireBehavior

[FireBehavior]
%Unburned
rule : { #macro(unburned) } 1
       { cellpos(2) = 0 and
         ( #macro(unburned) > (0,0,0) OR time <= 20 ) AND
        (0,0,0) < 573 AND (0,0,0) != 209 }

%ti
rule : { time / 100 } 1 { cellpos(2) = 1 AND
                          (0,0,-1) >= 573 AND (0,0,0) = 1.0 }

%Burning
rule : { #macro(burning) } 1
       { cellpos(2) = 0 AND
         ( ( (0,0,0) > #macro(burning) AND(0,0,0) > 333 ) OR
           ( #macro(burning) > (0,0,0) AND (0,0,0) >= 573 )) AND
         (0,0,0) != 209 }

%Burned
rule : { 209 } 100 { cellpos(2) = 0 AND
                     (0,0,0) > #macro(burning) AND
                     (0,0,0) <= 333 AND (0,0,0) != 209 }

%Stay Burned or constant
rule : { (0,0,0) } 1 { t }
```

And these are the macros used in this model, which are declared in the file rules.inc.

```
#BeginMacro(unburned)
( 0.98689 * (0,0,0)
+ 0.0031 * (0,-1,0)
+ 0.0031 * (0,1,0)
+ 0.0031 * (1,0,0)
+ 0.0031 * (-1,0,0)
+ 0.213 )
#EndMacro

#BeginMacro(burning)
( 0.98689 * (0,0,0)
+ 0.0031 * (0,-1,0)
+ 0.0031 * (0,1,0)
+ 0.0031 * (1,0,0)
+ 0.0031 * (-1,0,0)
+ 2.74 * exp ( -0.19 * ( ( time + 1 ) / 100 - (0,0,1) ) )
+ 0.213 )
#EndMacro
```

8.2 State Variables

```
#include(rules.inc)

[top]
components : ForestFire

[ForestFire]
type : cell
dim : (6,6)
delay : transport
defaultDelayTime : 1000
border : nowrapped
neighbors : ForestFire(-1,0) ForestFire(0,-1) ForestFire(1,0)
neighbors : ForestFire(0,1)  ForestFire(0,0)
initialValue : 300
initialCellsValue : init.val
stateVariables: ti
stateValues: 0
initialVariablesValue: var.val
localTransition : FireBehavior

 [FireBehavior]
%Unburned
rule : { #macro(unburned) } 1 { ( #macro(unburned) > (0,0) OR
                                  time <= 20 ) AND
                                (0,0) < 573 AND
                                (0,0) != 209 }

%Burning and ti
rule : { #macro(burning) } 1 { (0,0) > #macro(burning) AND
                               (0,0) > 333 AND
                               ( (0,0) < 573 OR $ti != 1.0 ) }
rule : { #macro(burning) } 1 { #macro(burning) >= (0,0) AND
                              (0,0) >= 573 AND $ti != 1.0 }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { #macro(burning) >= (0,0) AND (0,0) >= 573 AND
         $ti = 1.0 }
rule : { #macro(burning) } { $ti := time / 100; } 1
       { #macro(burning) <  (0,0) AND (0,0) >= 573 AND
         $ti = 1.0 }


%Burned
rule : { 209 } 100 { (0,0) > #macro(burning) AND
                     (0,0) <= 333 AND (0,0) != 209 }

%Stay Burned or constant
rule : { (0,0) } 1 { t }
```

The macros have also been modified.

```
#BeginMacro(unburned)
( 0.98689 * (0,0)
+ 0.0031 * (0,-1)
+ 0.0031 * (0,1)
+ 0.0031 * (1,0)
+ 0.0031 * (-1,0)
+ 0.213 )
#EndMacro

#BeginMacro(burning)
( 0.98689 * (0,0)
+ 0.0031 * (0,-1)
+ 0.0031 * (0,1)
+ 0.0031 * (1,0)
+ 0.0031 * (-1,0)
+ 2.74 * exp ( -0.19 * ( ( time + 1 ) / 100 - $ti ) )
+ 0.213 )
#EndMacro
```

8.3 State Variables Optimized

```
#include(rules.inc)

[top]
components : ForestFire

[ForestFire]
type : cell
dim : (6,6)
delay : transport
defaultDelayTime : 1000
border : nowrapped
neighbors : ForestFire(-1,0) ForestFire(0,-1) ForestFire(1,0)
neighbors : ForestFire(0,1)  ForestFire(0,0)
initialValue : 300
initialCellsValue : init.val
localTransition : FireBehavior
stateVariables: ti
stateValues: 0
initialVariablesValue: var.val

