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

2001

Modificaciones a CD++ para simulación paralela y distribuida de modelos Parallel Cell-DEVS

TESIS DE LICENCIATURA

Autor
Alejandro Troccoli

Director
Dr. Gabriel Wainer

CONTENIDOS

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

2001

Modificaciones a CD++ para simulación paralela y distribuida de modelos Parallel Cell-DEVS

INFORME CIENTÍFICO

Autor
Alejandro Troccoli

Director
Dr. Gabriel Wainer

Table of Contents

Abstract

Cell-DEVS es un formalismo para describir modelos celulares que se diferencia de los autómatas celulares tradicionales por la expresividad que provee para definir el avance del tiempo. Cell-DEVS se ha utilizado para modelar varias aplicaciones: tráfico, incendios forestales, inyección de flujo y otras. Pero la ejecución de modelos Cell-DEVS grandes y complejos requiere un poder de cómputo que muchas veces una sola computadora no provee, pero que sí se puede obtener utilizando ejecución paralela y distribuida. Por este motivo se hicieron modificaciones a Cell-DEVS que dieron origen a Parallel Cell-DEVS, un formalismo revisado que extiende Cell-DEVS para simulación en paralelo. El presente trabajo define un mecanismo de simulación que permite ejecutar modelos Parallel Cell-DEVS en ambiente paralelos, haciendo énfasis en aquellos que son distribuidos.

Cell-DEVS is a formalism intended to model cell spaces. It describes cellular models using timing delay constructions, allowing simple definition of complex timing. Large Cell-DEVS models require a computing power that their execution in a standalone machine is not feasible. As parallel and distributed environments became more accessible, the Cell-DEVS formalism was revised to permit parallel specification of these models. This work defines a new simulation mechanism suited for distributed environments and presents a tool for the simulation of Parallel DEVS and Cell-DEVS models on a network of computers.

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 begins 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. From the 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.

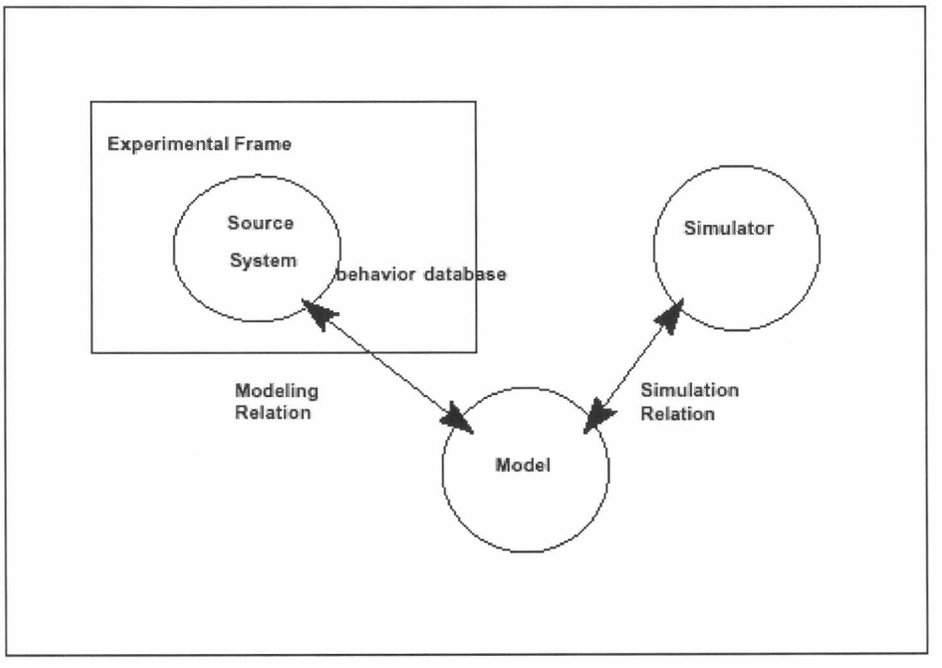


Figure 1 : The basic entities and their relationships [Zei00]

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 [Zei76, Zei00] provides a framework for the construction of hierarchical models in a modular manner, allowing for model reuse and reducing development and testing time time. In DEVS a model is specified as a black box with a state and a 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 a 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 one known as a **coordinator**. These simulators progress through the simulation by exchanging messages as described by the abstract simulation mechanism.

In [Wai01] the Timed Cell-DEVS formalism was presented. Cell-DEVS is a formalism based on DEVS for the simulation of cellular models. A traditional 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 cell's current state and the values its neighbors. Cells are updated synchronously all at the same time. Timed Cell-DEVS takes a different approach. It 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 traditional synchronous approach. In Timed Cell-DEVS each cell defines its own update delay, giving the modeler more precision and reducing the execution time.

To simulate DEVS and Cell-DEVS models a toolkit, CD++, was developed [Rod99]. CD++ has been used to simulate a variety of models including: traffic, forest fires, ants and watershed simulation [Ame00]. Simple models are easily handled by the tool. However, the execution of complex models requires a computing power that stand alone computers do not provide, but that can be provided by parallel and distributed systems.

Not only parallel execution was being demanded for Cell-DEVS but also for DEVS models. In [Cho94a] the Parallel DEVS formalism was introduced. This formalism is a revision of DEVS that eliminates serialization constraints that made it unsuitable for parallel execution. Similarly, Parallel Cell-DEVS [Wai00a] was introduced as a revised version of Cell-DEVS that eliminates serialization constraints and inconsistencies with zero delay cells and multiple simultaneous events.

This work presents changes to CD++ to run Parallel DEVS and Parallel Cell-DEVS models on a distributed environment, providing a tool that will not only reduce execution times but also allow larger models. To begin, a new abstract simulator will be presented because the Parallel DEVS simulator introduced in [Cho94b], though well suited for an implementation on a parallel system with shared memory, does not allow for an efficient implementation over a network of computers. Basically, the simulator in [Cho94b] does not distinguish messages sent over the network from those sent between objects on the same process, incurring in an unnecessary overhead. Therefore, there was a need to extend it for distributed environments. This work addresses this issues.

In parallel simulation, the execution is divided into a set of *Logical Processes*, each running on a different CPU. Each *Logical Processes* hosts a set of simulation objects. For flexibility, the new parallel simulator was designed as a layered architecture application. The topmost layer implements the abstract simulator, the middle layer carries out all required synchronization in the *Logical Process* level, and the lowest layer is in charge of communications. For the middleware, the Warped project [Mar97] was selected. Warped provides an API for running parallel simulation. There are three approaches to synchronization between *Logical Processes*: optimistic, pessimistic, and no synchronization (application level synchronization) and Warped currently provides two of these. A Time Warp kernel implements the optimistic Time Warp protocol, and a No Time kernel [Rao98] implements an unsynchronized protocol. The parallel simulator has been written to support both kernels and is currently being run with the No Time kernel.

The final release of parallel CD++ runs both, distributed and standalone simulation. For simple and small models, the standalone version performs well. For complex and big models the distributed version is preferred. The development was carried out in Linux machines. Testing has been done on different Linux clusters at the Universidad de Buenos Aires and at the University of Carleton in Ottawa.

This work is organized as follows. The first two chapters, Chapter 2 and 3, present the DEVS, Parallel DEVS, Cell-DEVS and Parallel Cell-DEVS formalisms. In chapter 4, the new abstract simulator suited for distributed environments is introduced. Chapter 5 will make a short presentation of synchronization techniques for parallel discrete event systems, introducing the optimistic, pessimistic and unsynchronized protocols. After this presentation, Chapter 6 will introduce CD++ and Chapter 7 the parallel version, with

special mention of implementation issues using the Warped kernels. A first set of results is presented in Chapter 8.

After the first results, some bottlenecks were detected, so the simulator was revised. These revisions are presented in Chapter 9. Chapter 10 makes evaluates the performance of the two simulators and analyses other factor affecting performance, such as model workload and choice of partition.

Chapter 11 introduces a chemical diffusion model and further performance analysis. Finally the conclusions follow.

2
The DEVS and Parallel DEVS formalisms

2.1 The original DEVS formalism

Systems whose variables are discrete and the time advance is continuos are known as **DEDS – Discrete Events Dynamic Systems**, as opposed to **CVDS – Continuous Variable Dynamic Systems** [Wai98]. 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 time specific point of time $t_i \in \boldsymbol{R}$.

DEVS [Zei76, Zei00] is a formalism for modeling and simulating DEDS systems. It defines a way of specifying systems whose states either 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). 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 $\delta_{ext}(s, e, x)$ where s is the current state, e the time elapsed since the last transition and x the external input value.

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) = ∞, the system will stay in that state forever unless an external event is received. In such a case, s is called a **passive state**. Figure 2 illustrates this definition, and Figure 3 shows how to define a CPU model with DEVS.

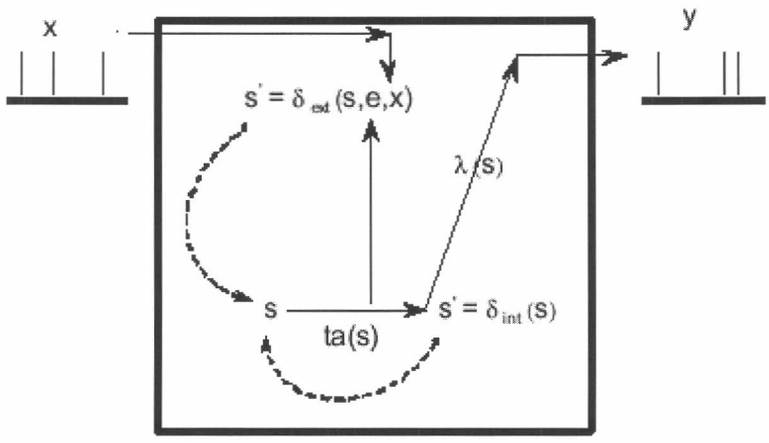


Figure 2 : DEVS Semantics

A computer processor can be specified as a DEVS model. A processor would have to 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 a 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 its 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.

Figure 3 : Definition of a CPU using DEVS

DEVS models can be put together to form *coupled* models.

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.

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

I_i is the set of influencees 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.

This structure is subject to the constraints that for each i in D,

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

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, this has to be mapped as another's component 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 if there are more multiple components scheduled for an internal transition at the same time. If the first component to make its internal transition produces an output that maps to an external event for another component that is already scheduled for an internal transition, then it is not clear which transition this second component should execute first. There two alternatives: to execute the external transition first with $e = ta(s)$ and then the internal transition, or to execute the internal transition first followed by the external transition with $e = 0$. The way the DEVS formalism solves this is by the use of the *select* function. This function defines an order on the components so that only one component of the group of imminent models is allowed to be with $e = 0$. The other imminent models will be divided in two groups: those that receive an external output from this

model, and the rest. The first group will execute their external transitions functions with $e = ta(s)$ and the second group will be imminent during 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 produce 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.

2.2 The Parallel DEVS formalism

The Parallel DEVS formalism [Cho94a] keeps all the nice properties of the DEVS formalism and eliminates all the serialization constraints that made simultaneous execution in a parallel environment not feasible.

Chow required that the following properties hold:

- Collision handling: the behavior of a collision must be controllable by the modeler.
- Parallelism: the formalism must not use any serialization function that prohibits possible concurrencies.
- Uniformity: the hierarchical construction must have uniform behavior: different hierarchical constructs of the same model must display the same behavior.

In DEVS, neither the first nor the second condition hold. Parallel DEVS resolves these issues.

As in DEVS, a P-DEVS model is described as a set of basic and coupled models. Atomic models are still the most basic constructions, which can be combined with other models into coupled models. A Parallel-DEVS coupled model satisfies the closure property [Cho94b], so it can used as another basic model. Therefore, Parallel-DEVS preserves the hierarchical properties of the original DEVS formalism.

A basic Parallel DEVS is a structure:

$$DEVS = < X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

where

$X_M = \{(p,v) | p \in IPorts, v \in X_p\}$ is the set of *input ports and values*;

$Y_M = \{(p,v) | p \in OPorts, v \in Y_p\}$ is the set of *output ports and values;*

S is the set of *sequential* states;

$\delta_{ext}: Q \times X_M^b \rightarrow S$ is the *external state transition functio*n;

$\delta_{int}: S \rightarrow S$ is the *internal state transition functio*n;

$\delta_{con}: Q \times X_M^b \rightarrow S$ is the *confluent transition functio*n;

$\lambda : S \rightarrow Y_M^b$ is the *output function;*

$ta : S \rightarrow R_0^+ \cup \infty$ is the *time advance functio*n;

with $Q := \{ (s, e) \mid s \in S, 0 \leq e \leq ta(s) \}$ the set of *total state*s.

The differences between the DEVS and Parallel-DEVS formalism are the following:

- The model interface has been extended to include ports and values. A model will now have input and output ports through which all interaction with the environment takes place. Events determine values appearing on such ports. A model receives outside events through its input ports. Upon reception of such events, the model description must determine how it responds to them. In addition, internal events arising within the model change its state, and manifest themselves as events on the output ports to be transmitted to other model components.

- The external and output functions no longer handle one event at a time. Instead, bags of events are now being handled, allowing then for simultaneous processing of multiple events.

- A new transition function has been defined, the confluent function δ_{con}. This function will define a new model's state when there is a collision between internal and external transitions. Basically, this function will allow the modeler to specify how the model should behave in the presence of collisions.

The semantics of the Parallel-DEVS definition are then as follows. At any given time, a basic model is in a state s and in the absence of external events, it will remain in that state for a period of time as defined by ta(s). When an internal transition takes place, the system outputs the value $\lambda(s)$, and changes to state $\delta_{int}(s)$. If one or more external events $E = \{ x_1 .. x_n / x \in X_M \}$ occurs before ta(s) expires, i.e., when the system is in total state (s, e) with $e \leq$ ta(s), the new state will be given by $\delta_{ext}(s,e,E)$. When an external and internal transition collide, i.e. external events E arrives when e = ta(s), the new system's state could either be given by $\delta_{ext}(\delta_{int}(s),e,E)$ or $\delta_{int}(\delta_{ext}(s,e,E))$. To avoid a fix behavior, the modeler can define the most appropiate behavior with the δ_{conf} function. Then, in the Parallel DEVS formalism, in the presence of collisions the new system's state will be the one defined by $\delta_{conf}(s,E)$.

A Parallel DEVS *coupled model* is defined by:

$$CM = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC \rangle$$

where

$X = \{(p,v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values;

$Y = \{(p,v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values;

D is the set of the component names;

The following constraints apply to the components:

Components are DEVS models:

for each $d \in D$

$M_d = (X_d, Y_d S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$ *is a DEVS basic structure*

with $X_d = \{(p,v) \mid p \in IPorts, v \in X_p\}$;

$Y_d = \{(p,v) \mid p \in OPorts, v \in Y_p\}$;

The couplings are subject to the following conditions:

- *external input couplings (EIC)* connect external inputs to component inputs:

$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$

- *external output couplings (EOC)* connect component outputs to external outputs:

$EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$

- *internal couplings (IC)* connect component outputs to component inputs:

$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$

No direct feedback loops are allowed, i.e., no output port of a component may be connected to an input port of the same component,

$((d, op_d), (e, ip_d)) \in IC$ implies $d \neq e$.

- Range inclusion constraints: the values sent from a source port must be within the range of accepted values of a destination port, i.e.,

$\forall((N, ip_N), (d, ip_d)) \in EIC : X_{ipN} \subseteq X_{ipd}$

$\forall ((a, op_a), (N, op_N)) \in EOC : Y_{opa} \subseteq Y_{opN}$

$\forall ((a, op_a), (b, ip_b)) \in IC : Y_{opa} \subseteq X_{ipb}$.

The Parallel-DEVS definition eliminated the *select* function. If there multiple imminent components, then all their outputs will be first collected and mapped to their influencees. Then, the corresponding transition function will be executed for each model.

As an example, a generator-processor-transducer (gpt) model will be shown. The aim of this model is to calculate the usage of a given processor. It is made of three atomic models:

- A generator that generates new jobs at random time intervals.
- A processor that consumes the jobs that the generator produces.
- A transducer: a model that will keep count of the number of jobs processed and the time it took to process each job.

The generator has two input ports: *start* and *stop*, and an output port *out*. Whenever a new job is generated, a new event is sent through the out port. The processor has one output port *in* and an output port *out.* A new job is received through the *in* port and when it has been processed after an elapsed time *t*, an event is sent through the *out* port. The transducer has two input ports: *arriv* and *solved*, and one output port *result*. When an event is recevied through *arriv* a timer is started and a job count is increased by one. When an event is received through the *solved* port the counter is stoped. After an pre-defined observation period of time, the processor usage is sent through the out port. The whole coupled has two input ports *start* and *stop*, and two output ports *out* and result. The couplings are shown in Figure 4.

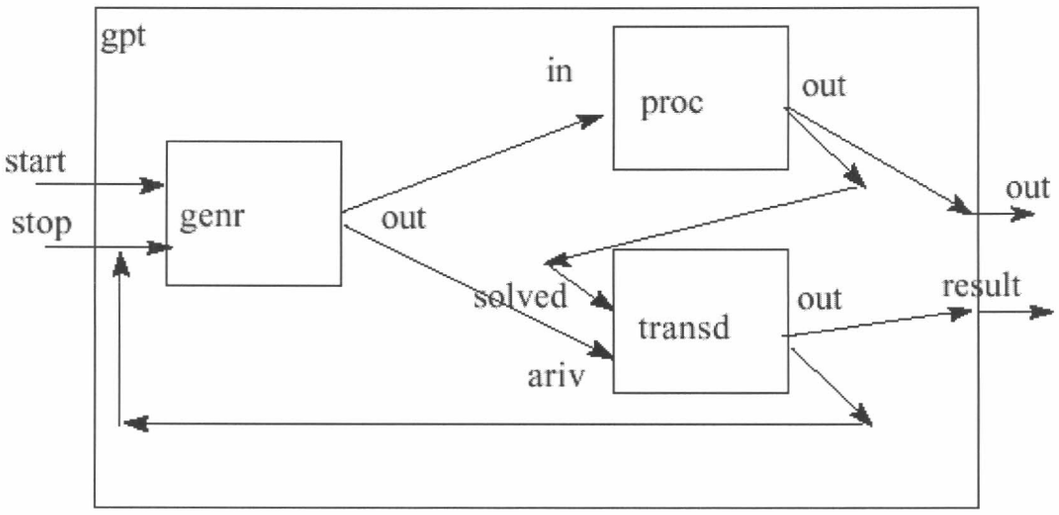


Figure 4: The GPT coupled model. [Zei00]

3
The Cell –DEVS and Parallel Cell-DEVS formalisms

3.1 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).

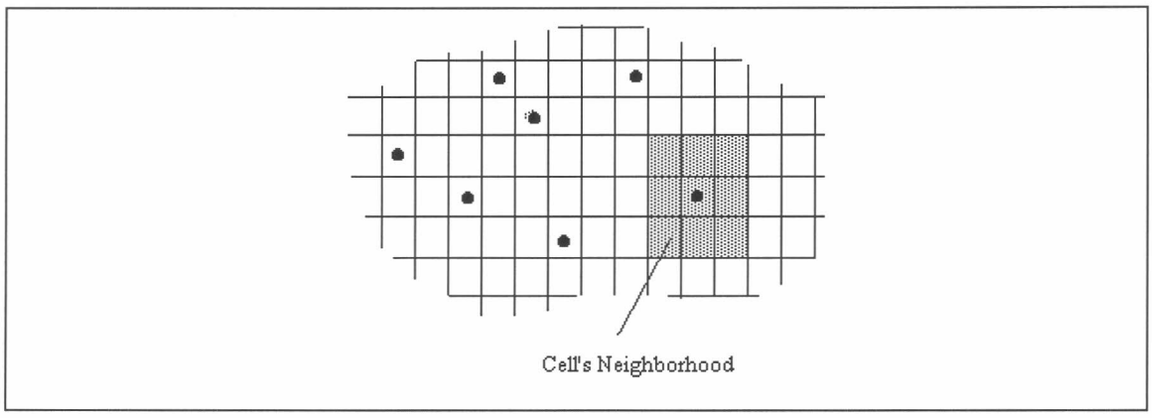


Figure 5 : Sketch of a Cellular Automaton [Wai00a]

When cellular automata are used to simulate complex systems, large amounts of compute 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 [Wai98] 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.

3.2 The Timed Cell-DEVS formalism

Cell-DEVS defines a cells as DEVS atomic models. A Cell-DEVS atomic model is defined by [Wai98]:

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

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 cell state definition;
N	is the set of states for the input events;
d	is the 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 N 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 delayed 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.

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	represents the definition of the interface for the modular model;
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.

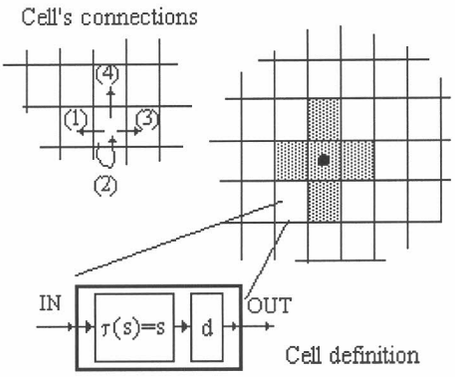


Figure 6 : Informal definition of a Cell-DEVS model [Wai98]

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 disallows [Wai00a]:

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

To forbid 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.

3.3 The Parallel Cell-DEVS formalism

A parallel Cell-DEVS basic model can be formally defined as:

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

where

In this case, $\#T < \infty \wedge T \in \{N, Z, R, \{0,1\}\} \cup \{\phi\}$;

$X \subseteq T$;

$Y \subseteq T$;

$I = < \eta, \mu^x, \mu^y, P^x, P^y >$. Here, $\eta \in N$, $\eta < \infty$ is the neighborhood's size, $\mu^x, \mu^y \in N$, $\mu^x, \mu^y < \infty$ is the number of other input/output ports, and $\forall j \in [1, \eta]$, $i \in \{X, Y\}$, P_j^i is a definition of a port (input or output respectively), with $P_j^i = \{ (N_j^i, T_j^i) / \forall j \in [1, \eta+\mu^i], N_j^i \in [i_1, i_{\eta+\mu}]$ (port name), y $T_j^i \in I_i$ (port type)}, where $I_i = \{ x / x \in X \text{ if } X \}$ or $I_i = \{ x / x \in Y \text{ if } i = Y \}$;

$S \subseteq T$;

$\theta = \{$ (s, phase, σqueue, f, σ) /

s ∈ S is the status value for the cell,

s' ∈ S is an intermediate status value for the cell;

phase ∈ {active, passive},

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

$f \in T$; and

$\sigma \in R_0^+ \cup \infty \}$;

$N \in S^{\eta+\mu}$;

$d \in R_0^+$, $d < \infty$;

δ_{int}: $\theta \rightarrow S$;

δ_{ext}: $QxX^b \rightarrow \theta$, $Q = \{ (s, e) / s \in \theta \text{ x } N \text{ x } d; e \in [0, D(s)]\}$;

δ_{con}: $\theta x X^b \rightarrow S$;

τ: $N \rightarrow S$ x {inertial, transport} x d;

τ_{con}: $X^b x N \rightarrow S$ x {inertial, transport} x d;

λ: $S \rightarrow Y^b$; and

D: θ x N x d $\rightarrow R_0^+ \cup \infty$.

A Cell-DEVS atomic model is a specialization of a Parallel DEVS basic model. The difference between an atomic model and a Cell-DEVS model is the existence of a cell neighborhood, a delay *d* and a local computation function τ. The *I* interface defines a fixed number of ports for message exchange to neighbor cells.

Originally, only one kind of delay of a given duration was related with each cell. Now, the local transition function will return the type and length of the delay, and the cell's outputs will be delayed accordingly. This redefinition allows to include complex timing behavior.

In the presence of collisions between internal and external events, the confluent transition function δ_{con} is activated. It must activate the confluent local transition function τ_{con}, whose goal is to analyze the present values for the input bags, and to provide a unique set of input values for the cell. In this way, the cell will compute the next state by using the values chosen by the modeler. Basically, what τ_{con} does is to choose members from the bag, and update the inputs for the cell. After, it deletes the unnecessary members of the bag.

The following figure shows a sketch of the contents of each cell.

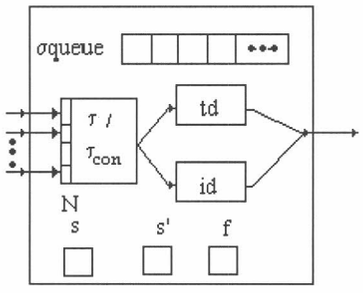


Figure 7: Cell's definition [Wai00a]

Atomic Cell –DEVS models can be put together to form coupled Cell-DEVS models. A parallel Cell-DEVS coupled model can be represented as:

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

Xlist is the input coupling list;

Ylist is the output coupling list;

I represents the definition of the interface for the modular model;

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; and

Z is the translation function.

$$C = \{ C_C / c \in \mathbf{I} \wedge C_C = < I_C, X_C, Y_C, S_C, N_C, d_C, \delta_{intC}, \delta_{extC}, \delta_{conC}, \tau_C, \tau_{conC} \lambda_C, D_C > \},$$

where C_C is a parallel Cell-DEVS atomic model, and $I = \{ (i_1,...,i_n) / (i_k \in N \wedge i_k \in [1, t_k]) \forall k \in [1, n]\}$. That is, each cell in the space is a parallel Cell-DEVS atomic cell using the δ_{con} and τ_{con} functions to avoid collisions.

As stated in [Wai00a], the following lemmas apply.

Lemma 1
The Parallel Cell-DEVS models are equivalent to parallel DEVS models.

Lemma 2
Closure under coupling for parallel Cell-DEVS models: a coupled parallel Cell-DEVS model is equivalent to a basic parallel Cell-DEVS model.

This two lemmas imply that within a coupled Parallel DEVS model, a Cell-DEVS model can be used as if it were a basic Parallel DEVS model. This property will be used in the next section, when the abstract simulator is described, to prove that the abstract simulator for Parallel DEVS models will also execute Parallel Cell-DEVS models.

If a parallel Cell-DEVS model can be viewed as parallel DEVS model, then it should be possible to define its corresponding δ_{ext}, δ_{int}, δ_{conf}, and λ functions. The semantics for these functions will be now presented.

Note: σqueue is a list of pairs (delay, value) sorted by ascending order of delay. These are the values scheduled for output. The following operations are defined for the queue:

first: the first pair.

head: the set of pairs from the front of the queue with minimum delay.

tail: queue – head

add: adds a new pair to the queue.

δ_{int}:

$$\frac{\sigma = 0; \qquad \sigma queue \neq \{\varnothing\}; \qquad phase = active}{\forall\, i \in [1, m],\ a_i \in \sigma queue,\ a_i.\sigma = a_i.\sigma - head(\sigma queue.\sigma); \qquad \sigma queue = tail(\sigma queue); \qquad \sigma = head(\sigma queue.\sigma);}$$

$$\frac{\sigma = 0; \qquad \sigma queue = \{\varnothing\}; \qquad phase = active}{\sigma = \infty \ \wedge \ phase = passive}$$

λ:

$$\frac{\sigma = 0;}{out = \{\ a_i.v \mid a_i \in head(queue)\ \};}$$

δ_{ext}:

$$\frac{N_c = \tau_{con}(X^b); \quad (s', transport) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \text{ x } N \text{ x } d); \quad phase = active;}{s \neq s' \Rightarrow (s = s' \ \wedge \ \forall\, i \in [1,m]\ a_i \in \sigma queue,\ a_i.\sigma = a_i.\sigma - e \ \wedge \ \sigma = \sigma - e;\ add(\sigma queue, <s', d>) \wedge f = s\ )}$$

$$\frac{N_c = \tau_{con}(X^b); \quad (s', transport) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \text{ x } N \text{ x } d); \quad phase = passive;}{s \neq s' \Rightarrow (\ s = s' \ \wedge \ \sigma = d \ \wedge \ phase = active \ \wedge \ add(\sigma queue, <s', d>) \ \wedge f = s\ )}$$

$$\frac{N_c = \tau_{con}(X^b); \quad (s', inertial) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \text{ x } N \text{ x } d); \quad phase = passive;}{s \neq s' \Rightarrow (\ s = s' \ \wedge \ phase = active \ \wedge \ \sigma = d \ \wedge \ f = s\ )}$$

$$\frac{N_c = \tau_{con}(X^b); \ (s', inertial) = \tau(N_c); \ \sigma \neq 0; \quad e = D(\theta \text{ x } N \text{ x } d); \ phase = active;}{s \neq s' \Rightarrow \ s = s' \ \wedge \ (f \neq s' \ \Rightarrow \ \sigma queue = \{\varnothing\} \wedge \sigma = d \ \wedge \ f = s)}$$

3.3 Cell-DEVS Quantization

Recently, a theory of quantized models was developed [Zei98a, Zei98b]. When using a quantized model, after a cell's state value will be only informed to its neighbors if its difference with the previous value is greater than a given *quantum*. This idea is shown in Figure 8. Here, a continuos curve is represented by the crossings of an equal spaced set of boundaries, separated by the *quantum* size. A *quantizer* checks for boundary crossings whenever a change in a model takes place. Only when such a crossing occurs, a new value is sent to the receiver. This operation reduces substantially the frequency of message updates, while potentially incurring into error.

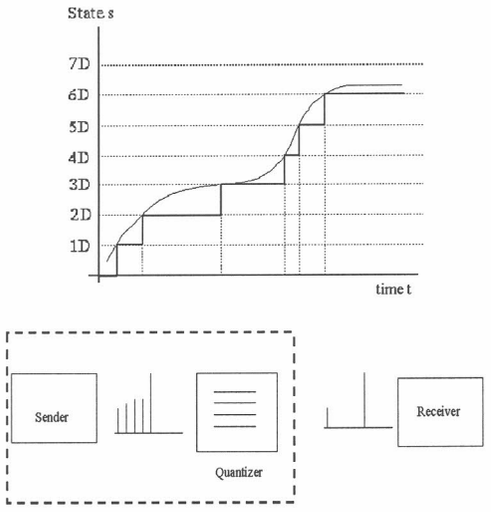


Figure 8 : Quantization (Zeigler et al 1999)

In [Wai00b] several experimental tests were done in order to analyze the behavior of quantized Cell-DEVS models. The results showed that quantization reduced both, the total number of messages sent and the execution time, but introduced an error. The error obtained is a function of the local computing function, the number of simulation steps and the quantum. Since the future input values for a cell depend on the present results, a nonlinear error may be observed. The error magnitude will depend on the cell's neighborhood size. It was shown in [Wai00b] that as the quantum gets higher, the error gets bigger.

Choosing an adequate quantum will then depend on the precision desired.

When quantization is used with a quantum value **d**, δ_{ext} is defined as:

δ_{ext}:

$$N_c = \tau_{con}(X^b); \quad (s', \text{transport}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \times N \times d); \quad \text{phase} = \text{active};$$

$$s \neq \text{value}(s',d) \Rightarrow$$
$$(s = s' \wedge \forall i \in [1,m]\ a_i \in \sigma\text{queue}, a_i.\sigma = a_i.\sigma - e \wedge \sigma = \sigma - e;\ \text{add}(\sigma\text{queue}, <s', d>) \wedge f = s)$$

$$N_c = \tau_{con}(X^b); \quad (s', \text{transport}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \times N \times d); \quad \text{phase} = \text{passive};$$

$$s \neq \text{value}(s',d) \Rightarrow (s = s' \wedge \sigma = d \wedge \text{phase} = \text{active} \wedge \text{add}(\sigma\text{queue}, <s', d>) \wedge f = s)$$

$$N_c = \tau_{con}(X^b); \quad (s', \text{inertial}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \times N \times d); \quad \text{phase} = \text{passive};$$

$$s \neq \text{value}(s',d) \Rightarrow (s = s' \wedge \text{phase} = \text{active} \wedge \sigma = d \wedge f = s)$$

$$N_c = \tau_{con}(X^b); \quad (s', \text{inertial}) = \tau(N_c); \quad \sigma \neq 0; \quad e = D(\theta \times N \times d); \quad \text{phase} = \text{active};$$

$$s \neq \text{value}(s',d) \Rightarrow s = s' \wedge (f \neq s' \Rightarrow \sigma\text{queue} = \{\varnothing\} \wedge \sigma = d \wedge f = s)$$

where

$$\text{value}(v,d) = v' \text{ such that } \exists\ q \in N / v' = q.d \wedge v' \leq v.$$

i.e. the lowest boundary as defined by the quantum size.

e.g.: value(23.45, 0.1) = 23.4 value(550, 100) = 500

4
Abstract simulator for distributed Parallel-DEVS

The DEVS formalism separates the model from the actual simulator. In [Cho94b] an abstract simulator for the Parallel DEVS formalism was presented. Though well suited for shared memory parallel environments, this abstract simulator does not distinguish between intra-process messages and inter-process messages. Distributed environments have an important communications overhead that affects inter-process messages. To keep this overhead low, this type of messages should be minimized

A new abstract simulator suitable for distributed environments has been developed and is presented next.

4.1 Parallel DEVS Abstract Simulators

The simulation of DEVS models is carried out by *Processors* that drive the simulation forward by exchanging messages. There are two types of *Processors*: *Simulators*, driving the simulation of atomic models, and *Coordinators*, in charge of executing coupled models and coordinating the activities of all their dependants. *Processors* are organized in a hierarchy that resembles the model hierarchy, as show in Figure 9.

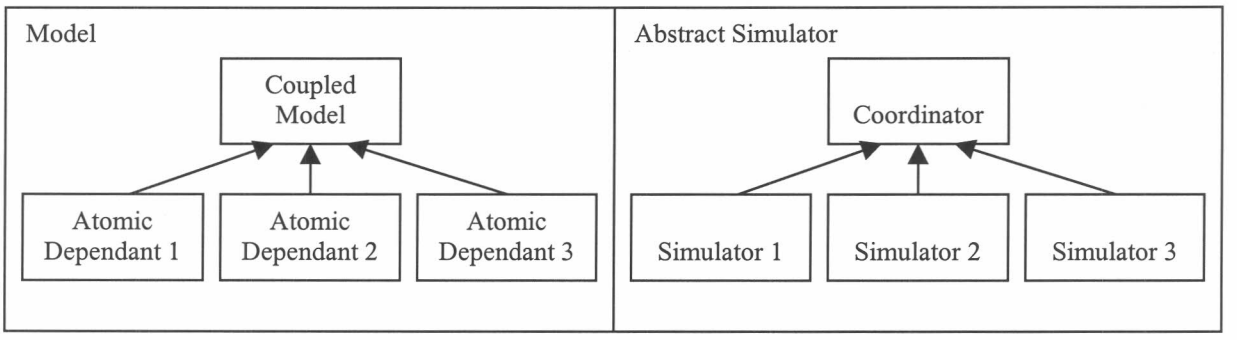


Figure 9 : Correspondence between the model and the DEVS processors

In the same way a coupled model has a set of components, every *coordinator* has a set of child *Processors*, one for each component of the coupled model. When a simulation is run in distributed fashion, each machine will run one a *Logical Process* that will host one or more *Processors*. Under these assumptions, a *coordinator's* children need not be executing on the same *Logical Process*. Then every message sent to child *Processors* running on a different *Logical Process* will require inter-process communication. Figure 10(a) illustrates this case. It shows a *coordinator* sending a message to its 8 children distributed on two machines. Four inter-process messages are required for the four children running on processor 1. From now on, *Processors* that are running on the same *Logical Process* will be called local to each other.

When the number of children *Processors* is high (as it usually is for coupled Cell-DEVS), the number of messages sent across the network will be significant. This can be avoided if every coupled model has more than one *coordinator*. Figure 10(b) illustrates this case. For the same coupled model, there are two *coordinators*, one in *Logical Process* 0 and another one in *Logical Process* 1. In this case, only one message is sent over the network.

So, to reduce inter-process messages, coupled models will require a *coordinator* on each *Logical Process* where a child *Processor* is running. Children *Processors* will send messages to the local *coordinator*, which will decide how to handle the received messages. But care should be taken because the existence of multiple *coordinators* for one coupled model can cause duplicate messages. To avoid this, there will be only one *coordinator* that will communicate with the parent's model. This specialized *coordinator* will be a *master coordinator* and the others will be *slaves*.

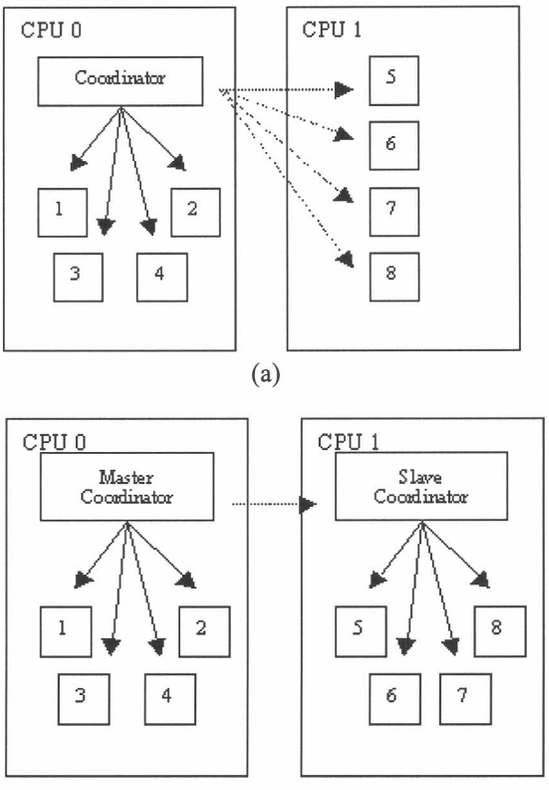


Figure 10 : (a) A single coordinator sending a message to all its child processor. Dashed lines = interprocess messages. (b) A master- slave pair sending messages to all their children processors.

When *master* and *slave coordinators* are used, *Processors* are organized in a hierarchy, which does not have a one to one correspondence with the model hierarchy . Therefore a parent child-relationship that takes into account the existence of *master* and *slave coordinators* is defined as follows:

a. for each *simulator*, the parent *coordinator* will be the parent's model local *Processor* (it is guaranteed that this will exist)

b. for each *slave coordinator*, the parent *coordinator* will be the model's *master coordinator.*

c. for each *master coordinator*, the parent *coordinator* will be the parent's model local processor; just as if it were a *simulator.*

At the beginning of the simulation, each *simulator* will be assigned to a *Logical Process*. Then for each coupled model, a *coordinator* will be placed in every *Logical Process* where there is a *simulator* or a *master coordinator* that corresponds to a component of the model. One of the *coordinators* will be designated as *master coordinator.*

The simulation is message driven. *Processors* exchange messages of the form *(type, time)* and can belong to one of two categories: synchronization messages and content messages.

Synchronization messages:

(@, t)	*Collect message*
(, t)*	*Internal message*
(Done , t)	*Done message*

Content messages:

(q , t)	*External message*
(y, t)	*Output message*

In addition, a *Processor* has a set of internal variables to keep the time of the simulation:

t_L = *Time of last transition*
t_N = *Time of next change*

and a *bag* to temporarily store the external messages (q , t).

At any instant t, a *Processor* is said to be imminent if $t = t_N$.

A simulation cycle starts when the topmost *coordinator* sends a *(@, t)* message. This message tells all the imminent *simulators* to execute their output functions. At the same time, *coordinators* will make the necessary translations of the resulting *(y, t)* messages to *(q, t)* messages that are sent to a model's influencees. When a *Processor* has finished sending its outputs, it sends a *(done , t)* message to its parent *coordinator*. When the topmost coordinator receives a *(done , t)* all the outputs have been processed, so it sends a *(*, t)* message to trigger the execution of a model's transition function.

A *simulator* receiving a *(*, t)* message will execute one of the three transition functions of its associated atomic model: δ_{int}, δ_{ext}, or δ_{con} . If the model is imminent and has not received any external event, then δ_{int} is executed. If the model is not imminent and has received external events, then δ_{ext} is executed. Finally, if a model is imminent and received external events, δ_{con} is executed, which will decide which of the external or internal transition function should be executed.

A *coordinator* receiving a *(*, t)* message will forward this message to all its dependants that are either imminent or that have received external events.

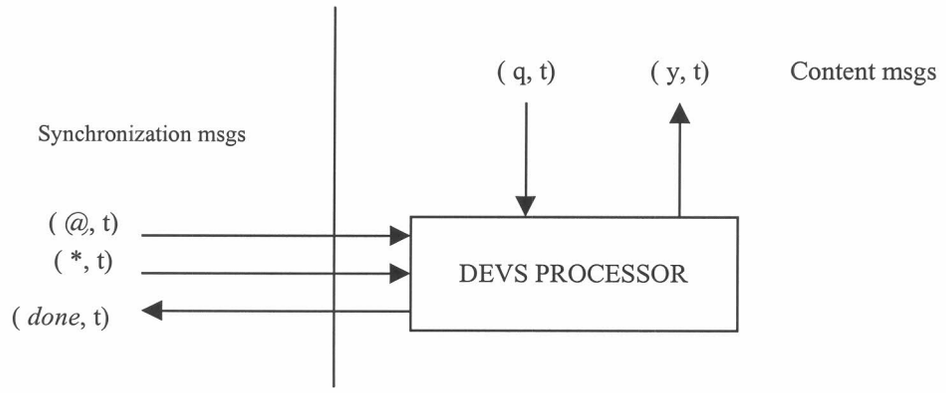


Figure 11: Messages a DEVS processor receives and sends

The complete behavior of a *Processor* is defined by how it handles each of these messages. To completely define the abstract simulator, the behavior of the *simulator*, *master coordinator*, *slave coordinator* and *root coordinator* will be described.