[FireBehavior]
%Unburned
rule : { #macro(unburned) } 1 { (0,0) != 209 AND (0,0) < 573 AND
                                ( time <= 20 OR
                                  #macro(unburned) > (0,0) ) }
%Burning and ti
rule : { #macro(burning) } 1 { (0,0) > 333 AND
                               ( (0,0) < 573 OR $ti != 1.0 ) AND
                               (0,0) > #macro(burning) }
rule : { #macro(burning) }
                    { $ti := if($ti = 1.0, time / 100, $ti); } 1
                    { (0,0) >= 573 AND #macro(burning) >= (0,0) }
rule : { #macro(burning) } { $ti := time / 100; } 1
                             { $ti = 1.0 AND (0,0) >= 573 AND
                               #macro(burning) < (0,0) }

%Burned
rule : { 209 } 100 { (0,0) != 209 AND (0,0) <= 333 AND
                     (0,0) > #macro(burning) }

%Stay Burned or constant
rule : { (0,0) } 1 { t }
```

The macros have not been affected by this optimization. This means that they are the same macros exposed in 8.2.

8.4 Neighbor Ports

```
#include(rules.inc)

[top]
components : ForestFire

[ForestFire]
type : cell
dim : (6,6)
delay : transport
defaultDelayTime : 1000
border : nowrapped
neighbors : ForestFire(-1,0) ForestFire(0,-1) ForestFire(1,0)
neighbors : ForestFire(0,1)  ForestFire(0,0)
initialValue : -1
initialCellsValue : init.val
neighborports: temp ti
localTransition : FireBehavior

[FireBehavior]
%Unburned
rule : { ~temp := #macro(unburned); } 1
       { ((#macro(unburned)) > (0,0)~temp OR time <= 20 ) AND
         (0,0)~temp < 573 AND (0,0)~temp != 209 AND
         (0,0)~temp > 0 }

%Burning and ti
rule : { ~temp := #macro(burning); }                    1
       { (0,0)~temp > #macro(burning) AND
         (0,0)~temp > 333 AND
         ((0,0)~temp < 573 OR (0,0)~ti != 1.0) }
rule : { ~temp := #macro(burning); }                    1
       { #macro(burning) >= (0,0)~temp AND
         (0,0)~temp >= 573 AND (0,0)~ti != 1.0 }
rule : { ~temp := #macro(burning); ~ti := time / 100; } 1
       { #macro(burning) >= (0,0)~temp AND
         (0,0)~temp >= 573 AND (0,0)~ti = 1.0 }
rule : { ~temp := #macro(burning); ~ti := time / 100; } 1
       { #macro(burning) < (0,0)~temp AND
         (0,0)~temp >= 573 AND (0,0)~ti = 1.0 }

%Burned
rule : { ~temp := 209; } 100 { (0,0)~temp > #macro(burning) AND
                               (0,0)~temp <= 333 AND
                               (0,0)~temp != 209 AND
                               (0,0)~temp > 0 }
%initialization
rule : { ~temp := 300; ~ti := 0; } 1
       { (0,0)~temp = -1 and (0,0)~ti = -1 }
rule : { ~temp := 600; ~ti := 1; } 1
       { (0,0)~temp = -2 and (0,0)~ti = -2 }

%Stay Burned or constant
rule : {  } 1 { t }
```

The macros have also been modified

```
#BeginMacro(unburned)
( 0.98689 * (0,0)~temp
+ 0.0031 * (0,-1)~temp
+ 0.0031 * (0,1)~temp
+ 0.0031 * (1,0)~temp
+ 0.0031 * (-1,0)~temp
+ 0.213 )
#EndMacro

#BeginMacro(burning)
( 0.98689 * (0,0)~temp
+ 0.0031 * (0,-1)~temp
+ 0.0031 * (0,1)~temp
+ 0.0031 * (1,0)~temp
+ 0.0031 * (-1,0)~temp
+ 2.74 * exp ( -0.19 * ( ( time + 1 ) / 100 - (0,0)~ti ) )
+ 0.213 )
#EndMacro
```

8.5 Neighbor Ports Optimized

```
#include(rules.inc)

[top]
components : ForestFire

[ForestFire]
type : cell
dim : (6,6)
delay : transport
defaultDelayTime : 1000
border : nowrapped
neighbors : ForestFire(-1,0) ForestFire(0,-1) ForestFire(1,0)
neighbors : ForestFire(0,1)  ForestFire(0,0)
initialValue : -1
initialCellsValue : init.val
neighborports: temp ti
localTransition : FireBehavior