The *simulator* is responsible of invoking the atomic model's $\lambda(s)$, δ_{ext}, δ_{int}, δ_{con} functions. The description that follows is based on the one in [Cho94b], with some minor changes:

SIMULATOR

when a (@ , t) message is received

if $t = t_N$ **then**

$y := \lambda(s)$

send (y , t) to the parent coordinator

send (*done*, t) to the parent coordinator

end if

else raise error

end when

SIMULATOR

when a (q , t) message is received

lock the *bag*

Add event q to the *bag*

unlock the *bag*

end when

SIMULATOR

when a (* , t) message is received

case $t_L \leq t < t_N$

$e := t - t_L$

$s := \delta_{ext}(s, e, bag)$

empty *bag*

end case

case $t = t_N$ and *bag* is empty

$s := \delta_{int}(s)$

end case

case $t = t_N$ and *bag* not is empty

$s := \delta_{con}(s, bag)$

empty *bag*

end case

case $t > t_N$ or $t < t_L$

raise error

end case

$t_L := t$

$t_N := ta\ (s)$

send (*done*, t_N) to parent coordinator

end when

The *(*, t)* message is received when a model's transition function must be executed. The transition function to be executed will depend on t and the content's of the bag. If $t < t_N$, then it is not the time for an internal transition, and it must be the case that the bag is not empty and δ_{ext} should be executed. If $t = t_N$, it is the time for an internal transition. If no external messages have been received then δ_{int} is executed, but if there are external messages, then δ_{con} should be called instead.

Now the *master coordinator* will be described. A *coordinator*, whether *master* or *slave*, drives the simulation of a coupled model. Each *coordinator* has a set of child *Processors*. The role of the *coordinator* is to keep track of the imminent *Processors* and to translate output events to input events.

For a *master coordinator* the set of child *Processors* is made of:

- the set of *slave coordinators*
- the set of local *simulators* and *master coordinators* that correspond to components of the coupled model.

To simplify the following description it is necessary to define the function *coordinator*.

coordinator : M x P → C
where
M is a coupled model
P is a *Processor*
C is a *coordinator (master or slave)*

coordinator (M, j) = i , where *i* is the *coordinator* associated to coupled *M* that is local to child *j*. The following restrictions apply for the function to be well defined:

j is a DEVS processor associated to a dependant of *M*

i is one of the *coordinators* associated with *M*

MASTER COORDINATOR

when a (@ , *t*) message is received from parent coordinator
  if $t = t_N$ **then**
    $t_L := t$
    for all imminent child processors *i* with minimum t_N
      send (@, *t*) to child *i*
      cache *i* in the *synchronize* set
    end for
    wait until (*done*, *t*)'s have been received from all imminent processors
    send (*done*, *t*) to parent coordinator
  end if
  else raise error
end when

When a *master coordinator* receives an output message, two cases need to be distinguished:

an output message (*y* , *t*) received from a *slave coordinator*

an output message (*y* , *i*, *t*) forwarded from a *slave coordinator* that received (*y* , *t*) from a local child *i*.

MASTER COORDINATOR

when a (y , t) message is received from child i

for all influencees, j of child i

if j is a local processor

$q := z_{i,j}(y)$

send (q, t) to child j

cache j in the *synchronize* set

else

$s :=$ coordinator(*self*, j)

if $s \notin$ *slave-sync* set then

send (y, i, t) to s

cache s in the *slave-sync* set

cache s in the *synchronize* set

end if

end if

end for

if $self \in I_i$ (y is to be transmitted upward) **then**

$y := z_{i,\ self}(y)$

send (y, t) to parent coordinator

end if

clear *slave-sync* set

end when

when a (y , i, t) message is received from a slave s

cache s in the *slave-sync set* and proceed as if a (y , t) message had been received from child i

end when

Here *slave-sync* is used to avoid forwarding an output message twice to a *slave coordinator*. It is important to note that instead of forwarding a (q, t) message to a *slave coordinator*, a (y, i, t) is sent. A *slave coordinator* might be the parent *coordinator* for more than one of the influencees of i. If (q , t) messages were to be forwarded, then there will be one (q, t) message for each influencee of i. For Cell–DEVS models, this can be an important overhead. Instead, just one (y, i, t) message is sent across the network. The recipient *slave coordinator* will generate the appropiate (q, t) messages.

As mentioned in [Cho94b], all children ready for a transition are cached in a *synchronize* set to later distinguish active from inactive components.

MASTER COORDINATOR

when a (q , t) message is received from parent coordinator

lock the *bag*

Add event q to the *bag*

unlock the *bag*

end when

MASTER COORDINATOR

when a (* , t) message is received from parent coordinator

if $t_L \leq t \leq t_N$

for all $q \in bag$

for all receivers of q, $j \in I_{self}$

if j is a local processor

$q := z_{self,j}(q)$

send (q, t) to j

cache j in the synchronize set

else

$s :=$ coordinator(*self*, j)

if $s \notin$ *slave-sync* set then

send (q , t) to s

cache s in the *slave-sync* set

cache s in the *synchronize* set

end if

end if

end for

clear *slave-sync* set

end for

empty *bag*

for all i in the *synchronize* set

send (*, t) to i

end for

wait until all (*done*, t_N)'s are received

$t_L := t$

$t_N :=$ minimum of components' t_N's

clear the *synchronize* set

send (*done*, t_N) to parent coordinator

else raise an error

end when

When a *coordinator* receives a (*,t) two actions must be taken. First, all external events that were stored in the bag need to be forwarded to the corresponding models. If an external event needs to be routed

down to a *slave coordinator* the z translation is not be applied. Instead, the original q message is sent. Therefore, care must be taken not to forward a message twice to a *slave coordinator*. Here again, the *slave-sync* is used for that purpose.

In a second phase, all processors in the *synchronize* set are sent a (*,t) message.

The *slave coordinator* will be introduced next. It differs from the *master coordinator* in only one way: when a message needs to be sent a processor that is not local, it will be sent to the *master coordinator* instead.

For a *slave coordinator*, the set of child processors is made of

- the set of local *simulators* and *master coordinators* that correspond to components of the coupled model.

SLAVE COORDINATOR

when a (@ , t) message is received from master coordinator

if $t = t_N$ then

$t_L := t$

for all imminent child processors i with minimum t_N

send (@, t) to child i

cache i in the *synchronize* set

end for

wait until (*done*, t)'s have been received from all imminent processors

send (*done*, t) to master coordinator

end if

else raise error

end when

As it can be noticed, there is no difference on how both *master* and *slave coordinators* handle a (@, t).

SLAVE COORDINATOR

when a (y , t) message is received from child i

sent_to_master := false

for all influencees, j of child i

if j is a local processor

$q := z_{i,j}(y)$

send (q, t) to child j

cache j in the *synchronize* set

else

if not *sent_to_master*

send (y, t) to master coordinator

sent_to_master := true

end if

end if

end for

if *self* $\in I_i$ (y is to be transmitted upward) **then**

if not *sent_to_master*

send (y, t) to master coordinator

end if

end if

end when

when a (y , i, t) message is received from master coordinator

sent_to_master := true

proceed as if a (y , t) message had been received from child i

end when

When an output event is received from a child i, the *slave coordinator* sorts the message to the influencees of i. If any influencee is local, the z function is applied a (q , t) message is sent. If there are non-local influencees, then the output event is sent to the *master coordinator*, who will then sort the message to other *slave coordinators* if necessary. Only one (y , t) message should be forwarded to the *master coordinator*.

When the *slave coordinator* receives an output event that has been forwarded by the *master coordinator* on behalf of child i, it will handle the event as if i had been local, but no (y, t) messages will be forwarded back to the *master coordinator* if there is a non-local influencee. This is to avoid infinite loops of messages being forwarded back and forth.

SLAVE COORDINATOR

when a (q , t) message is received from master coordinator

lock the *bag*

Add event q to the *bag*

unlock the *bag*

end when

SLAVE COORDINATOR

when a (* , t) message is received from master coordinator

if $t_L \leq t \leq t_N$

for all $q \in bag$

for all receivers of q, $j \in I_{self}$

if j is a local processor

$q := z_{self,j}(q)$

send (q, t) to j

cache j in the synchronize set

else

do nothing

end if

end for

end for

empty *bag*

for all i in the *synchronize* set

send (*, t) to i

end for

wait until all (*done*, t_N)'s are received

$t_L := t$

$t_N :=$ minimum of components' t_N's

clear the *synchronize* set

send (*done*, t_N) to master coordinator

else raise an error

end when

The root coordinator is a special processor that is above the topmost coordinator. It is responsible for driving the simulation and advancing the virtual simulation time. The root coordinator can also handle external events which are stored in a sorted queue of events.

ROOT COORDINATOR

load *queue* of external events and sort them by arrival time.

$t :=$ minimum of t_N of topmost coordinator and t_N of *queue*.

while $t \neq \infty$

if $t = t_N$ of *queue*

for all q in *queue* with time t

send (q , t) to topmost coordinator

end for

end if

if $t = t_N$ of topmost coordinator

send (@, t) to topmost coordinator

wait until (*done, t*) is received from it

end if

send (*, t) to topmost coordinator

wait until (*done, t*) is received from it

end while

raise simulation completed

The abstract simulator has been now completely defined. This abstract simulator will be able to handle both, Parallel DEVS and Parallel Cell-DEVS models.

5
Parallel Simulation

When running parallel and distributed simulation, the whole model is divided among a set of *Logical Process*, each of which will execute on a different CPU. In general terms, each *Logical Process* will host one or more simulation objects. For the present discussion, those simulation objects will be any of the three DEVS *Processors: simulators, master and slave coordinators*.

Logical Processes (LPs) communicate by using time-stamped events that move the simulation forward. In order to obtain correct results, LPs must process messages in strictly non-decreasing timestamp order. Each LP has an input queue of messages to process. Figure 12 shows two LPs, each with one event in its input queue. Both events are processed simultaneously, and as a result of processing C with time 2, a new event D is generated for LP 1 with time stamp 5. But LP 1 has already processed an event with timestamp 8 so the simulation is incorrect. Such an error is called a causality error.

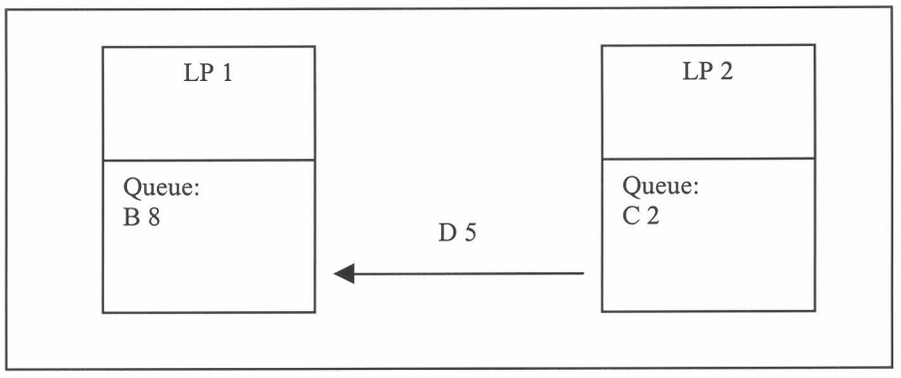


Figure 12: Execution of the first queued message does not always guarantee correct results.

Therefore, either LPs must agree on a synchronization mechanisms, or the application programmer has to ensure the application will keep the LPs synchronize.

For event driven simulation, there are three types of synchronization strategies:

1. No synchronization at all (synchronization is ensured by the application).
2. Optimistic synchronization.
3. Pessimistic (conservative) synchronization.

The first approach assumes all messages will always arrive in the order defined by their time-stamp, and no out of order message will ever be received. It is an optimistic strategy that relies on the synchronization being handled by the simulation objects instead of the logical process themselves. It is a very efficient implementation that does not require event queues; each event is processed as soon as it arrives.

The other two rely on synchronization being handled by the LPs. Input events are queued in order of earliest time-stamp and the following two constraints must be always valid [Zei00]:

- All outputs resulting from the processing of an input event must have a time-stamp greater or equal to the input time. This means processing can't proceed backwards in time.
- Messages must be processed in order of time-stamps in the queues.

Optimistic and conservative schemes differ on the way they enforce the second constraint. In conservative schemes the time-stamped order constraint is never violated. On the other hand, optimistic schemes allow a temporary violation that must be repaired before the final simulation output is presented.

5.1 Conservative synchronization

The conservative approach is illustrated in Figure 13, where there are two *Logical Process* LP1 and LP2 with queues of time stamped messages.

Starting in the upper left corner, LP 1 has a message with timestamp 3 and LP 2 has an earliest message with timestamp 1. Therefore, LP 1 can not execute its message because there is a potential risk of LP 2 producing an output with timestamp less than 3. Conservative schemes must therefore find a way to determine when it is safe to process input events. If a LP has an unprocessed event with timestamp *t* and no event with earlier timestamp can be received, then the event can be safely processed. A LP that has in its queue an unprocessed event from all the other LPs can safely process the one with lowest timestamp because future messages will have a later timestamp. This process can be repeated as long as there are unprocessed messages from all the other LPs. But if this is not so, there is a risk of deadlock.

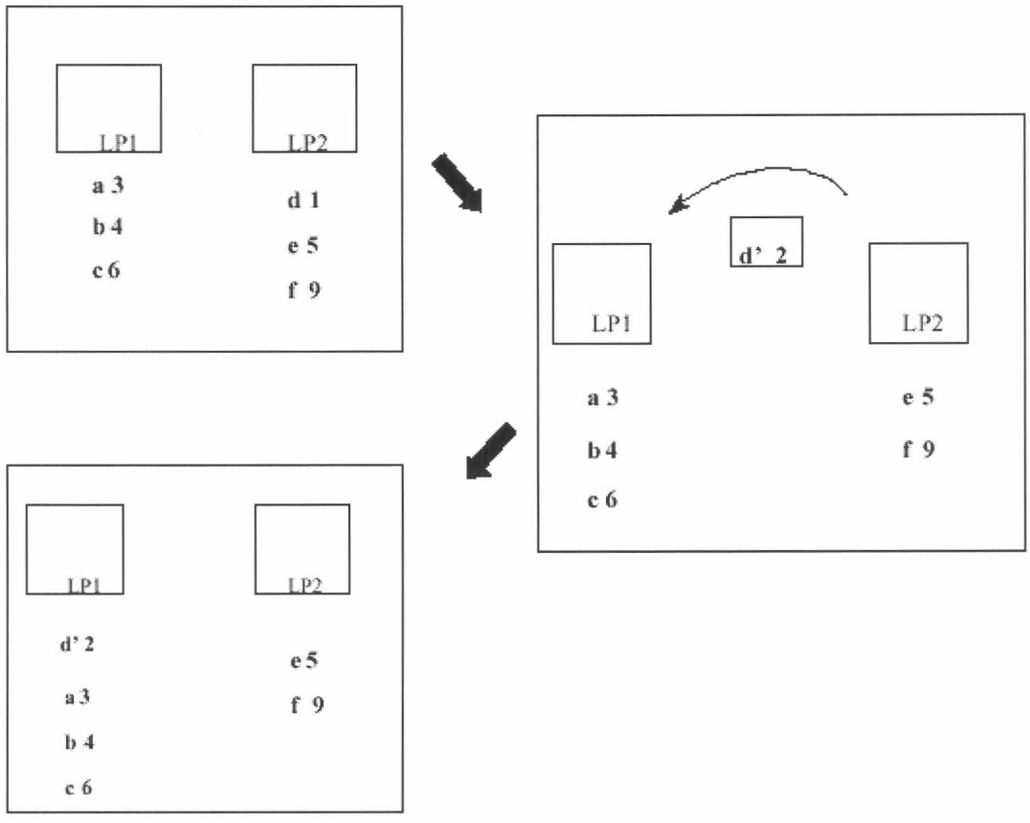


Figure 13: LPs with conservative synchronization [Zei00]

To avoid deadlock, each LP provides a time in the immediate future up to which it promises not to send input events. This is done through null messages. An LP will send a null message to other LPs with a lookahead time up to which it is safe to process messages. In Figure 13, the lookahead for LP 2 is 1. Therefore, when LP 1 receives a null message with this lookahead time, it knows it must not process message (a,3). Large lookahead values are needed to gain advantages over sequential simulation, but unfortunately, such large lookaheads are difficult to find in many representations of reality.

A safe lookahead value is the timestamp of the first unprocessed message in the input queue. If after processing an event all *Logical Processes* send a null message with the timestamp of the next input event, a deadlock will be rare. There is only one case in which a deadlock may occur, and that is the case when all LPs are about to process an input event with the same time stamp.

Null messages can increase the simulation overhead considerably. An improvement on the described mechanism is to send null messages on demand. When a process is about to block, it will request the next

events from the LPs it does not have a timestamp. This reduces the number of null messages being sent, but increases the overhead.

5.2 Optimistic synchronization

The optimistic schemes process their input queues as fast as they can. If a message out of place in the time-stamp order of processing is received, usually known as a straggler, a recovery and rollback mechanism is started to rectify this situation.

Figure 14 shows such a situation. In the upper left hand corner, LP1 and LP2 have arrived at the situation where LP2 has processed events (d,1) and (e,5) and sent input events (d',5) and (e',6) to LP1. Now, LP1 processes event (a,3) which causes it send an input (a',3) to LP2 as shown in the middle. However, since LP2 has already processed event (e,5), the new input (a',3) a straggler.

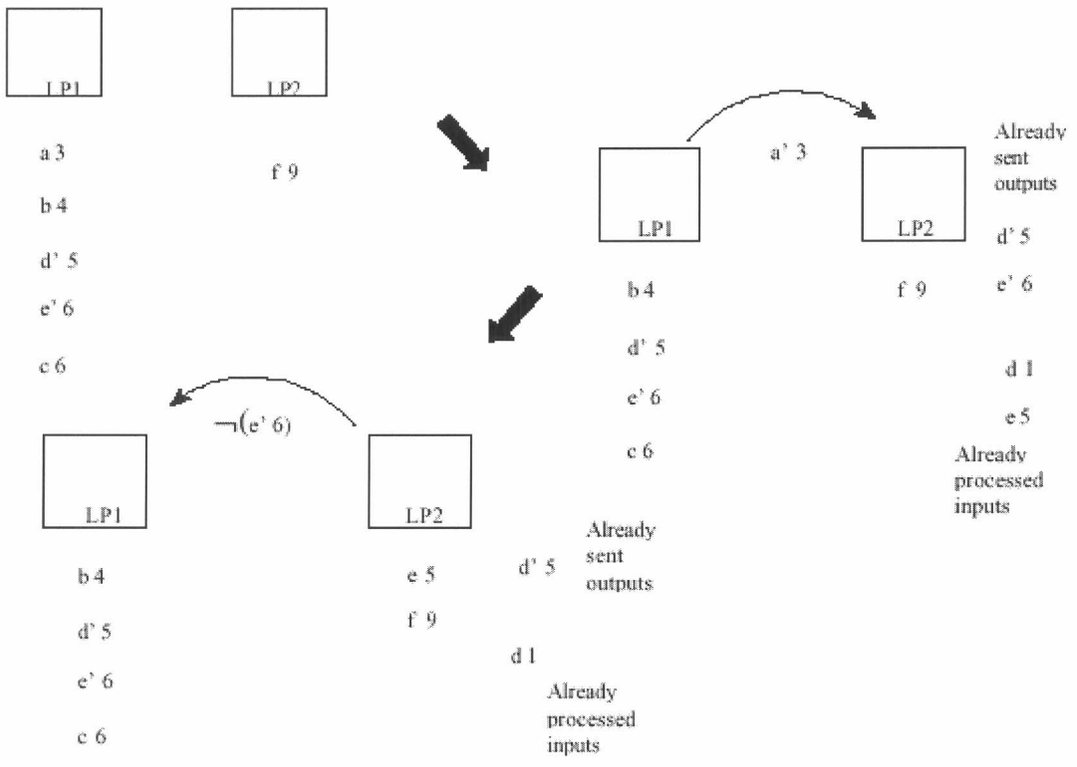


Figure 14: Event processing in an optimistic scheme

To rectify an abnormal situation, an anti-message such as (e',6) that annihilates the effects of already sent messages must be sent. To be able to return to a previous state, each simulation object must maintain a queue of already processed inputs and their outputs, and a queue of previous states. When an anti-message is received, the queues are restored to the anti-message time and new anti-messages are sent for every output sent that should not have been sent. This starts a chain reaction of rollbacks. An optimization technique known as lazy cancellation delays the anti-messages until the simulation object is sure the previous output must be cancelled. It might be the case that the previous and new output are the same, so nothing should be done.

The overhead for running an optimistic scheme is quite considerable. There is a memory overhead because three queue must be kept: input events, output events and state. And there is a processing overhead during rollbacks, too. In addition, a fossil collection mechanism that will delete those queue elements that are no longer required must be conveyed to avoid exhausting system resources. Logical process have a local time know as Local Virtual Time. There is also a Global Virtual Time, the time of the system, that is equal to the least LVT. After a number of simulation cycles, LPs will exchange their

LVTs and the GVT will be determined. This GVT is broadcasted, triggering the fossil collection process on each LP. All those input events, output events and states that have a time-stamp earlier than the GVT can be safely deleted. A high GVT calculation frequency saves memory but generates a big processing overhead. On the contrary, a low frequency will generate less processing overhead and require more memory.

The protocol just described is known as TimeWarp and was proposed by Jefferson [Jeff87]

6
CD++

CD++ [Rod99] is a tool for running DEVS and Cell-DEVS models according to the original DEVS formalism.

The tool is built as a hierarchy of classes, each of them related with a simulation entity. Atomic models can be programmed and incorporated into a basic C++ class hierarchy. Coupled and Cell-DEVS models do not need programming. Instead, the tool provides a specification language that allows the modeler to define the model's coupling, including the initial values and external events, and the local transition rules for Cell-DEVS models.

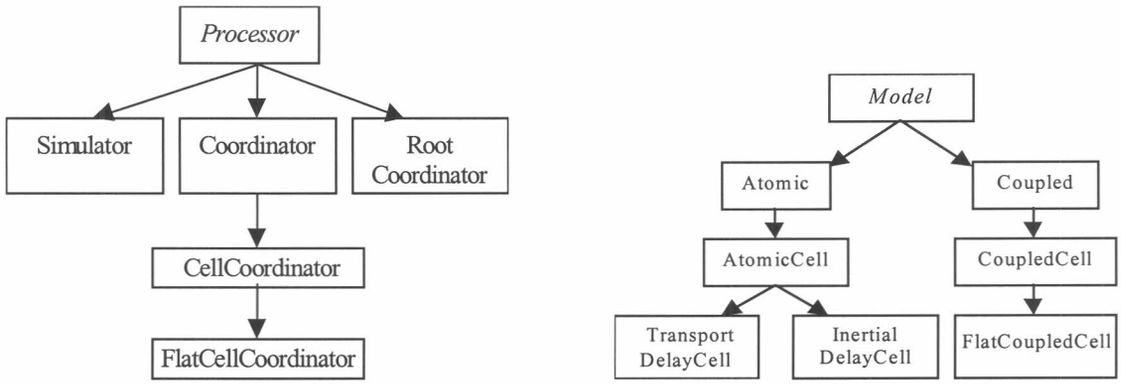


Figure 15: CD++ Models and Processors.

Figure 15 shows the class hierarchy. This class hierarchy implements the model theoretical definition presented in the previous section. New atomic models must be incorporated to the class hierarchy as subclasses of the Atomic Model class.

6.1 Atomic model definition

A new atomic model is created by including a new class that inherits from *Atomic*. In doing so, the following methods may be overloaded:

• ***initFunction***: this method is invoked when the simulation starts. It allows to define initial values and to execute any initialization procedure for the model. When this method is executed, the value of *sigma* next scheduled event) is set to infinite and the model phase to *passive*. The *sigma* variable is used to implement the duration function: it stores the time up to the next event in the model. This variable is related with the elapsed time value, which is maintained by an independent simulation mechanism.

• ***externalFunction***: this method is invoked when an external event arrives from an input port.

• ***internalFunction***: this method is started when the value of *sigma* is zero, since an internal event has occurred.

• ***outputFunction***: this method executes before the internal function, allowing to provide outputs for the model.

After defining these functions, new models can be incorporated to the modelling class hierarchy. Finally, the model must be registered using the method *MainSimulator.registerNewAtomics*(). The following primitives can be used in defining the atomic's model behavior:

• ***holdIn***(state, time): a model executing this sentence will remain in *state* during *time*. When the time is consumed *(sigma* = 0), the model executes the internal transition. This macro was included to make easy the definition of the duration function.

• ***passivate***(): the model enters in passive mode *(phase* = *passive*; *sigma* = *infinite*) and it will be reactivated by an external event.

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

• ***state***(): it returns the present model phase.

7
Parallel CD++

The main goal of this work has been to extend CD++ into Parallel CD++, a tool for the simulation of Parallel DEVS and Parallel Cell-DEVS models on a distributed environment. For this to be accomplished in a modular and portable fashion, a layered architecture was chosen. The topmost layer implements the abstract simulator, the middle layer carries out all required synchronization in the *Logical Process* level, and the lowest layer is in charge of communications.

For the middleware, the Warped project [Mar97] was selected. Warped provides an API for running parallel simulation. Two simulation kernels are currently provided for parallel and distributed simulation: a TimeWarp kernel and a NoTime kernel. The first one implements the TimeWarp protocol as defined by Jefferson's paper [Jeff87]; the second is an unsynchronized kernel.

For the distributed simulation kernels, Warped uses MPI for the message passing. The complete layered architecture is shown in Figure 16.

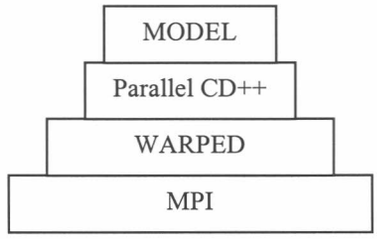


Figure 16 : Parallel CD++ layered architecture

7.1 Synchronization for the Parallel DEVS abstract simulator

To choose between the TimeWarp and NoTime kernels, the abstract simulator of section 4 was analyzed. The following properties were observed:

- During a simulation cycle, all messages carry the same timestamp *t*.
- The root coordinator is the only *Processor* that will advance the time.

In fact, each simulation cycle starts with the root coordinator sending a *(@, t)*. After all the *(done,t)* messages from the child processors have been received, it sends a *(*,t)* message and when all the corresponding *(done,t)* messages are sent back again, the simulation cycle finishes. Only then, the time is advanced.

In the scope of the abstract simulator, a message will only be considered a straggler if its timestamp *t* is less than the LVT of the receiving object. The following lemma holds.

Lemma 3
The abstract simulator of Section 4 can not produce a straggler message.

Proof
Warped and MPI guarantee that when two or more events are sent from a source object S to a destination object D they preserve the same ordering upon arrival to D.

Assume a message *m* with timestamp t_s is sent by a simulation object S to a simulation object D with timestamp t_d, with $t_s < t_d$. Since all messages carry the timestamp of the simulation cycle being executed, it must be the case that the current simulation cycle either corresponds to time t_d or to time t_s.

If it is the first case, i.e. the current cycle's time is t_d, then the root coordinator has sent a message with timestamp t_d. And the root coordinator would only send such a message after receiving a *(done, t_s)* message from all the components that were active at time t_s, and S would have only sent a *(done, t_s)* upon finishing its simulation cycle. The fact that m has time $t_s < t_d$ is a contradiction, because S could have never sent a message timestamped t_s after sending *(done, t_s)*.

Now, if it is the second case, i.e. the current cycle's time is t_s, then it is impossible for D to have a timestamp $t_d < t_s$ because the root coordinator has not yet sent a message with timestamp t_d.

Having proved that the abstract simulator of Section 4 can not produce a straggler message, then no synchronization mechanism at the LP level is needed, because the synchronization is provided by the abstract simulator itself. Then, the NoTime kernel can be used safely.

7.2 Warped API

The Warped system is implemented in C++ and utilizes the object oriented capabilities of the language to provide an application interface. It provides base classes for simulation objects (Warped objects), events and object's states. The user creates its own application by creating new classes that derive from the ones provided. The benefit of this type of design is that the end user can redefine functions without directly changing the kernel code. Though this interface was designed to be used with the TimeWarp protocol, it is simple to switch from one kernel to another. Figure 17 shows the Warped API.

```
class TimeWarp {

      // Methods the user defines
      virtual void initialize();
      virtual void finalize();
      virtual void executeProcess();
      BasicState* allocateState();

      //Simulation kernel services
      void sendEvent (BasicEvent * );
      BasicEvent* getEvent();
};

class BasicEvent {

      int size;
      Vtime sendTime;
      Vtime recvTime;

      int sender;
      int dest;
}

class BasicState {

      BasicState* copyState( BasicState*);
}
```

Figure 17 : Warped API

In Warped, objects are modeled as entities which send and receive events to and from each other, and act on these events by applying them to their internal state. Thus, the kernel provides basic functions for the application to send and receive events. Since the TimeWarp protocol requires periodic state saving for a potential rollback and recovery process, Warped provides an interface for defining each object's state. Other facilities the Warped API provides include the possibility of having user define the data each event will carry.

In return, the user application must provided several functions to the kernel. The most important function defines what each simulation object does in each simulation cycle. Other functions define such things as how to initialize and destroy each simulation object.

7.3 An overview of Parallel CD++

Following the original design of CD++, Parallel CD++ provides an API for users to define new atomic models. The original CD++ atomic's model interface was changed slightly to satisfy the Parallel DEVS formalism. The new interface allows simultaneous external events to be handled together, defines a confluent function and requires the user to give a definition of a model's state (Figure 18).

```
class Atomic {

      // Methods the user should define
      Model& internalFunction();
      Model& externalFunction (MessageBag&)
      Model& outputFunction();
      Model& confluentFunction();
      ModelState* allocateState();

      //Simulation kernel services
      void sendOutput ( Port&, BasicMsgValue* );
      const Vtime& lastChange();
      void holdIn( state, Vtime );
};
```

Figure 18 : The Atomic class

In addition, Parallel CD++ provides a way of allowing the user to define the data carried by output and external events, which in CD++ to real numbers (Figure 19).

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

};

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

        Value v;

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

Figure 19 : The BasicValue class for defining the contents of external and output events.

To run parallel and distributed simulation, it is required that the user defines the set of available machines and a model partition. The set of available machines must be defined as specified by MPI, either by the use of procgroup file or by adding the corresponding entries to machines.ARCH. Details on how this is done are provided in the Parallel CD++ User's guide.

To define the model partition, Parallel CD++ requires that the user specifies a machine for each atomic model. For Cell-DEVS models, the user has to define the location of each cell or cell-range. This is done through a partition file, which is specified as a command line parameter, allowing for the definition of different partitions for the same model.

Parallel CD++ has been compiled and tested with both, the NoTime and TimeWarp kernel. Since the Parallel DEVS abstract simulator provides a synchronization mechanism that guarantees in order execution of events, the NoTime kernel was adopted for the final release, being this kernel more efficient

in the use of system resources. Still, the possibility of changing the Parallel DEVS abstract simulator mechanism for exploiting the full capabilities of the TimeWarp protocol is left open to further exploration.

The NoTime kernel can also be compiled to run in standalone mode without using MPI. Parallel CD++ supports compilation for standalone execution as well.

Further details are provided in the User's guide.

8
Preliminary Results

The main motivation for this work was to make CD++ run faster by means of parallel execution. Therefore, the results to be presented in this chapter will show how execution time of different models changes with different configurations. But as it will be seen, it is not always the case that adding more machines to a simulation will reduce the execution time. After a first set of results was obtained, some bottlenecks were identified in the master-slave abstract simulator of section 4, and a new one, which will be explained in the next section, was proposed.

The simulations were carried out with the Alpha network of the RADS group at the Systems and Computing Engineering Department of the University of Carleton. The Alpha network consists of 14 Pentium machines with 128Mb of RAM running Red Hat Linux 6.2.

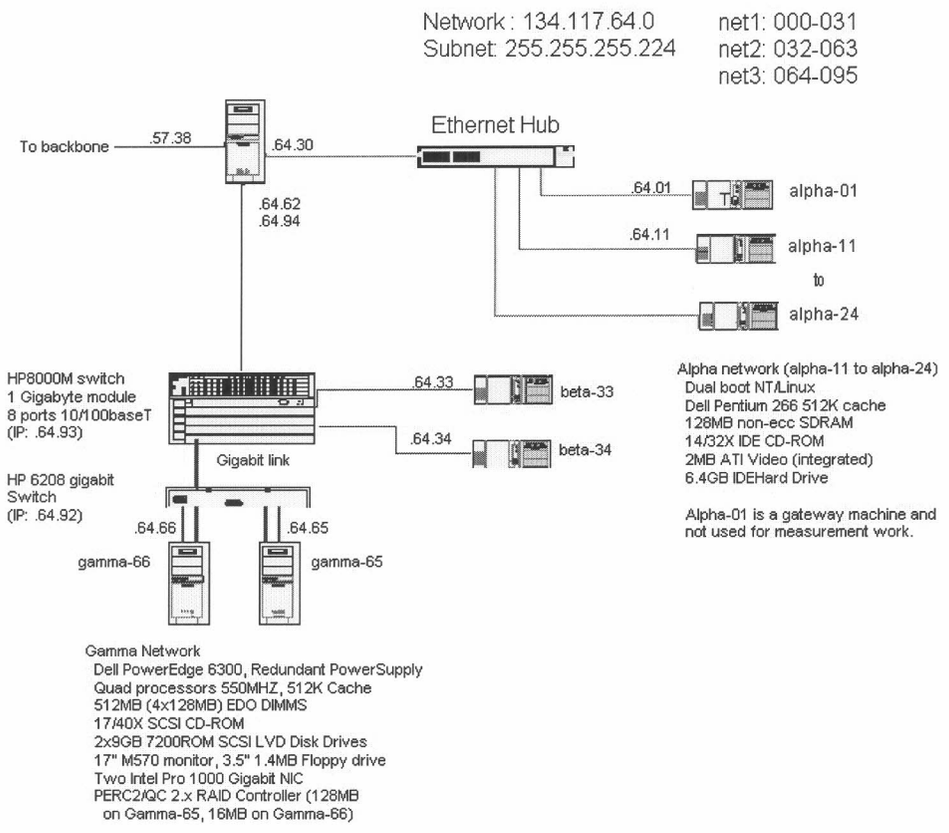


Figure 20: The RADS measurement networks

8.1 An extended version of the GPT model

The parallel simulator was first tested with an extended version of the Generator-Producer-Transducer model (GPT). The GPT model simulates a CPU receiving jobs and calculates its throughput and load. It consists of a generator, a queue, a processor and a transducer, as shown in Figure 21. The generator outputs jobs periodically. When a new job id is sent through the out port, it is received by the queue and the transducer. If the queue is empty, the job will directly be forwarded to the processor; otherwise, the job will be queued till the processor is released. When the processor finishes a job it sends its id through its out port to the transducer and the queue. If the queue has jobs waiting, it will send the next job to the processor. Meanwhile, the transducer will compute the turnaround time and update the throughput and CPU usage values, which it will output periodically.

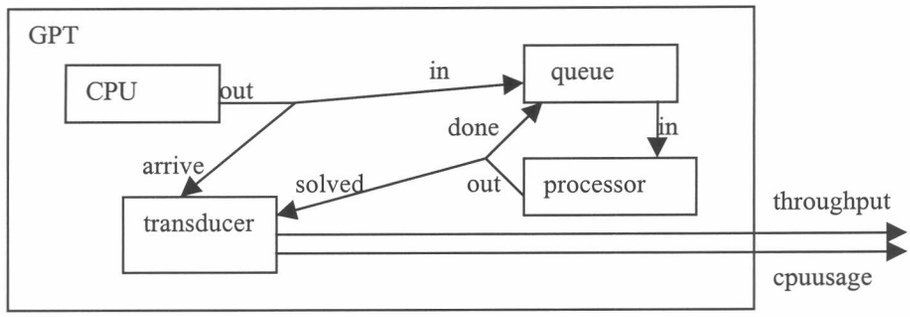


Figure 21: The GPT model

The CD++ definition of this model is shown in Figure 22.

```
00 [top]
01 components : Queue@queue Processor@CPU Transducer@transducer Generator@generator
02 Out : throughput
03 Out : cpuusage
04 Link : out@generator arrived@transducer
05 Link : out@generator in@queue
06 Link : out@queue in@processor
07 Link : out@processor done@queue
08 Link : out@processor solved@transducer
09 Link : throughput@transducer throughput
10 Link : cpuusage@transducer cpuusage

...
```

Figure 22 : Definition of the GPT model

The extended version of the GPT model consists of several instances of the GPT model just shown. In addition, all random variables that were present in the model definition were eliminated to obtain comparable results. Tests were conducted with 12, 48 and 96 instances, running on 1 to 12 machines. Figure 23 shows the execution times for this model.

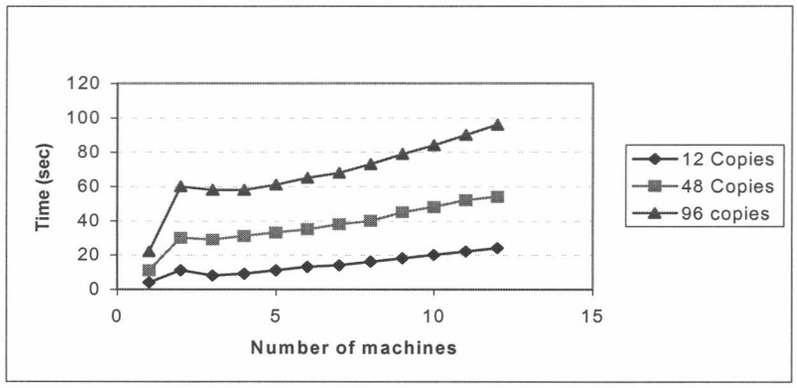


Figure 23: Execution time in seconds of 12, 48, and 96 copies

As it can be seen, the execution times did not behave as expected. As more machines are added, the execution times increases. To verify if the communications overhead was being the cause for such an increase in the running time, the model was rewritten to increase the execution workload. This would increase the computing time at each simulation cycle. If the computing time for a simulation cycle is greater than the communications overhead, then it is expected that adding more machines will reduce the overall simulation time. The new results confirm this hypothesis and are shown in Figure 24.

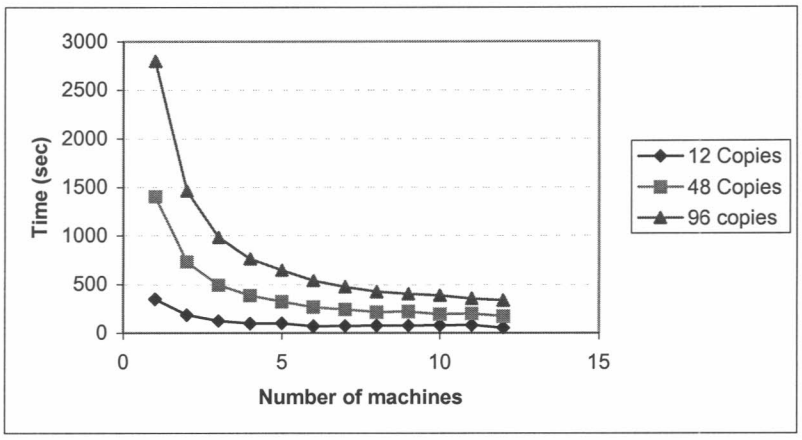


Figure 24: Execution time in seconds of 12, 48, and 96 copies of the GPT model with an increased workload on 1 to 12 machines, for a simulation virtual time of 10 minutes. Results show the minimum time of three runs independent runs.

8.2 Results for a heat diffusion model

The parallel simulator was also tested using a heat diffusion model. In this model, a surface is represented by a 100 x 100 cellular space where each cell contains a temperature value. Initially, all cells have a different value, and as the simulation progresses, the temperature of a cell is updated to the average temperature of the neighborhood.

The model definition using CD++ is shown in Figure 25. Line 2 defines the top model with only one component: the heat surface. Between lines 4 and 16 this model is defined as a 100 x 100 cell space with a standard nine cells neighborhood and a local transition function called heat-rule, which is later defined in lines 18 and 19. The initial values are read form the file calor.map.

```
01   [top]
02   components : surface
03
04   [surface]
05   type : cell
06   width : 100
07   height : 100
09   delay : transport
10   defaultDelayTime  : 100
11   border : wrapped
12   neighbors : surface(-1,-1) surface(-1,0) surface(-1,1)
13   neighbors : surface(0,-1)  surface(0,0)  surface(0,1)
14   neighbors : surface(1,-1)  surface(1,0)  surface(1,1)
15   initialmapvalues : calor.map
16   localtransition : heat-rule
17
18   [heat-rule]
19   rule : { ((0,0) + (-1,-1) + (-1,0) + (-1,1) + (0,-1) + (0,1) + (1,-1) + (1,0) + (1,1)) / 9 } 10000 { t }
```

Figure 25 : Definition of the heat diffusion model

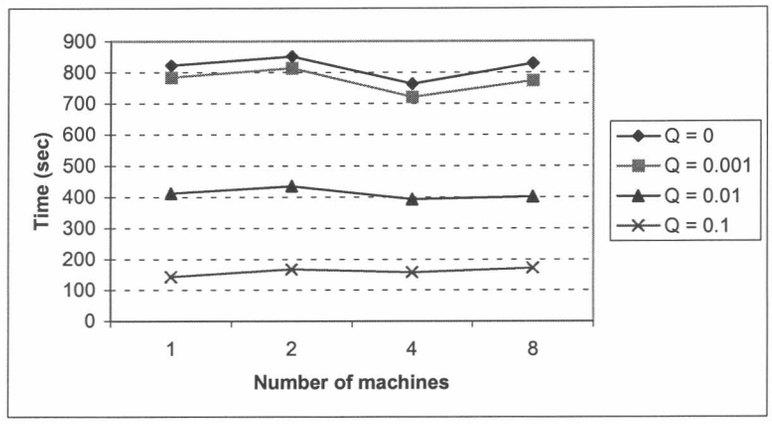
Figure 26 shows a model partition for running the heat diffusion model on 4 machines. A total of 10000 simulators have been assigned to 4 CPUs.

```
01 0 : surface(0,0)..(24,49)
02 0 : surface(25,0)..(49,49)
03 1 : surface(50,0)..(74,49)
04 1 : surface(75,0)..(99,49)
05 2 : surface(0,50)..(24,99)
06 2 : surface(25,50)..(49,99)
07 3 : surface(50,50)..(74,99)
08 3 : surface(75,50)..(99,99)
```

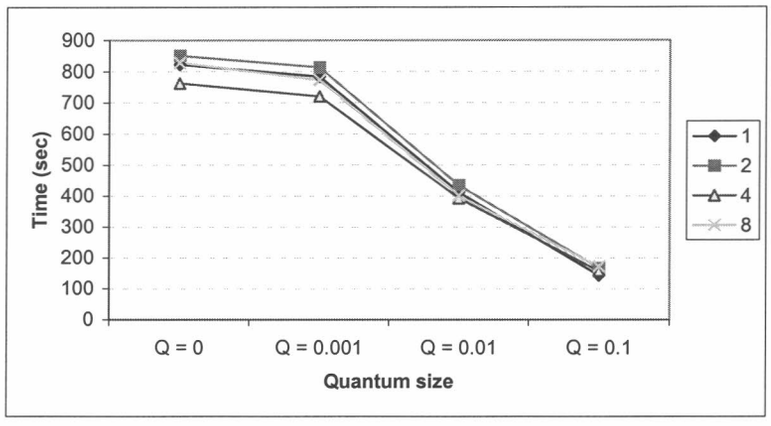
Figure 26 : A model partition for 4 processors

The heat diffusion model was run on 1, 2, 4 and 8 machines, for a virtual time of 2 minutes and using quantum values of 0.001, 0.01 and 0.1. When a quantum size of 0.1 was used, the simulation ended after

a 44sec of simulation time (virtual time) because the model reached a stable state. The execution times are shown in Figure 27.