[FireBehavior]
%Unburned
rule : { ~temp := #macro(unburned); } 1
       { (0,0)~temp > 0 AND (0,0)~temp != 209 AND
         (0,0)~temp < 573 AND
         (time <= 20 OR (#macro(unburned)) > (0,0)~temp ) }

%Burning and ti
rule : { ~temp := #macro(burning); }                          1
       { (0,0)~temp > 333 AND
         ( (0,0)~temp < 573 OR (0,0)~ti != 1.0 ) AND
         (0,0)~temp > #macro(burning) }
rule : { ~temp := #macro(burning);
         ~ti := if( (0,0)~ti = 1.0, time / 100, (0,0)~ti); } 1
       { (0,0)~temp >= 573 AND #macro(burning) >= (0,0)~temp }
rule : { ~temp := #macro(burning); ~ti := time / 100; }       1
       { (0,0)~ti = 1.0 AND (0,0)~temp >= 573 AND
         #macro(burning) < (0,0)~temp }
%Burned
rule : { ~temp := 209; } 100 { (0,0)~temp != 209 AND
                               (0,0)~temp > 0 AND
                               (0,0)~temp <= 333 AND
                               (0,0)~temp > #macro(burning) }
%initialization
rule : { ~temp := 300; ~ti := 0; } 1
       { (0,0)~temp = -1 and (0,0)~ti = -1 }
rule : { ~temp := 600; ~ti := 1; } 1
       { (0,0)~temp = -2 and (0,0)~ti = -2 }

%Stay Burned or constant
rule : {  } 1 { t }
```

The same macros exposed in 8.4 are used in the model.

9 Appendix C – Life Game

9.1 Original And Compatibility

```
[top]
components : life

[life]
type : cell
width : 21
height : 21
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
initialvalue : 0
initialrowvalue : 4  000011100000000000000
initialrowvalue : 5  000011100010000000000
initialrowvalue : 6  000011100110000000000
initialrowvalue : 10 000000000111000000000
initialrowvalue : 11 000000000111000100000
initialrowvalue : 12 000000000111001100000
initialrowvalue : 14 000000000000001110000
initialrowvalue : 15 000000000000001110001
initialrowvalue : 16 000000000000001110011
localtransition : conrad-rule

[conrad-rule]
rule : 1 100 { (0,0) = 1 and (truecount = 3 or truecount = 4) }
rule : 0 100 { (0,0) = 1 and (truecount < 3 or truecount > 4) }
rule : 1 100 { (0,0) = 0 and truecount = 3 }
rule : 0 100 { (0,0) = 0 and truecount != 3 }
```

9.2 State Variables

```
[top]
components : life

[life]
type : cell
width : 21
height : 21
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
initialvalue : 0
initialrowvalue : 4  00001110000000000000
initialrowvalue : 5  00001110001000000000
initialrowvalue : 6  00001110011000000000
initialrowvalue : 10 00000000111000000000
initialrowvalue : 11 00000000111000100000
initialrowvalue : 12 00000000111001100000
initialrowvalue : 14 00000000000001110000
initialrowvalue : 15 00000000000001110001
initialrowvalue : 16 00000000000001110011
localtransition : conrad-rule
statevariables: value
statevalues: 0
initialvariablesvalue: life.stvalues

[conrad-rule]
rule : { $value } { $value := 1; } 100
       { $value = 1 and (truecount = 3 or truecount = 4) }
rule : { $value } { $value := 0; } 100
       { $value = 1 and (truecount < 3 or truecount > 4) }
rule : { $value } { $value := 1; } 100
       { $value = 0 and truecount = 3 }
rule : { $value } { $value := 0; } 100
       { $value = 0 and truecount != 3 }
```

This model uses the following initial values (life.stvars file):

```
(4,4)=1
(4,5)=1
(4,6)=1
(5,4)=1
(5,5)=1
(5,6)=1
(6,4)=1
(6,5)=1
(6,6)=1
(5,10)=1
(6,9)=1
(6,10)=1
(10,9)=1
(10,10)=1
(10,11)=1
(11,9)=1
(11,10)=1
(11,11)=1
(12,9)=1
(12,10)=1
(12,11)=1
(10,15)=1
(11,14)=1
(11,15)=1
(14,14)=1
(14,15)=1
(14,16)=1
(15,14)=1
(15,15)=1
(15,16)=1
(16,14)=1
(16,15)=1
(16,16)=1
(15,20)=1
(16,19)=1
(16,20)=1
```

9.3 Default Ports