(a)



(b)

Figure 27 : Execution time for the simulation of the 100x100 heat diffusion model during a virtual time of 2 minutes.

It can be observed that:

- When the same quantum is used, adding more machines does not necessarily reduce the simulation time. For all quantum values, the transition from 4 to 8 machines did not reduce the execution time.
- As the quantum is increased there is a reduction in the execution time (b)

After these results, the abstract simulator of section 4 was studied thoroughly to determine the causes for the unexpected behavior, especially for the time increase observed in the transition from 4 machines to 8 machines. As explained in the next chapter, it was determined that the *master coordinator* was acting as a bottleneck.

9
A revised abstract simulator.

When executing Cell-DEVS models in parallel, there is an invariant that is independent of the abstract simulator being used: adding more machines to a simulation increases the number of cells that have a neighbor running in a different *Logical Process*. As an example, Figure 28 shows how the number of remote cells varies for the 100x100 heat diffusion model.

#Machines	#Cells
1	0
2	400
4	784
8	1168

Figure 28: Number of cells with remote neighbors when different partitions are used.

It is important not to ignore these figures because when a cell sends an output *(y , t)*, this value has to be forwarded to all neighbor cells, which can be local or remote. For remote cells, a message through the network is required. The abstract simulator of section 4, though well suited for dealing efficiently with *(@ , t)*, *(*, t)* and *(done , t)* messages, does not handle *(y , t)* messages very efficiently. In fact, when a *slave coordinator* determines that a *(y , t)* message should be forwarded to another *coordinator*, it just forwards the *(y , t)* message to the *master coordinator* which will then forward it to the corresponding recipients. Thus, an output message whose final recipient is a *slave coordinator* will make two hops: one from the originating *slave coordinator* to the *master coordinator*, and a second one from the *master coordinator* to the final *slave* recipient. Figure 29 shows how an output message from cell (25,0) is forwarded to cell (25,49). The dashed lines represent messages sent over the network.

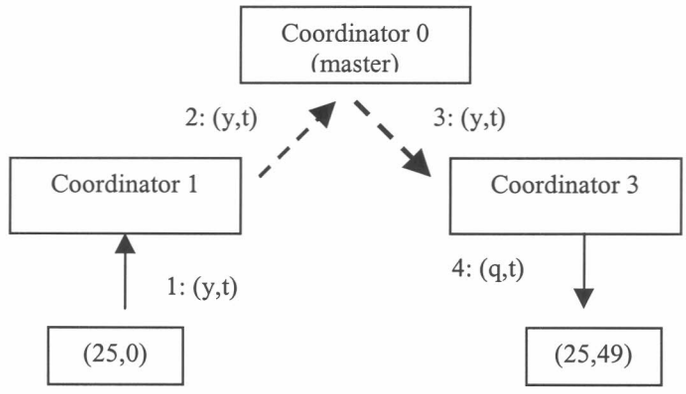


Figure 29 : Master - Slave coordinator output relaying.

This way of relaying messages between *coordinators* has a negative impact on:

- The *master coordinator*, who receives all output messages, even those that are not addressed to models in his *Logical Process*.
- The number of messages being sent over the network, which is almost doubled due to message relaying.

Figure 28 shows that for 8 machines, if all cells have an output to send, then the *master coordinator* will receive 1168 messages. Of these messages, 1022 will then be forwarded to a *slave coordinator*.

To reduce this overhead, a different approach can be taken. When a *slave coordinator* has an output message to a remote model, it could send it directly to the recipient's *coordinator*, without going through the *master coordinator*. In this way, the relaying is avoided, as shown in Figure 30.

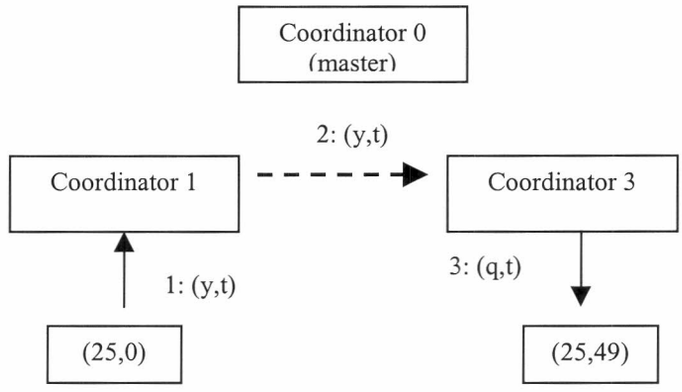


Figure 30: Revised output relaying.

Though simple as it may seem, this new way of relaying messages requires a complete new abstract simulator because it is not enough to change the way output messages are handled.

Section 4 mentioned that during a simulation cycle:

1. A *(@, t)* is sent to all imminent components.
2. All imminent components send their outputs *(y, t)* which are sorted into *(q, t)* messages. Now, all those components that received a *(q, t)* are also imminent.
3. A *(* , t)* is sent to all imminent components.

When centralized relaying of messages is used, the *master coordinator* has complete knowledge of who the active *slave coordinators* are (these are the coordinator that should received the *(* , t)* message). In Figure 29, the *master coordinator* knows that the coordinator 3 will be imminent and should receive a *(* , t)*. Instead, when distributed relaying is used, the *master coordinator* does not know who the active slave coordinators are. As Figure 30 shows, the *master coordinator* does not know *coordinator* 3 has received an output message. If *coordinator* 3 had not received a *(@, t)*, then the *master coordinator* would not know coordinator 3 is now imminent.

The solution to this problem is to have the *master coordinator* send a *(* , t)* to **all** *slave coordinators*. Those that are not imminent would just respond with a *(done, t)* doing nothing else. This would work if the message passing interface (MPI) would guarantee that all messages are delivered in the same order they are sent. But unfortunately, this is not so. MPI can guarantee that if two messages are sent from *Logical Process* 1 to *Logical Process* 2, they will arrive in the same order they were sent. But if two messages are sent from *Logical Process* 1 to *Logical Process* 2, and a third message is sent from *Logical Process* 1 to *Logical Process* 3, there is no guarantee those two first messages will arrive before the third one. This can lead to a special situation were a *(* , t)* is received before a *(y , t)* message as shown in Figure 31.

In Figure 31, *coordinator* 1 first sends a *(y,t)* message to *coordinator* 3 and then a *(done, t)* message to *coordinator* 0. However, the *(done, t)* message is received before *coordinator* 3 receives the *(y,t)* message. Then the *master coordinator* sends a *(done, t)* and receives a *(*,t)* that is forwarded to *coordinator* 1 and 3. *Coordinator* 3 may end up receiving the *(*,t)* message before the *(y,t)* message, producing incorrect results. The problem here is that no *coordinator* knows when all the sorting of output messages has concluded. This was not a problem with the abstract simulator of section 4 because the *master coordinator* did know.

A correct abstract simulator would delay the *(done, t)* messages until all outputs have been received. One first solution would be to acknowledge a *(y,t)* message, but this again, leads to an enormous number of messages being sent.

Instead, the following approach will be taken:

2. When a *slave coordinator* receives a *(@, t)*, any generated *(y, t)*messages will directly be sent to the corresponding *coordinator*.
3. After a *slave coordinator* has sent all the *(y, t)*, a new *($, t)* message will be sent to all the other *coordinators* (except to the master). This new message, called output synchronization, is a way of telling the other *coordinators* that no more output messages will be sent.
4. After a *slave coordinator* has sent all its *($, t)* messages and received the *($, t)* messages from the other *coordinators*, a *(done, t)* message will be sent.

In this way, when the *master coordinator* receives all the *(done, t)*, a *(*, t)* message can be safely sent.

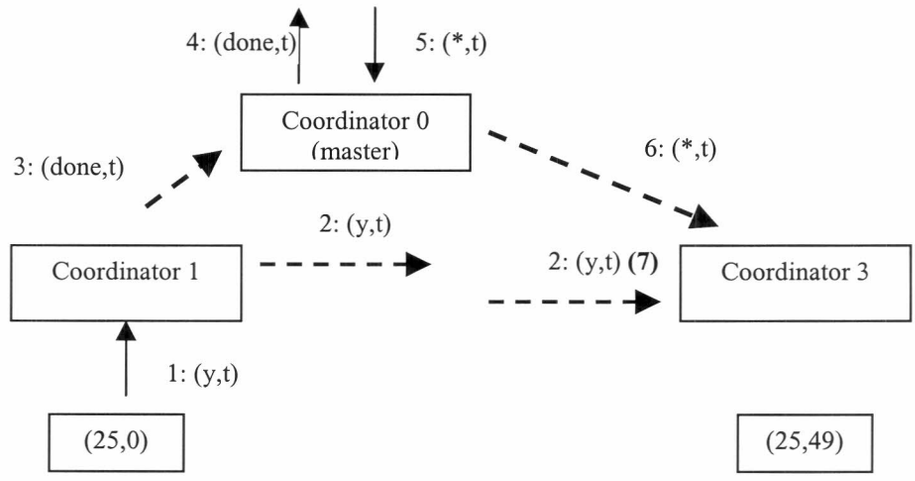


Figure 31: A (*,t) message is received before a (y,t) message

9.1 Revised abstract simulator

The new abstract simulator will be defined next. Only message handlers that have changed will be presented.

MASTER COORDINATOR

when a (@ , t) message is received from parent coordinator

    if $t = t_N$ **then**

        $t_L := t$

        for all slave processors i

            send (@, t) to slave i

        end for

        for all other imminent child processors i with minimum t_N

            send (@, t) to child i

            cache s in the *synchronize* set

        end for

        wait until (*done*, t)'s have been received from all imminent processors

        send (*done*, t) to parent coordinator

    end if

    else raise error

end when

The *master coordinator* will send a (@ , t) message to all *slave coordinators*, whether they are imminent or not. Then, when all (*done*, t) messages have been received it can be sure output sorting has finished.

MASTER COORDINATOR

when a (y , t) message is received from child i

for all influencees, j of child i

if j is a local processor

$q := z_{i,j}(y)$

send (q, t) to child j

cache j in the *synchronize* set

else

$s :=$ coordinator(*self*, j)

if $s \notin$ *slave-sync* set then

send (y, i, t) to s

cache s in the *slave-sync* set

end if

end if

end fo

if $self \in I_i$ (y is to be transmitted upward) **then**

$y := z_{i,\ self}(y)$

send (y, t) to parent coordinator

end if

clear *slave-sync* set

end when

when a (y , i, t) message is received from a slave s

cache s in the *slave-sync set* and proceed as if a (y , t) message had been received from child i

end when

For (y , t) messages, the *master coordinator* behaves almost as previously defined. The only difference is that it is no longer necessary to cache s in the *synchronize* set.

MASTER COORDINATOR

when a (* , t) message is received from parent coordinator

if $t_L \leq t \leq t_N$

for all $q \in bag$

for all receivers of q, $j \in I_{self}$

if j is a local processor

$q := z_{self,j}(q)$

send (q, t) to j

cache j in the synchronize set

else

$s :=$ coordinator(*self*, j)

if $s \notin$ *slave-sync* set then

send (q , t) to s

cache s in the *slave-sync* set

end if

end if

end for

clear *slave-sync* set

end for

empty *bag*

for all slave processor s

send (*, t) to s

end for

for all i in the *synchronize* set

send (*, t) to i

end for

wait until all (*done*, t_N)'s are received

$t_L := t$

$t_N :=$ minimum of components' t_N's

clear the *synchronize* set

send (*done*, t_N) to parent coordinator

else raise an error

end when

As with (@ , t) messages, the new *master coordinator* will forward (* , t) messages to all *slaves* because there might be some *slaves* that will execute an external transition which the *master coordinator* does not about.

The new *slave coordinator* will be described next. Quite a few changes have been introduced. To begin with, a *slave coordinator* will receive (@ , t) and (* , t) messages even if it is not imminent, so a check should is done to determine if the *slave coordinator* should actually do something. Second, outputs to remote cells are routed to the corresponding *coordinators*. And finally, a new message, ($, t) has been added and should be handled.

SLAVE COORDINATOR

when a (@ , t) message is received from master coordinator

if $t = t_N$ then

$t_L := t$

for all imminent child processors i with minimum t_N

send (@, t) to child i

cache i in the *synchronize* set

end for

wait until (*done*, t)'s have been received from all imminent processors

send (*$*, t) to all slave coordinators

wait until (*$*, t)'s have been received from all slave coordinators

send (*done*, t) to master coordinator

else

send (*$*, t) to all slave coordinators

wait until (*$*, t)'s have been received from all slave coordinators

send (*done*, t) to master coordinator

end if

end when

The *slave coordinator* should only respond to a (@ , t) with a (*done*, t) when it can assure it will receive no longer receive (y , t) messages from other slaves. To ensure this, (*$*, t) messages where introduced. When a *slave coordinator* has received a (*done*, t) from all its child processors it can be sure it will not send any more (y , t) messages. Hence, it sends all other *slaves* a (*$*, t) message to indicate this condition. Then a *slave coordinator* should wait to receive all (*$*, t) messages before sending a (*done*, t) message.

The event that has been changed the most is when outputs messages (y , t) are received. Again, it is necessary to distinguish two types of output messages: those that are received from local children *processors* and those that are received from the *master* or other *slaves coordinators*.

SLAVE COORDINATOR

when a (y , t) message is received from child i

for all influencees, j of child i

if j is a local processor

$q := z_{i,j}(y)$

send (q, t) to child j

cache j in the *synchronize* set

else

$s :=$ coordinator(*self*, j)

if $s \notin$ *slave-sync* set then

send (y , t) to s

cache s in the *slave-sync* set

end if

end if

end for

if *self* $\in I_i$ (y is to be transmitted upward) **then**

if parent $\notin$ *slave-sync*

send (y, t) to master coordinator

end if

end if

clear *slave-sync*

end when

when a (y , i, t) message is received from *master* or *slave* coordinators

for all influencees, j of child i

if j is a local processor

$q := z_{i,j}(y)$

send (q, t) to child j

cache j in the *synchronize* set

end if

end for

end when

When an output event is received from a child i, the *slave coordinator* sorts the message to the influencees of i. If any influencee is local, the z function is applied a (q , t) message is sent. If there are non-local influencees, then the output event is sent to the corresponding *coordinator*. Because these non-local influencees can be under the same *coordinator* care should be taken to avoid forwarding duplicate (y , t) messages. This accomplished using a *slave-sync* set which keeps track of those *coordinators* that have received the message. Output messages received from other *coordinators* are sorted as external events to the local children.

It remains to define how the *slave coordinator* will handle internal transition messages. However, there will be no changes here. It might be the case that a *slave coordinator* that receives a (*, t) message has no imminent dependants, but in this case, the *synchronize* set will be empty and the existing procedure will work.

SLAVE COORDINATOR

when a (* , t) message is received from master coordinator

if $t_L \leq t \leq t_N$

for all $q \in bag$

for all receivers of q, $j \in I_{self}$

if j is a local processor

$q := z_{self,j}(q)$

send (q, t) to j

cache j in the synchronize set

else

do nothing

end if

end for

end for

empty *bag*

for all i in the *synchronize* set

send (*, t) to i

end for

wait until all (*done*, t_N)'s are received

$t_L := t$

$t_N :=$ minimum of components' t_N's

clear the *synchronize* set

send (*done*, t_N) to master coordinator

else raise an error

end when

9.2 Synchronization for the revised Parallel DEVS abstract simulator

In Section 7 it was proved that the abstract simulator of Section 4 could be executed using no synchronization at the logical process level (i.e. without producing a straggler message). For the new abstract simulator to be able to execute using the same unsynchronized protocol, it should be proved that it will not produce a straggler message.

As defined earlier, a straggler message is a message whose timestamp is less than the recipients current time. For this definition, the proof of section 7 still holds, so the Warped NoTime kernel can still be used. It is important to notice however, that the absence of straggler messages does not guarantee correctness during a simulation cycle.

During a simulation cycle all messages carry the same timestamp, so there are no stragglers at the logical process level. However, as it was noticed in Figure 31, there was a potential risk of having straggler messages at the application level, i.e. a message that has a correct timestamp but that is received out of phase. This was corrected introducing the ($, t) messages.

10
Performance analysis

In this section further results about the performance of Parallel Cell-DEVS models will be presented. First, the effects of quantization will be discussed. Following, the performance of the revised simulator will be compared against the performance of the original one. Then, other factors affecting the performance of distributed simulation will be introduced. In particular, model partitioning and workload will be analyzed.

10.1 The effect of quantization

The use of a quantizer reduces significantly the number of active cells. Because most of the results in this section will be related to the heat diffusion model, it is important to understand how the use of a quantizer impacts on model execution.

Figure 32 shows how the average number of active cells on each simulation cycle of the 100 x 100 heat diffusion varies with the quantum size.

Quantum size	Active cells
0	10000
0.001	9975
0.01	6262
0.1	1922

Figure 32 : Average number of active cells in each simulation cycle

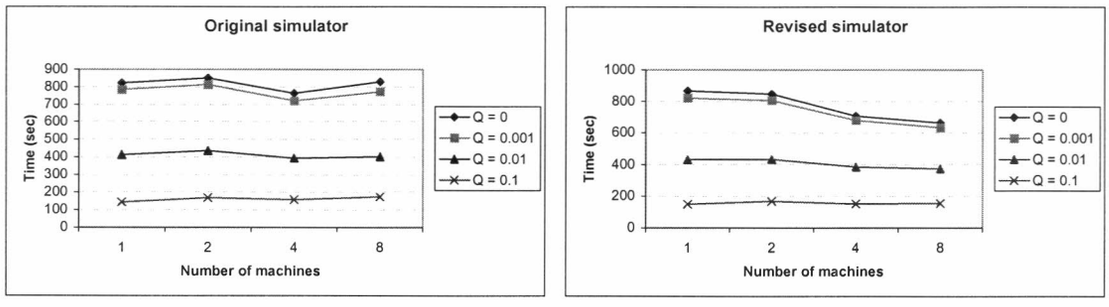
When no quantum is used, all 10000 cells are active during each cycle. As the quantum size increases, cells reach a stable value more rapidly and then there are less active cells. This number of active cells has a direct impact on execution time because:

more active cells = more workload = more time

Having understand this, it will possible then to discuss how the workload affects the performance of distributed execution.

10.2 Revised simulator vs Original simulator

Figure 33 shows the execution times for the heat diffusion model using both, the original and the revised simulators.



(a) *(b)*

Figure 33: Execution times of the heat diffusion model using the original and revised simulators.

As it is shown, the revised simulator improved the execution times considerably, especially for 4 and 8 machines and quantum sizes of 0 and 0.001.

Figure 34 shows a further comparison between the original and revised simulators.

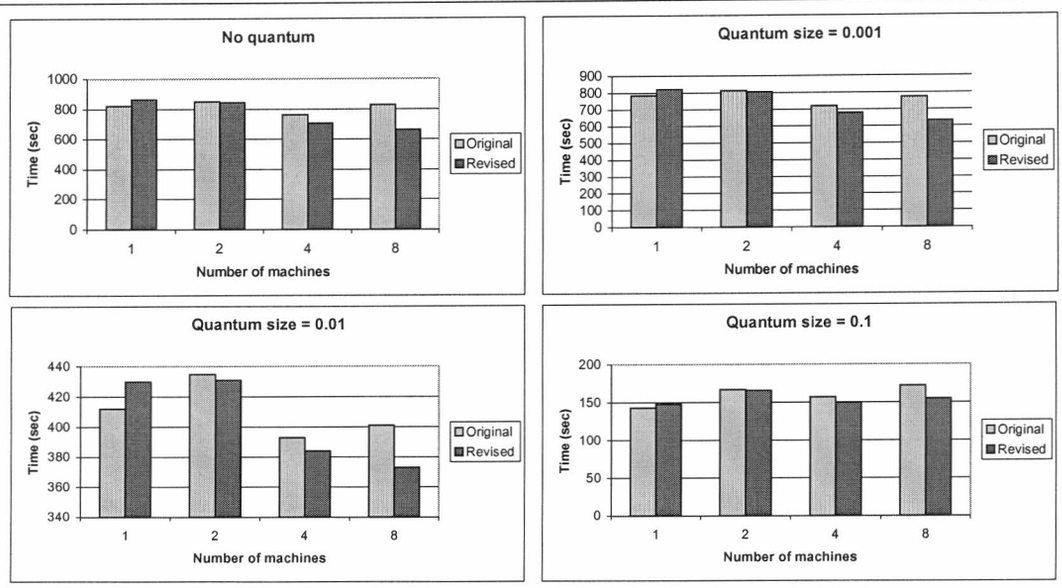


Figure 34: Further comparisons between the original and revised simulators

Again, the revised simulator outperformed the original one for all quantum sizes. Some exceptions are observed for executions on 1 and 2 machines. For 4 and 8 machines, the improvement is significant for heavy load models, such as those of quantum size 0 and 0.001.