```
[top]
components : life

[life]
type : cell
width : 21
height : 21
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
initialvalue : 0
initialrowvalue : 4  000011100000000000000
initialrowvalue : 5  000011100010000000000
initialrowvalue : 6  000011100110000000000
initialrowvalue : 10 000000000111000000000
initialrowvalue : 11 000000000111000100000
initialrowvalue : 12 000000000111001100000
initialrowvalue : 14 000000000000001110000
initialrowvalue : 15 000000000000001110001
initialrowvalue : 16 000000000000001110011
localtransition : conrad-rule

[conrad-rule]
rule : { ~out := 1; } 100 { (0,0)~neighborChange = 1 and
                            (truecount = 3 or truecount = 4) }
rule : { ~out := 0; } 100 { (0,0)~neighborChange = 1 and
                            (truecount < 3 or truecount > 4) }
rule : { ~out := 1; } 100 { (0,0)~neighborChange = 0 and
                            truecount = 3 }
rule : { ~out := 0; } 100 { (0,0)~neighborChange = 0 and
                            truecount != 3 }
```

9.4 Non-default Ports

```
[top]
components : life

[life]
type : cell
width : 21
height : 21
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
initialvalue : 0
initialrowvalue : 4  000011100000000000000
initialrowvalue : 5  000011100010000000000
initialrowvalue : 6  000011100110000000000
initialrowvalue : 10 000000000111000000000
initialrowvalue : 11 000000000111000100000
initialrowvalue : 12 000000000111001100000
initialrowvalue : 14 000000000000001110000
initialrowvalue : 15 000000000000001110001
initialrowvalue : 16 000000000000001110011
localtransition : conrad-rule
neighborports : value

[conrad-rule]
rule : { ~value := 1; } 100
       { (0,0)~value = 1 and (statecount(1, ~value) = 3 or
                              statecount(1, ~value) = 4) }
rule : { ~value := 0; } 100
       { (0,0)~value = 1 and (statecount(1, ~value) < 3 or
                              statecount(1, ~value) > 4) }
rule : { ~value := 1; } 100
       { (0,0)~value = 0 and statecount(1, ~value) = 3 }
rule : { ~value := 0; } 100
       { (0,0)~value = 0 and statecount(1, ~value) != 3 }
```

10 References

[Ame00] Ameghino, J.; Wainer, G. "Application of the Cell-DEVS Paradigm Using N-CD++". In *Proceedings of the 32nd SCS Summer Computer Simulation Conference.* Vancouver, Canada. 2000.

[Aie01] Aiello, A.; Innocenti, E.; Muzy, A.; Santucci, J.; Wainer, G. "Comparing Simulation Methods for Fire Spreading Across a Fuel Bed". Computer Modeling, University of Corsica. 2001.

[Bar98] Barylko, A.; Beyoglonián, J.; Wainer, G. "CD++: una herramienta de implementación de modelos Cell-DEVS binarios". Informe Técnico 98-006, Departamento de Computación, FCEN, Universidad de Buenos Aires. 1998.

[Rod99a] Rodríguez D.; Wainer G, "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Multiconference on Computer Simulation.* 1999.

[Rod99b] Rodríguez D. "Implementación de modelos Cell-DEVS n-dimensionales – Informe Científico". Departamento de Computación, FCEN, Universidad de Buenos Aires. 2000.

[Rod99c] Rodríguez D. "Implementación de modelos Cell-DEVS n-dimensionales – Informe Técnico". Departamento de Computación, FCEN, Universidad de Buenos Aires. 2000.

[Tro01] Troccoli, A. "Parallel DEVS and Cell-DEVS models". M. Sc. Thesis, Departamento de Computación, FCEN, Universidad de Buenos Aires. 2001.

[Wai96] Wainer, G. "Introducción a la Simulación de Eventos Discretos". Informe Técnico 96-005, Departamento de Computación, FCEN, Universidad de Buenos Aires. 1996.

[Wai97] Wainer, G.; Giambiasi, N.; Frydman, C. "An Environment for Cellular DEVS Model Simulation". In *Proceedings of the SCS European Multiconference on Simulation.* Istanbul, Turkey. 1997.

[Wai00] Wainer, G. "Improved Cellular Models with Parallel Cell-DEVS". In *transactions of the Society for Computer Simulation.* 2000.

[Wai01] Wainer, G.; Giambiasi, N. "Timed Cell-DEVS: Modeling and Simulation of Cell Spaces". In *Discrete Event Modeling & Simulation: Enabling Future Technologies.* Springer-Verlag. 2001.

[Wai02] Wainer, G.; Giambiasi, N. "N-dimensional Cell-DEVS". In *Discrete Events Systems: Theory and Applications*, Kluwer, Vol. 12 N° 1, January 2002. pp 135-157.

[Zei00] Zeigler, B.; Kim, T.; Praehofer, H. "Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems". Academic Press. 2000.