10.2 The effect of the choice of partition

Another factor that can affect the performance of the parallel simulator is the choice of model partition. A fair partition distributes the execution load evenly through out the machines. So far, all the results shown for the heat diffusion model were obtained with a fair partition. Tests were also conducted with a set of uneven partitions as shown in Figure 35.

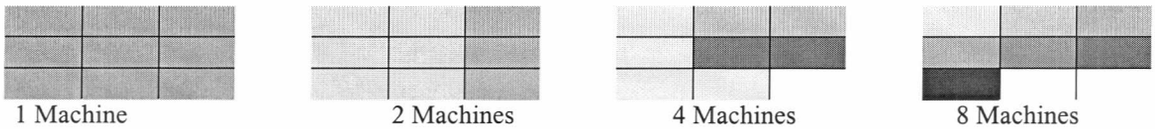


Figure 35: Uneven partitions: each colored area runs on a different machines

The execution times when this set partitions is used are shown Figure 36.

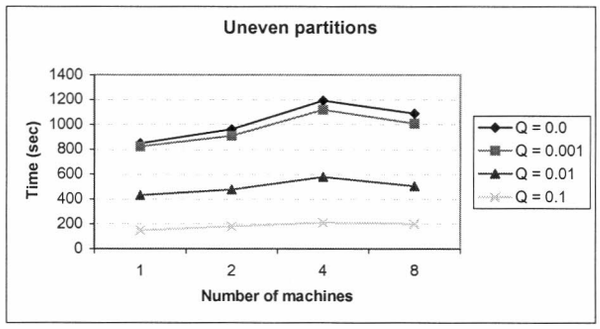


Figure 36 : Execution times for uneven partitions

As expected, the choice of uneven partitions causes a slowdown as more machines are used. There is one exception, this is the transition from 4 to 8 machines. The reason for this is that the partition for 8 machines is almost even, as it will be shown next.

10.4 A metric for model parallelism

To asses if a model is suitable for parallel execution, a parallelism metric has been developed. This measure has its greatest value when all of the machines have the same load, i.e. there is simultaneous execution; and its least value when all the simulation is done by only one of the available machines.

In Parallel DEVS, one way to determine how much activity there is on each simulation cycle is to count the number of received *(*, t)* messages. If in addition this information is obtained for each logical process and each simulation cycle, a clear picture of how much activity is taking place can be drawn. The expression

$$Count(LP_{num}, t)$$

will be used to denote the number of *(*, t)* messages received by LP number LP_{num} during the simulation cycle at time *t*.

But counting the messages by itself does not give the sort of metric being sought, it just gives the number of messages. For a better measure, it can be assumed that the processing of each *(*, t)* will require the same computing time. Then, assuming also an homogeneous set of machines, the execution time for each simulation cycle will be given by

$$CycleTime(t) = c \cdot Max_{i=0}^{NumLPs-1}\left(Count(LP_i, t)\right)$$

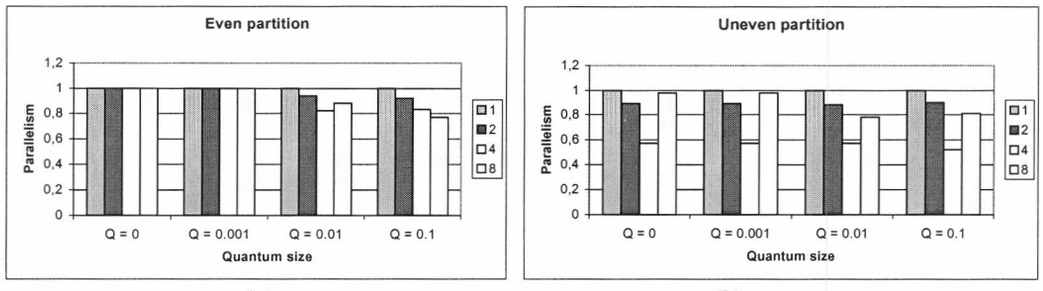
That is to say, the execution time of a simulation cycle will be equal to the time it takes the LP that receives the highest number of *(*, t)*. The constant *c* is the time it takes for a cell to process a *(*, t)*. For the purpose of evaluating how even the workload is, it can be ignored. Once the cycle time is known, the CPU usage at each LP can be obtained by dividing the used time by the cycle time

$$Usage(LP_{num}, t) = \frac{Count(LP_{num}, t)}{CycleTime(t)}$$

The LP with the maximum number of messages will have a usage measure of 1. If all the LPs receive the same number of *(*, t)* messages then all LPs will have a CPU usage of 1, being this the case of maximum parallelism. The CPU usage of all LP's can be averaged to give a measure for the whole system.

The parallelism metric will depend on two factors: the model and its partition. For maximum parallelism to be achieved the model has to be partitioned in a way that all LPs will have an equal number of active models. For some models, such a partition might exist, but for some others, it might not. Most probably, the load of each LP will vary with time. Model partitions in Parallel CD++ are static.

This metric was used to asses the suitability of the heat diffusion model for parallel execution. The results are shown in Figure 37.



(a) *(b)*

Figure 37: Parallelism metric for the heat diffusion model

Figure 37 (a) shows the parallelism value for the heat diffusion model when the even set of partitions are used. The executions with quantum sizes of 0 and 0.001 show a perfect value of 1. This means that all

Logical Process have the same number of active cells during the whole simulation and maximum parallelism is achieved. Highly parallel models should show a reduction in the execution times as more machines are added. As Figure 33 shows, this is the case.

When quantum sizes of 0.01 and 0.1 are used, a reduction in the model parallelism is observed. This is so because as the simulation progresses there are more cells that reach a stable state and become inactive, and the distribution of this inactive cells is not necessarily even.

Figure 37(b) shows the parallelism value when the uneven partition of Figure 35 is used. It can be observed that the parallelism is drastically reduced for 2 and 4 machines, and gets high again for 8 machines. The uneven partitions cause an uneven distribution of the load and hence a reduction in the parallelism. This causes a slowdown in the model execution as it was shown in Figure 36. The partition for the 8 machines case is not so uneven, as the parallelism metric shows.

As a subproduct, the evolution of the parallelism through the whole simulation process can be obtained. Figure 38 shows the how the parallelism evolves when the even set of partitions is used.

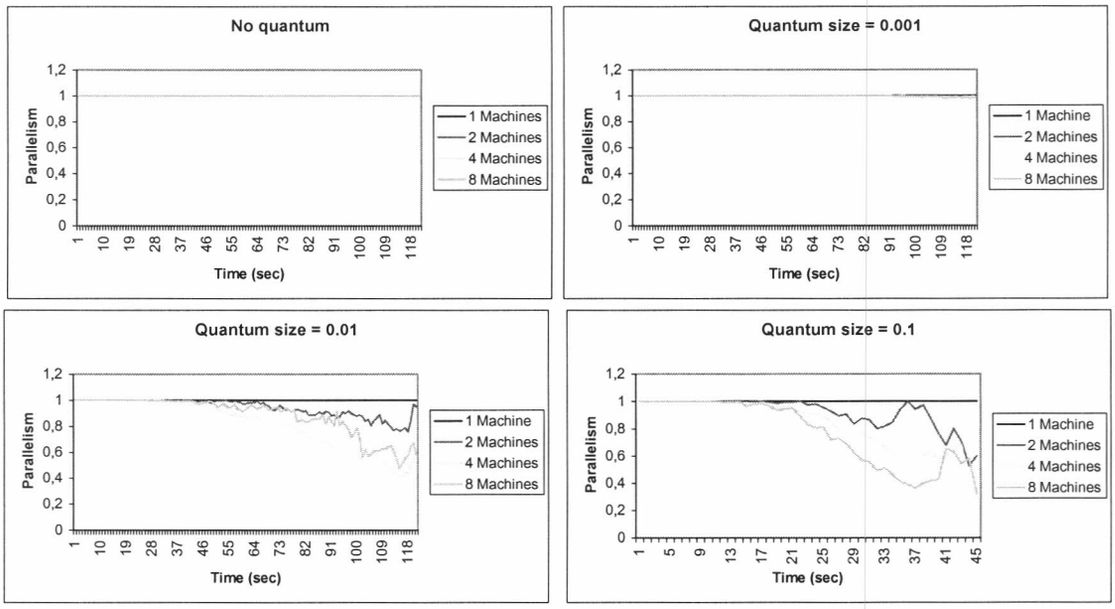
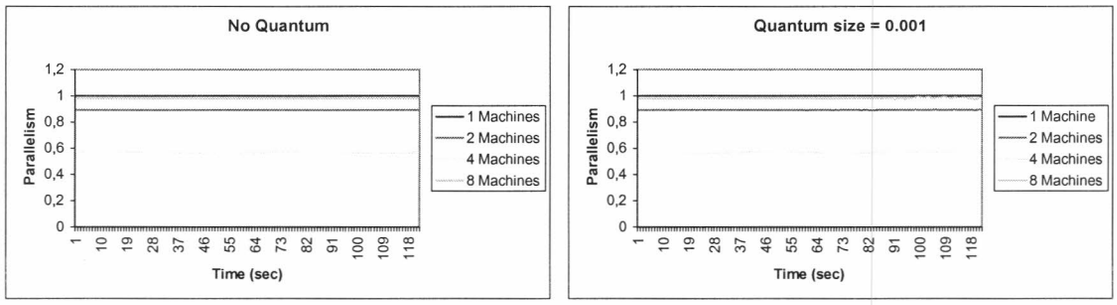


Figure 38 : Parallelism evolution for the heat diffusion model when even partitions are used.

For quantum sizes of 0 and 0.001, the parallelism keeps a constant value of 1 during the whole simulation. This is not the case for quantum sizes of 0.01 and 0.1. In this cases, the parallelism starts at 1 and then varies as cells reach a stable value.

Figure 39 shows how the parallelism evolves for the heat diffusion model when the set of uneven partitions is used.



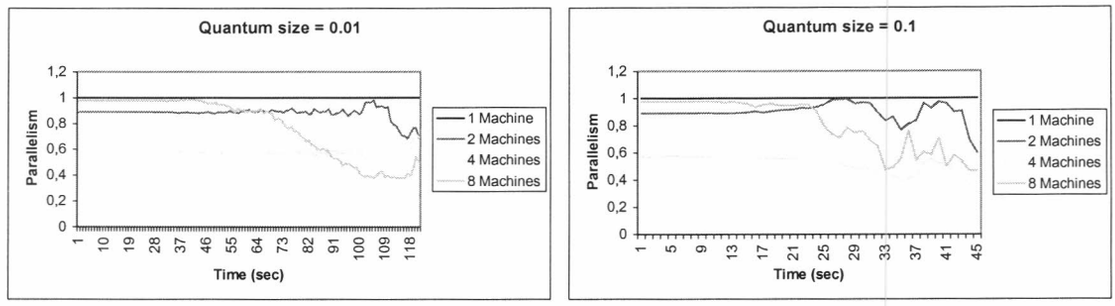


Figure 39: Parallelism evolution for the heat diffusion model when uneven partitions are used.

Here, for quantum sizes of 0 and 0.001, the parallelism value also keeps constant during the whole simulation, but not with a value of 1. The behavior for quantum sizes of 0.01 and 0.1 is similar to the one observed in Figure 38.

To conclude this section, it is important to mention that the parallelism metric so far discussed is only another tool to asses how well a model may execute in parallel. This metric does not take into account network traffic, which for certain models and partitions, may impact more on performance than an even workload distribution.

11
A flow-injection Cell-DEVS model

In this section, a Cell-DEVS model for a flow-injection system will be presented. This model has been developed together with people working at the Laboratorio de Análisis de Trazas – Facultad de Ciencias Exactas y Naturales – Universidad de Buenos Aires.

11.1 Flow injection analysis

Flow-injection methods are analytical methods used for automated sample analysis of liquid samples. In a flow injection analyser, a small, fixed volume of a liquid sample is injected as a discrete zone using an injection device into a liquid carrier which flows through a narrow tube. As a result of convection at the beginning, and later of axial and radial diffusion, this sample is progressively dispersed into the carrier as it is transported along the tube. The addition of reagents at different confluence points (which mix with the sample as a result of radial dispersion) produces reactive or detectable species which can be sensed by flow-through detection devices. Figure 40 presents a simple flow-injection apparatus.

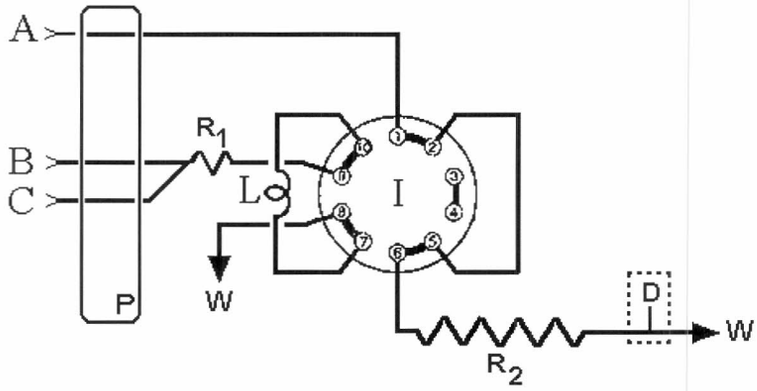


Figure 40 : A FIA manifold.

This device (called a FIA manifold) consists of a peristaltic pump (P) that adds carrier solution (A) into a valve (I) that connects to a tube called a reactor (R2). At the end of the tube a detector is placed to sense a specific property of the flowing solution. The valve can be turned to allow the flow of the sample (B) into the reactor. The sample is held in the loop L and when the valve is rotated its contents flow into the reactor, where chemical activity will usually take place between the sample and the carrier solution. As a result, a change will be observed in the signal produced by D, making it possible to quantify the sample after comparing the results with those obtained by known samples.

In a FI system convective transport yields a parabolic velocity profile with molecules at the tube walls having speed zero and those at the center having twice the average velocity. At the same time, the presence of concentration gradients develops axial and radial diffusion of sample molecules. It has been reported that in FI systems of practical interest, axial molecular diffusion has almost no influence in the overall dispersion, but radial diffusion is the main contributor. For a pump proving a net flow of q ml/min in a coil of radius a, the average flow velocity is given by:

$$V_a = \frac{q}{60 \cdot (\pi \cdot a^2)} \qquad \textbf{(Equation 1)}$$

At a point at distance r from the center, the flow velocity is described by:

$$v(r) = 2 \cdot V_a \cdot \left(1 - \frac{r^2}{a^2}\right) \qquad \textbf{(Equation 2)}$$

As mentioned in [AIT98], it is very difficult, if not impossible, to correlate the experimentally obtained response curve with the actual spatial mass distribution of the system. This is a consequence of the selected method of measurement, which fixes spatially and temporally the point of detection. Under these circumstances, any event occurred before the detection point is inferred from the response curve profile. Therefore, this detection approach is a powerful tool for predicting response curves, but ignores the processes leading to the generation of such response. In [AIT98] a method for continuously monitoring a FI system was proposed. A FI system using nitric acid as the carrier solution, water as the injected sample and a digital conductimeter with a couple of wires at both ends of the carrier stream detector was used to follow the radial mass distribution of the sample zone.

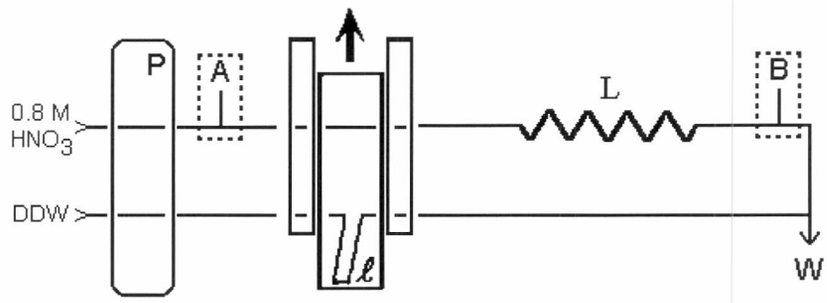


Figure 41: FIA manifold for continuously monitoring. P = pump; l = loop; L = reactor; W=waste; A, B = detection points. Punctual detection: suitable detector in point B; integrated detection: Pt wires located at points A-B. [AIT98]

When the water sample is injected, it acts as a blocking disc, and no electric conductance is measured. As convective transport and diffusion gradient forces the water sample to be released from the walls, causing a reduction of the blocking area and allowing electric current to flow, conductivity values different from zero are measured. Figure 42 shows the characteristic conductivity curve obtained by such a system.

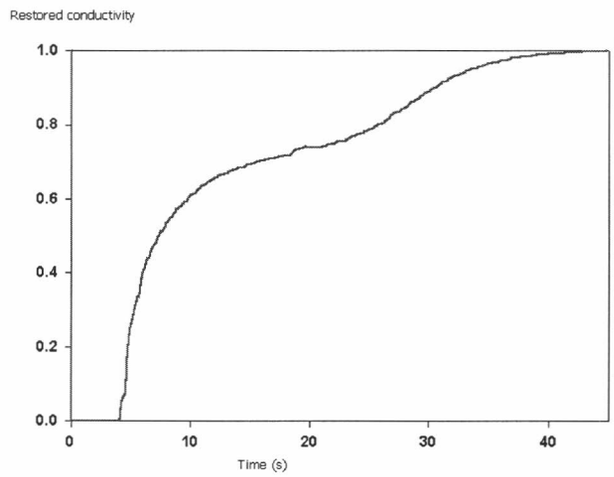


Figure 42: Characteristic conductivity curve [AIT98]

11.2 A Cell-DEVS model for flow-injection

As mentioned, it is impossible to analyze the detailed behavior of the changes in the mass distribution profile. Therefore, we decided to build a Cell-DEVS model describing the integrated conductivity flow-injection system (ICM) in detail. In this way, the internal complex behavior can be analyzed by studying the simulated results. The ICM system consists of a 0.025 cm radius tube, a 10.75 cm loop and a 9,25 reactor coil . We assumed the total tube length of the tube to be of 20cm. For this system, a cell space of 25 rows and 200 columns was defined, each cell representing a 0.001 x 0.1cm of a half tube section. Row 0 represents the center of the tube and row 24 the section of the tube touching its walls and the value of each cell will represent the nitric acid concentration.

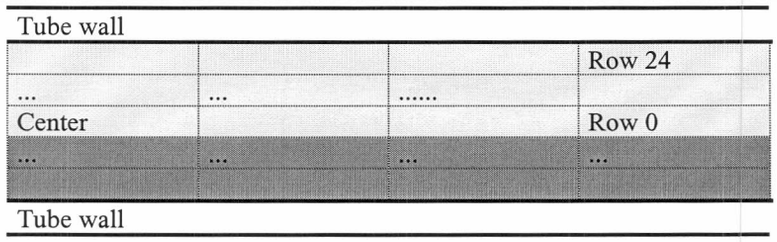


Figure 43 : Correspondence between the cell-space and the actual tube

Figure 43 shows in light gray a tube section representing a cell. This is a longitudinal cut of the tube. The final aim is to build a 3 dimensional space representing a cylindrical section of the tube, but in this case each cell represent a flat section.

To deal with convective transport and radial diffusion at the same time, the model reacts in two phases: transport and diffusion. The local computing function simulates the transport phase, and all cells are connected to an external generator sending an event which triggers the diffusion phase. The model is built as a coupled DEVS model with two components: a Cell-DEVS (named *fia*) representing the tube, and an atomic model (named *generator*). The generator has one output port (*out*) to send the diffusion triggering event. This port is mapped to the *diffuse* input port of the *fia* model (line 2). This means all ouput events sent through the *out* port will be received as external events by the *fia* model through the *diffuse* port.

```
00 [Top]
01 components :  fia generator@ConstGenerator
02 link : out@generator diffuse@fia
03
04 [generator]
05 frecuency : 00:00:00:014
```

Figure 44 : Components of the DEVS model

The frequency of diffuse events is defined by Equation 3. This equation computes the the characteristic distance a particle of a given solution of diffusion coefficient c will travel in dt seconds.

$$ds = \sqrt{2 \cdot c \cdot dt} \qquad \textbf{(Equation 3)}$$

Solving the equation for $c = 3{,}5 \times 10^{-5}$ cm/s and $ds = 0.001$ cm, we obtain a dt of 14ms. We used for the ds value the cell height to find out how long it would take for two cells to diffuse homogeneously. We did not take into account the cell width because axial diffusion can be ignored.

```
05 [fia]
06 in : diffuse
07 width : 200
08 height : 25
09 delay :   inertial
10 border :   nowrapped
11 neighbors : fia(-1,-1) fia(-1,0) fia(-1,1)
12 neighbors : fia(0,-1) fia(0,0) fia(0,1)
13 neighbors : fia(1,-1) fia(1,0) fia(1,1)
14 localtransition :  transport
```

Figure 45 : Definition of the FIA coupled cell model

Figure 45 shows the definition of the parameters for the coupled Cell-DEVS *fia*. Line 6 defines the *diffuse* input port, and lines 7 and 8 define the cell space dimensions. Line 9 sets the cell delay type to inertial. An inertial delay cell that has a scheduled future value f will preempt this value if upon receiving an external event and evaluating the local transition rules a new future value f_1, with $f \neq f_1$, is obtained. In this case, f_1 will be scheduled as the future value with a given delay d. Line 10 defines non-wrapped borders and lines 11 to 13 define a cell's neighborhood shape. Finally, line 14 defines the sets the local transition function rules, which is defined in Figure 45.

```
18 [transport]
19 rule : { (0,-1) } { 0.1 / ( 22.57878 * ( 1 - power( cellPos(0) * 0.001 + 0.0005 , 2)
        / 0.000625 )) * 1000 } { cellpos(1) != 0 }
20 rule : { 0.8 } { 0.1 / ( 22.57878 * ( 1 - power( cellPos(0) * 0.001 + 0.0005 , 2) /
        0.000625 )) * 1000 } { cellpos(1) = 0 }
```

Figure 46 : The local transition rules

The convective transport has been arbitrarly been defined in the direction of increasing column values, so that in visual representations the carrier will be seen flowing from left to right. Being this the case, a local transition rule for the transport phase should set a cell's value to the current value of its (0,-1) neighbor cell. The rate at which this is done depends on the velocity of the flow at the cell, which, as mentioned before, has its maximum at the centre of the tube and decreases towards its walls. This is stated in the first transport rule in line 19. As mentioned in section 2, a local transition rule has three components, a value, a delay and a condition. For this rule, this components are:

Value: `{ (0,-1) }  //The value of the cell's left neighbor`

Delay:
```
{ 0.1 / ( 22.57878 * ( 1 - power( cellPos(0) * 0.001 + 0.0005 , 2)
/ 0.000625 )) * 1000 }
```

Condition: `{ cellpos(1) != 0 }`

The delay is calculated using equations 1 and 2. For a pump with a constant flow of 1,33ml/min, the average velocity is 11,29 cm/s. This value can be substituted in equation 2 and multiplied by 2 to yield the number 22.57878 shown in the delay expression. In addition, for equation 2 to be solved, we also need to know the distance to the center of the tube. CD++ provides a built in function called *cellpos* that returns a requested coordinate of the cell whose value is being sought. For a 2 dimensional model, *cellpos(0)* returns the cell's row. Consequently,

```
cellPos(0) * 0.001 + 0.0005
```

is the distance of the centre of the cell to the centre of the tube and therefore,

```
( 22.57878 * ( 1 - power( cellPos(0) * 0.001 + 0.0005 , 2) / 0.000625 ))
```

is the solution to equation 2, for $a = 0.025$ cm. Having the velocity of flow $v(r)$, the delay will be the time in milliseconds for a particle moving at speed $v(r)$ cm/s to travel across a 0.1 cm cell. This time is given by the expression

$$0.1 \,/\, v(r) * 1000$$

concluding our explanation for the delay component of the rule.

The generic rule we have just given is only valid for all cells that have a valid (0,-1) neighbor. The left border cells (those in column 0) do not satisfy this prerequisite, stated in the condition component `cellpos(1) != 0`, and should therefore have a different rule.

The rule in line 20 is the rule for the left border cells. It simply states that for these cells the new value should be 0.8, which corresponds to the concentration of the carrier solution being pumped into the tube.

Table 1 shows the results of applying equation 2 to calculate the delays for each row. It is important to notice that some adjacent rows have different delay values, as is the case of rows 2 and 3. This might lead to the presumption that the convective transport behavior will not be preserved due to an early preemption a cell's scheduled future value. This is not the case, as we will show.

Row	Delay (ms)
0	4
1	4
2	4
3	5
4	5
5	5
6	5
7	5
8	5
9	5
10	5
11	6
12	6

Row	Delay (ms)
13	6
14	7
15	7
16	8
17	9
18	10
19	11
20	14
21	17
22	23
23	38
24	112

Table 1 – Calculated delays for each row

When the simulation starts at time 0, all cells will evaluate their local transition functions and schedule their next change. A cell in row 2 will schedule an internal transition at time t = 4ms and a cell in row three at t = 5ms. So at time t = 4ms, all cells in row 2 will send an output event to their neighbors. Cells in row 3 will receive this event and evaluate the local transition function, which says they should take the value of their left neighbor. But their left neighbor has not changed yet, so the new value will be the same as the previous *future value*. Therefore, they will keep their scheduled internal transition for t = 5 ms. At this time, all cells in row 2 with a scheduled internal transition will send their new value to their neighbors. A row 2 cell receiving a new value from its left neighbor will again evaluate its local transition function, but this time the delay has already expired and there is no *future value* scheduled, so the result of this evaluation will be scheduled as the *future value* for time t = 10 ms.

Figure 47 shows the radial diffusion rules. For a cell with valid top and bottom neighbors, the diffusion rule states that the new cell value will be the average of the three cells. This is the case of the rule in line 22. A delay of 1 ms was chosen. Though a 0 ms delay would be more appropriate, this is still not supported in the version of NCD++ for which the model was written. A new version that implements the Parallel Cell-DEVS formalism has been recently finished, and is currently being tested. This version will allow 0 time delays. The other three rules in lines 23 and 24 cover the special case of top and border cells. These cells do not have both, a valid top and bottom neighbor so instead of using three cells to obtain the average, only two are used.

```
21 [diffusion]
22 rule : { ((-1,0) + (0,0) + (1,0)) / 3 } 1 { cellpos(0) != 0 AND cellpos(0) != 24 }
23 rule : { ((-1,0) + (0,0)) / 2 } 1 { cellpos(0) != 0 AND cellpos(0) = 24 }
24 rule : { ((0,0) +  (1,0)) / 2 } 1 { cellpos(0) = 0 AND cellpos(0) != 24 }
```

Figure 47 : Radial diffusion rules.

So far we have shown the diffusion rule, but we have not yet defined that this ruled should be evaluated when an external event is received through the *diffuse* input port. Figure 48 shows the statements that link the *fia* model *diffuse* input port to a cell's *diffuse* input port (line 27) and set the diffusion rule to be evaluated upon the arrival of an external event through this port (line 28).

```
[fia]
27 link : diffuse diffuse@fia(x,y)
28 PortInTransition : diffuse@fia(x,y) diffusion
```

Figure 48 : External coupling of the FIA Cell-DEVS model.

11.3 Simulation results

The described model was run for 10s and the state of the whole cell space was logged every 100ms. A graphical representation of the model at five different stages is shown in Figure 49. The logged results were also used to draw the conductivity curve.

To obtain the conductivity of the whole system, we divided the cell space in axial segments, calculated the resistance of each, and assumed the whole resistance to be the result of combining all segments in serial mode. We took each segment to be a column of cells and calculated its resistance using equation 4.

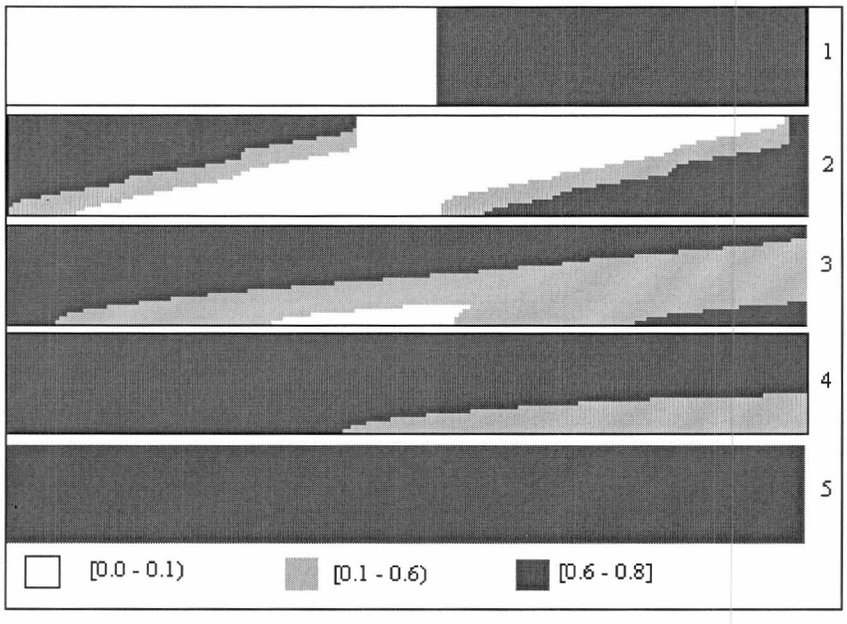


Figure 49: Different execution stages of the FIA model. (1) At time 0 the sample (white), has been injected. The other half of the tube contains the carrier solution (dark gray). (2,3,4) The convective transport makes the sample disperse faster at the middle of the tube than near the walls. (5) The whole tube now contains the carrier solution only.

$$R_{total} = \sum_{column=0}^{199} \left(\sum_{row=0}^{24} \frac{1}{R_{cell(row,col)}} \right)^{-1} \qquad \textbf{(Equation 4)}$$

To calculate the resistance, equation 5, which gives the conductivity of each cell, was used. The resistance of a cell can be obtained by calculating the inverse of the conductivity. All values are known, being the concentration of nitric acid the one that varies from cell to cell.

$$G_{cell} = \frac{1}{R_{cell}} = G_{HNO_3} + G_{H_2O} = \frac{Area_{cell}}{Length_{cell}} \left(\kappa_{HNO_3} \cdot [HNO_3] \right) \qquad \textbf{(Equation 5)}$$

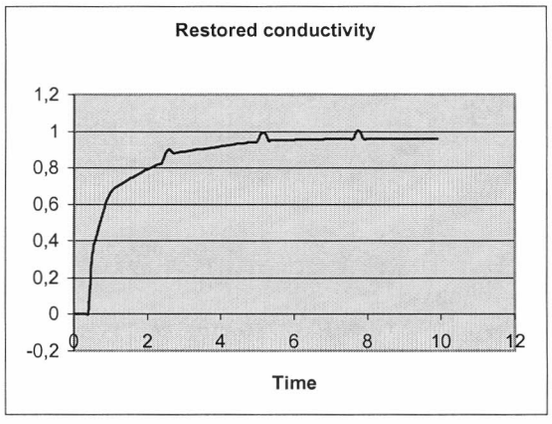


Figure 50 : Conductivity curve obtained

Figure 50 shows the conductivity curve obtained. For this example the curve is quite similar to the first part of the measured curve. It is a good starting point for the whole FIA manifold.

11.4 Performance analysis

The results shown for the FIA model were obtained running CD++ on a single machine. The simulation for 10 seconds of virtual time took more than 5 hours. Performance analysis, however, were conducted for executions for a virtual time of 2 seconds, using 1, 2, 5 and 10 machines.

In the FIA model, not all cells are active during the whole simulation. At the beginning, active cells concentrate near left end of the tube, and as the simulation progresses the activity shifts to the right, as shown in Figure 49. In addition, during the diffusion phase all cells are active.

The model was executed using two different set of partitions for 1, 2, 5 and 10 machines. Partition set I divides the tube in equal transversal sections and assigns each machine one section, as shown in Figure 51.

(a)	0	0	0	0	0	0	0	0	0	0
(b)	0	0	0	0	0	1	1	1	1	1
(c)	0	0	1	1	2	2	3	3	4	4
(d)	0	1	2	3	4	5	6	7	8	9

Figure 51: Partition I : the FIA tube is divided in 10 equal sections. (a) 1 machine (b) 2 machines (c) 5 machines (d) 10 machines

Partition set II divides the tube in 20 equal sections. Each machine is assigned a tube section from each end, as shows
Figure 53. Partition II attempts to make workload assignment even by combining sections that are active at different stages in one machine.

(a)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(b)	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
(c)	0	0	1	1	2	2	3	3	4	4	4	4	3	3	2	2	1	1	0	0
(d)	0	1	2	3	4	5	6	7	8	9	9	8	7	6	5	4	3	2	1	0

Figure 53: Partition II : the FIA tube is divided in 20 equal sections. (a) 1 machine (b) 2 machines (c) 5 machines (d) 10 machines

Figure 55 shows the execution times for the FIA model. As it can be seen, the FIA model does not experience a reduction in the execution time as did the heat diffusion. But it neither shows a big slowdown, except for the execution with 2 machines. This suggests the workload shifts from one machine to the other. A comparison between Partition I and II, shows this second one shows a better performance.

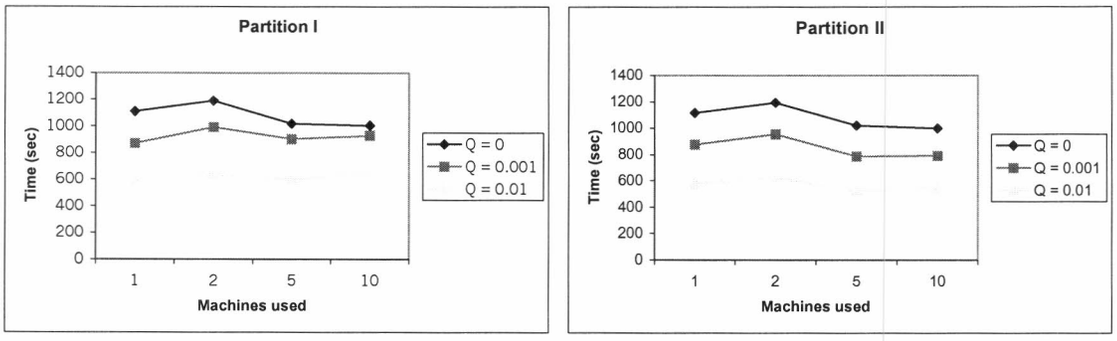


Figure 55 : Execution times for the FIA model.

12
Conclusions and further developments

A tool for the simulation of Parallel DEVS and Parallel Cell-DEVS models in distributed environments has been developed. The development process required:

- A definition of a new abstract simulator suitable for distributed environments.
- A selection of a suitable simulation middleware.
- An application design that would fit both, the middleware and the existing CD++ software.

A first abstract simulator specialized *coordinators* into *master* and *slaves* to reduce the number of inter-process messages required. The preliminary results showed, however, that the simulator did not perform well when too many machines were being used. This was due to a bottleneck caused by a centralized distribution of output messages. A revised simulator solved this problem by replacing this centralized mechanism with a distributed one.

For the middleware, the Warped kernels where chosen because they provide a common API for different kernels. The parallel simulator has been developed to support both the TimeWarp and NoTime kernels.

Performance analysis were carried on different models: an extended GPT, a heat diffusion model and flow injection model. These analysis showed that:

- The revised abstract simulator outperformed the original one.
- Performance of parallel simulation is model and partition dependant. In general terms, a model and partition combination that distribute the workload evenly among the available set of machines will perform well in distributed environments.

To asses how suitable a model-partition combination is for parallel execution, a parallelism metric was given. This metric helps to understand how the workload changes during the whole simulation.

There are quite a few topics for further improvement and research.

Firstly, improvements can be made to the Warped middleware. In particular, the NoTime kernel can be improved two pack multiple simulation messages together into a single batch and send them in one connection. This will avoid the significant overhead of setting up a connection for each message sent. Heavy load models send thousands of messages in a simulation cycle, so it may be presumed that message aggregation will improve performance significantly. In addition, the NoTime kernel can be further improved by eliminating the finalization detection mechanism which involves quite a few messages over the network. This changes, however, will change the NoTime kernel from a general purpose kernel to one specialized for the Parallel DEVS abstract simulator.

Further studies on different models should be carried out to asses how the workload changes in each case. This would allow to classify applications and help to construct efficient partitions. In addition, this studies will be a starting point for a dynamic load balance mechanism. Dynamic load balancing can improve performance. The parallelism metric can be improved to take into account network traffic.

Improvements can be also made to specification language of Cell-DEVS models, which has been left unchanged. The Parallel Cell-DEVS formalism allows multiple simultaneous events, which the current language can not handle properly.

13
References

[AIT98] F.J. Andrade, F.J.; Iñon, F.A.; Tudino, M.B.; Troccoli, O.E. "Integrated conductimetric detection: mass distribution in a dynamic sample zone inside a flow injection manifold", *Analytica Chimica Acta 19211, 1998.*

[Ame00] Ameghino, J.; Wainer, G. "Application of the Cell-DEVS paradigm using N-CD++." In *Proceedings of the 2000 Summer Computer Simulation Conference.* July 2000.

[Cho94a] Chow, A., and Zeigler, B. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism." In *Winter Simulation Conference Proceedings*. SCS, Orlando, Florida. 1994.

[Cho94b] Chow, A.; Kim, D.; Zeigler, B. "Abstract Simulator for the parallel DEVS formalism". *AI, Simulation, and Planning in High Autonomy Systems,* December 1994.

[Jef87] JEFFERSON, D. "Distributed simulation and the Time Warp Operating System". In 11th. Symposium on OS principles. pp 77-93. November 1987.

[Mar97] Martin, D.; McBrayer, T.; Radhakrishnan, R.; Wilsey, P. "Time Warp Parallel Discrete Event Simulator". Technical Report. Computer Architecture Design Laboratory, University of Cincinnati. 1997.

[Rao98] Rao, D.; Thondugulam V.; Radhakrishnan R.; Wilsey P. "Unsynchronized Parallel Discrete Event Simulation."In *Proceedings of the Winter Simulation Conference.* 1998.

[Rod99] Rodriguez, D.; Wainer, G. "New Extensions to the CD++ tool." In *Proceedings of SCS Summer Multiconference on Computer Simulation*, Chicago, USA. 1999.

[Wai98] Wainer, G.; Giambiasi, N. "Specification, modeling and simulation of timed Cell-DEVS spaces". Technical Report n.: 98-007. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.

[Wai00a] Wainer, G. "Improved cellular models with parallel Cell-DEVS". *Transactions of the SCS.* June 2000.

[Wai00b] Wainer, G.; Zeigler, B. "Experimental results of Timed Cell-DEVS quantization". In *Proceedings of AIS 2000*, Tucson, AZ. 2000.

[Wai01] Wainer, G.; Giambiasi, N. 2001. "Timed Cell-DEVS: modeling and simulation of cell spaces." in Discrete Event Modeling & Simulation: Enabling Future Technologies, Springer-Verlag

[Zei76] Zeigler, B. "Theory of modeling and simulation". Wiley, 1976. (T) (d)

[Zei90] Zeigler, B.. *Object Oriented Simulation with Hierarchical, Modular Models.* Academic Press, San Diego, California, 1990.

[Zei98a] Zeigler, B. DEVS Theory of Quantization. DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.

[Zei98b] Zeigler, B.; Cho, H. ; Lee, J. and Sarjoughian, H. The DEVS/HLA Distributed Simulation Environment and its Support for Predictive Filtering. DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.

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

CD++

A tool for Parallel DEVS and Parallel Cell-DEVS simulation

User's Guide

Alejandro Troccoli
Daniel A. Rodriguez, Amir Barylko, Jorge Beyoglonian
Gabriel A. Wainer

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

2001

Contents

CD++

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

1 Installation

CD++ was developed to run in UNIX environments. It has been successfully tested in clusters of Linux machines running on Pentium processors. It supports both, parallel and standalone simulation.

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

The CD++ distribution includes the following utilities:

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

1.1 System requirements

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

Compiling for UNIX / Linux

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

Compiling for Windows

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

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

1.2 MPI

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

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

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

1. Uncompress the distribution files
   gunzip -c mpich.tar.gz | tar xovf

2. Run
   ./configure

   This script will try to set the optimum parameters for compilation on your system. If mpich will be installed in a shared location, then run
   ./configure -prefix= /usr/local/mpich-1.2.0. (or your preferred location)

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

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

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

6. (Optional) Install mpich on a shared location

   make install

Troubleshooting

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

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

1.3 CD++

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

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

will just do this.

The following directory structure will be created

```
CD++
    +--------------    warped
                           +---------------- TimeWarp
                           +---------------- NoTime
                           +---------------- Sequential
                           +---------------- common

    +--------------    models
                           +---------------- net
                           +---------------- airport
```

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

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

```
#CD++ Makefile.common

#===============================================================================
#CD++ Directory Details
export MAINDIR=/home/atroccol/tesis/CD++


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

Figure 1: Makefile.common – Setting the source location

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

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

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

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

Figure 2: Makefile.common – Choosing the Warped kernel

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

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

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

#Uncomment these lines to include the net models
#DEFINES_CPP += -DDEVS_NET
#INCLUDES_CPP += -I./models/net
#LDFLAGS += -L./models/net
#LIBS += -lnet
#############################################################################################
```

Figure 3: Makefile.common – Model selection

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

```
make depend
make
```

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

```
make clean
make
```

2 Starting the simulator

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

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

2.1 Workstation Mode

To run CD++, type

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

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

Usage:

```
./cd++ [-ehlLmotdpPDvbfrsqw]
  e: events file (default: none)
  h: show this help
  l: logs all messages to a log file (default: /dev/null)
  L[I*@XYDS]: log modifiers (logs only the specified messages)
  m: model file (default : model.ma)
  o: output (default: /dev/null)
  t: stop time (default: Infinity)
  d: set tolerance used to compare real numbers
  p: print extra info when the parsing occurs (only for cells models)
  D: partition details file (default: /dev/null)
  P: parallel partition file (will run parallel simulation)
  v: evaluate debug mode (only for cells models)
  b: bypass the preprocessor (macros are ignored)
  f: flat debug mode (only for flat cells models)
  r: debug cell rules mode (only for cells models)
  s: show the virtual time when the simulation ends (on stderr)
  q: use quantum to compute cell values
  y: use dynamic quantum (strategy 1) to compute cells values
  Y: use dynamic quantum (strategy 2) to compute cells values
  w: sets the width and precision (with form xx-yy) to show numbers
```

Figure 4: CD++ command line options

The command line options allowed are:

–**efilename**: External events filename. If this parameter is omitted, the simulator will not use external events. The format for external event files is described in section 5.3.

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

The log file format is described in the section 6.2.

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

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

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

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

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

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

The format for a partition file is described in section 5.4.

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

The format for the generated output is described in section 6.1.

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

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

HH: hours

MM: minutes (0 to 59)

SS: seconds (0 to 59)

MS: thousandths of second (0 to 999)

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

–pfilename: Shows additional information when parsing a cell's local transition rules. The parameter must be accompanied with the name of the file that will be used to store the detail. This mode is useful when a syntax error occurs on complex rules.
The format used to store the output is showed in the section 6.4.

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

The format of the output generated when this mode is enabled is described in section 6.5.

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

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

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

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

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

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

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

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

3 Model definition

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

3.1 Structure of .ma file

A model file is used to define coupled and Cell-DEVS models. Atomic models are added to the tool at compile time, and if new atomic models need to be defined, they must be code as detailed in section 8. A model file consists of a set of groups and definition clauses within the groups. A group is identified by writing its name between square brackets. All lines following a group declaration are taken to be parameters for that group and are of the form

Id : value

As an example, mygroup is defined below:

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

Figure 5: Defining groups and group parameters

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

3.1.1 Coupled Models

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

Components:

components : model_name1[@atomicclass1] [model_name2[@atomicclass2] ...

Lists the component models that make the coupled model. If this clause is not specified, an error will occur. A coupled model might have atomic models or other coupled model as components. For atomic components, an instance name and a class name must be specified. This allows a coupled model to use more than one instance of an atomic class. For coupled models, only the model name must be given. This model name must be defined as another group in the same file.

Out:

out : portname1 portname2 ...

Enumerates the model's output ports. This clause is optional because a model may not have output ports.

In:

in : portname1 portname2 ...

Enumerates the input ports. This clause is also optional because a couple model is not required to have input ports.

Link :

link : *source_port*[*@model*] *destination_port*[*@model*]

Defines the links between the components and between the components and the coupled model itself. If name of the model is omitted it is assumed that the port belongs to the coupled model being defined.

A model definition is shown below.

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

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

Figure 6 : Example for the definition of a DEVS coupled model

3.1.2 Atomic models

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

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

Figure 7: User defined values for atomic models

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

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

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

[processor]
distribution : exponential
mean : 10

[processor2]
distribution : poisson
mean : 50

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

Figure 8: Example of setting parameters to DEVS atomic models

3.1.3 Cell DEVS models

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

These parameters are:

type : [CELL | FLAT]

Defines the abstract simulator to be used. If **cell** is specified, there will be one DEVS processor for each cell. Instead, if **flat** is specified, one flat coordinator will be used. CD++ currently supports the **cell** option only.

width : integer

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

height : integer

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

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

Defines the dimensions of the cellular space.
All the x_i values must be integers.
Dim can not be used together with any of the **width** and **height** parameters.

The vector that defines the dimension of the cellular model must have two or more elements. For an unidimensional cellular model, the following form should be used: $(x_0, 1)$.

When referencing a cell, all references must satisfy:

$$(y_0, y_1, ..., y_n) \quad 0 \le y_i < x_i \qquad \forall\, i = 0, .., n$$

with y_i an integer value

In : Defines the input ports for a cellular model.

Out : Defines the output ports the cellular model.

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

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

Valid link definitions are of the form:

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

Border : [WRAPPED | NOWRAPPED]

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

Delay : [TRANSPORT | INERTIAL]

Specifies the delay type used for all cells of the model. By default the value TRANSPORT is assumed.

DefaultDelayTime : integer

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

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

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

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

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

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

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

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

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

Precondition:

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

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

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

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

InitialCellsValue : *fileName*

Defines the filename for the file that contains a list of initial value for cells in the model. Section 5.1 defines the format for these files. **initialcellsvalue** can be used with any size of cellular models and will have more precedence that **initialrow** and **initialrowvalue.**

InitialMapValue : *fileName*

Defines the filename for the file that contains a map of values that will be used as the initial state for a cellular model. Section 5.2 defines the format for these files.

LocalTransition : *transitionFunctionName*

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

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

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

Section 9.3 illustrates the use of the **portInTransition** clause.

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

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

As an example,

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

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

4 Writing local transition functions for cellular models.

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

rule : result delay { condition }

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

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

4.1 A grammar for writing the rules

The BNF for the grammar used for the rules is shown in Figure 9. Words written in bold lowercase represent terminals symbols, while those written in uppercase represent non terminals.

```
RULELIST        = RULE
                | RULE RULELIST

RULE            = RESULT RESULT { BOOLEXP }

RESULT          = CONSTANT
                | { REALEXP }

BOOLEXP         = BOOL
                | ( BOOLEXP )
                | REALRELEXP
                | not BOOLEXP
                | BOOLEXP OP_BOOL BOOLEXP

OP_BOOL         = and | or | xor | imp | eqv

REALRELEXP      = REALEXP OP_REL REALEXP
                | COND_REAL_FUNC(REALEXP)

REALEXP         = IDREF
                | ( REALEXP )
                | REALEXP OPER REALEXP

IDREF           = CELLREF
                | CONSTANT
                | FUNCTION
                | portValue(PORTNAME)
                | send(PORTNAME, REALEXP)
                | cellPos(REALEXP)
```

```
CONSTANT        = INT
                  |  REAL
                  |  CONSTFUNC
                  |  ?

FUNCTION        = UNARY_FUNC(REALEXP)
                  |  WITHOUT_PARAM_FUNC
                  |  BINARY_FUNC(REALEXP, REALEXP)
                  |  if(BOOLEXP, REALEXP, REALEXP)
                  |  ifu(BOOLEXP, REALEXP, REALEXP, REALEXP)

CELLREF         = (INT, INT REST_TUPLA

REST_TUPLA      = , INT REST_TUPLA
                  |  )

BOOL            = t  |  f  |  ?

OP_REL          = !=  |  =  |  >  |  <  |  >=  |  <=

OPER            = +  |  -  |  *  |  /

INT             = [SIGN] DIGIT {DIGIT}

REAL            = INT  |  [SIGN] {DIGIT}.DIGIT {DIGIT}

SIGN            = +  |  -

DIGIT           = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

PORTNAME        = thisPort  |  STRING

STRING          = LETTER {LETTER}

LETTER          = a | b | c |...| z | A | B | C |...| Z

CONSTFUNC       = pi | e | inf | grav | accel | light | planck | avogadro |
                  faraday | rydberg | euler_gamma | bohr_radius | boltzmann
|
                  bohr_magneton | golden | catalan | amu | electron_charge |
                  ideal_gas | stefan_boltzmann | proton_mass | electron_mass
|
                  neutron_mass  | pem

WITHOUT_PARAM_FUNC = truecount | falsecount | undefcount | time | random |
                     randomSign

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

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

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

Figure 9: Grammar used for the definition of a cell's local transition

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

4.2 Precedence Order and Associativity of Operators

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

$$C + B * A$$

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

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

$$C \textbf{ and } B \textbf{ or } D$$

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

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

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

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

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

4.3 Functions and Constants allowed by the language

4.3.1 Boolean Values

Boolean values in CD++ use trivalent logic.

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

4.3.1.1 *Boolean Operators*

4.3.1.1.1 Operator AND

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

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

Figure 10: operator AND truthtable

4.3.1.1.2 Operator *OR*

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

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

Figure 11: Operator OR truthtable

4.3.1.1.3 Operator *NOT*

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

NOT	
T	F
F	T
?	?

Figure 12: Behavior of the boolean operator NOT

4.3.1.1.4 Operator *XOR*

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

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

Figure 13: Operator XOR truthtable

4.3.1.1.5 Operator *IMP*

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

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

Figure 14: Operator IMP truthtable

4.3.1.1.6 Operator *EQV*

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

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

Figure 15: Operator EQV truthtable

4.3.2 Functions and Operations on Real Numbers

4.3.2.1 *Relational Operators*

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

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

4.3.2.1.1 Operator =

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

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

Figure 16: Behavior of the Relational Operator =

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

4.3.2.1.2 Operator !=

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

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

Figure 17: Behavior of the Relational Operator !=

4.3.2.1.3 Operator >

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

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

Figure 18 : Behavior of the Relational Operator >

4.3.2.1.4 Operator <

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

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

Figure 19 : Behavior of the Relational Operator <

4.3.2.1.5 Operator <=

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

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

Figure 20 : Behavior of the Relational Operator <=

4.3.2.1.6 Operator >=

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

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

Figure 21: Behavior of the Relational Operator >=

4.3.2.2 Arithmetic Operators

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

The available operators are:

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

Figure 22: Arithmetic Operators

Division by zero will result to the undefined value.

4.3.2.3 Functions on Real Numbers

4.3.2.3.1 Functions to Verify Properties of Real Numbers

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

Function Even

Signature: **even** : *Real* → *Bool*

Description: Returns *True* if the value is integer and even. If the value is undefined returns *Undefined*. In any other case it returns *False*.

Examples:
even(?) = F
even(3.14) = F
even(3) = F
even(2) = T

Function Odd

Signature: **odd** : *Real* → *Bool*

Description: Returns *True* if the value is integer and odd. If the value is undefined returns Undefined. In any other case it returns *False*.

Examples:
odd(?) = F
odd(3.14) = F
odd(3) = T

odd(2) = F

Function isInt

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

Function isPrime

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

Function isUndefined

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

4.3.2.3.2 Mathematical Functions

This section describes commonly used mathematical functions.

4.3.2.3.2.1 Trigonometric Functions

Function tan

Signature: **tan** : *Real a* → *Real*
Description: Returns the tangent of *a* measured in radians.
For the values near to π/2 radians, returns the constant *INF*.
If *a* is undefined then return undefined.
Examples: tan(*PI* / 2) = *INF*
tan(?) = ?
tan(*PI*) = 0

Function sin

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

Function cos

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

Function sec

Signature: **sec** : *Real a* → *Real*
Description: Returns the secant of *a* measured in radians.

If *a* has the value**?** then returns**?**.
If the angle is of the form $\pi/2 + x.\pi$, with *x* an integer number, then returns the constant *INF*.

Function cotan

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

Function cosec

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

Function atan

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

Function asin

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

Function acos

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

Function asec

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

Function acotan

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

Function sinh

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

Function cosh

Signature: **cosh** : *Real a* → *Real*

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

Function tanh

Signature: **tanh** : *Real a* → *Real*

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

Function sech

Signature: **sech** : *Real a* → *Real*

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

Function cosech

Signature: **cosech** : *Real a* → *Real*

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

Function atanh

Signature: **atanh** : *Real a* → *Real*

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

Function asinh

Signature: **asinh** : *Real a* → *Real*

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

Function acosh

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

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

Function asech

Signature: **asech** : *Real a* → *Real*

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

Function acosech

Signature: **acosech** : *Real a* → *Real*

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

Function acotanh

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

Function hip

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

4.3.2.3.2.2 Functions to calculate Roots, Powers and Logarithms.

Function sqrt

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

Function exp

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

Function ln

Signature: **ln** : *Real a* → *Real*
Description: Returns the natural logarithm of *a*.
If *a* is undefined or is less or equal than zero, then returns **?**.
Examples: ln(–2) = ?
ln(0) = ?
ln(1) = 0
ln(?) = ?
Note: ln(*x*) is equivalent to **logn**(*x*, *e*) $\forall x$

Function log

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

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

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

Function logn

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

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

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

Function power

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

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

Function root

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

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

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

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

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

Function LCM

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

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

Function GCD

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

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

Function remainder

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

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

Examples:
remainder(12, 3) = 0
remainder(14, 3) = 2
remainder(4, 2) = 0
remainder(0, y) = 0 $\forall$ y $\neq$ **?**
remainder(x, 0) = x $\forall$ x
remainder(1.25, 0.3) = 0.05
remainder(1.25, 0.25) = 0
remainder(?, 3) = ?
remainder(5, ?) = ?

4.3.2.3.3 Functions to Convert Real Values to Integers Values

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

Function round

Signature: **round** : *Real a* → *Real*
Description: Rounds the value a to the nearest integer.
If a is undefined **?**, then returns **?**.
Examples:
round(4) = 4
round(?) = ?
round(4.1) = 4
round(4.7) = 5
round(–3.6) = –4

Function trunc

Signature: **trunc**: *Real x* → *Real*
Description: Returns the greater integer number less or equal than x.
If x is undefined, then returns **?**.
Examples:
trunc(4) = 4
trunc(?) = ?
trunc(4.1) = 4
trunc(4.7) = 4

Function truncUpper

Signature: **truncUpper**: *Real x* → *Real*
Description: Returns the smallest integer number greater or equal than x.
If x is undefined, then returns **?**.
Examples:
truncUpper(4) = 4
truncUpper(?) = ?
truncUpper(4.1) = 5
truncUpper(4.7) = 5

Function fractional

Signature: **fractional** : *Real a* → *Real*
Description: Returns the fractional part of a, including the sign.
If a is undefined then returns **?**.
Examples:
fractional(4.15) = 0.15
fractional(?) = ?

fractional(-3.6) = -0.6

4.3.2.3.4 Functions to manipulate the Sign of numerical values

Function abs

Signature: **abs** : *Real a* → *Real*
Description: Returns the absolute value of *a*.
If *a* is undefined then returns **?**.
Examples: abs(4.15) = 4.15
abs(?) = ?
abs(-3.6) = 3.6
abs(0) = 0

Function sign

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

Function randomSign

See the section 4.3.2.3.8.

4.3.2.3.5 Functions to manipulate Prime numbers

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

Function isPrime

See the section 4.3.2.3.1.

Function nextPrime

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

Function nth_Prime

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

4.3.2.3.6 Functions to calculate Minimum and Maximums

Function min

Signature: **min** : *Real a* x *Real b* → *Real*
Description: Return the minimum between *a* and *b*.
If *a* or *b* are undefined then returns **?**.

Function max

Signature: **max** : *Real a* x *Real b* → *Real*
Description: Returns the maximum between *a* and *b*.
If *a* or *b* are undefined then returns **?**.

4.3.2.3.7 Conditional Functions

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

Function if

Signature: **if** : Bool *c* x Real *t* x Real *f* → *Real*
Description: If the condition *c* is evaluated to *TRUE*, then returns the evaluation of *t*, else returns the evaluation of *f*.
The values of *t* and *f* can even come from the evaluation of any expression that returns a real value, including another *if* sentence.
Examples: If you wish to return the value 1.5 when the natural logarithm of the cell (0, 0) is zero or negative, or 2 in another case. In this case, it will be written:

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

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

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

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

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

Function ifu

Signature: **ifu** : Bool *c* x Real *t* x Real *f* x Real *u* → *Real*
Description: If the condition *c* is evaluated to *TRUE*, then returns the evaluation of *t*. If it evaluates to *FALSE*, returns the evaluation of *f*. Else (i.e. is undefined), returns the evaluation of *u*.
Examples: If you wish to return the value of the cell (0, 0) if its value is distinct than zero and undefined, 1 if the value of the cell is 0, and π if the cell has the undefined value. In this case, it will be invoked:

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

4.3.2.3.8 Probabilistic Functions

Function randomSign

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

Function random

Signature: **random** : → *Real*

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

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

Function chi

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

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

Function beta

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

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

Function exponential

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

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

Function f

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

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

Function gamma

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

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

Function normal

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

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

Function uniform

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

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

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

Function binomial

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

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

Function poisson

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

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

Function randInt

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

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

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

4.3.2.3.9 Functions to calculate Factorials and Combinatorial

Function fact

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

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

Examples: fact(3) = 6
fact(0) = 1
fact(5) = 120
fact(13) = 1.93205e+09
fact(43) = *INF*

Function comb

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

Description: Returns the combinatory $\left(\frac{a}{b}\right)$

If *a* or *b* are undefined, negatives or zero, or non–integers, then returns **?**.
This value is also returned if a < b.
If an overflow occur when calculating the next prime number, the constant *INF* is returned.

4.3.2.4 Functions for the Cells and his Neighborhood

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

Function stateCount

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

Function trueCount

Signature: **trueCount** : → *Real*
Description: Returns the quantity of neighbors of the cell whose state is 1.
This function is equivalent to *stateCount*(1) and it is not removed from the language to offer compatibility with *CD++*.

Function falseCount

Signature: **falseCount** : → *Real*
Description: Returns the quantity of neighbors of the cell whose state is 0.
This function is equivalent to *stateCount*(1) and it is not removed from the language to offer compatibility with *CD++*.

Function undefCount

Signature: **undefCount** : → *Real*
Description: Returns the quantity of neighbors of the cell whose state is undefined (?).
This function is equivalent to *stateCount*(1) and it is not removed from the language to offer compatibility with *CD++*.

Function cellPos

Signature: **cellPos** : *Real i* → *Real*
Description: Returns the i^{th} position inside the tupla that references to the cell. That is to say, given the cell $(x_0,x_1,...,x_n)$, then *cellPos*(i) = x_i.
If the value of *i* is not integer, then it will be automatically truncated.
If $i \notin [0, n+1)$, where n is the dimension of the model, it will produce an errorr that will abort the simulation.
The value returned always will be an integer.
Examples: Given the cell (4, 3, 10, 2):
cellPos(0) = 4
cellPos(3.99) = cellPos(3) = 2
cellPos(1.5) = cellPos(1) = 3
cellPos(–1) y cellPos(4) will generate an error.

4.3.2.5 Functions to Get the Simulation Time

Function Time

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

4.3.2.6 Functions to Convert Values between different units

4.3.2.6.1 Functions to Convert Degrees to Radians

Function radToDeg

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

Function degToRad

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

4.3.2.6.2 Functions to Convert Rectangular to Polar Coordinates

Function rectToPolar_r

Signature: **rectToPolar_r** : Real x x Real $y \rightarrow$ *Real*
Description: Converts the Cartesian coodinate (x, y) to the polar form (r, θ), and returns r.
If x or y are undefined then return **?**.

Function rectToPolar_angle

Signature: **rectToPolar_angle** : Real x x Real $y \rightarrow$ *Real*
Description: Converts the Cartesian coordinate (x, y) to the polar form (r, θ), and returns θ.
If x or y are undefined then return **?**.

Function polarToRect_x

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

Function polarToRect_y

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

4.3.2.6.3 Functions to Covert Temperatures between different units

Function CtoF

Signature: **CtoF** : Real $\rightarrow$ *Real*
Description: Converts a value expressed in Centigrade degrees to Fahrenheit degrees.
If the value is undefined then returns **?**.

Function CtoK

Signature: **CtoK** : Real $\rightarrow$ *Real*
Description: Converts a value expressed in Centigrade degrees to Kelvin degrees.
If the value is undefined then returns **?**.

Function KtoC

Signature: **KtoC** : Real $\rightarrow$ *Real*
Description: Converts a value expressed in Kelvin degrees to Centigrade degrees.

If the value is undefined then returns **?**.

Function KtoF

Signature: **KtoF** : Real → *Real*
Description: Converts a value expressed in Kelvin degrees to Fahrenheit degrees. If the value is undefined then returns **?**.

Function FtoC

Signature: **FtoC** : Real → *Real*
Description: Converts a value expressed in Fahrenheit degrees to Centigrade degrees. If the value is undefined then returns **?**.

Function FtoK

Signature: **FtoK** : Real → *Real*
Description: Converts a value expressed in Fahrenheit degrees to Kelvin degrees. If the value is undefined then returns **?**.

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

Function portValue

Signature: **portValue** : String *p* → *Real*
Description: Returns the last value arrived through the input port *p* of the cell of the cell being evaluated. This function will only be available for *PortInTransition* rules (see section 9.3) . Other uses will generate an error.

If no message has arrived through port *p* before *portValue* is evaluated, an undefined value (**?**) will be returned. Otherwise, the last value received through the port will be returned.

When the string "***thisPort***" is used as the port name, the value received through the port associated with the current *PortInTransition* will be returned. For example:

The following model has two different *PortInTransitions*

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

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

[functionB]
rule: 10     100    { portValue(portB) > 10 }
rule: 0      100    { t }
```

Figure 23 : Example of use of the function portValue

If we wanted to avoid repeating the same transition twice, we could either give the two ports the same name or use *thisPort* as shown next:

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

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

Figure 24 : Example of use of the function portValue with thisPort

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

Function send

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

Description: Sends the value x through the output port p.

If the output port p has not been defined, an error will be raised and the simulation will be aborted. This function is usually used to send values to other DEVS models.

send always returns 0. This makes it possible to include the function ***send*** in the *result* section of a rule without modifying the actual results.

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

Note: **Send** is a function of the language that can be used in any expression, as for example, in the definition of a *condition*. However, this is not recommended because for every condition that is evaluated that includes the function send, a value will be sent. Instead, send should be used in the expression for the *delay* or the *value* of the cell.

4.3.3 Predefined Constants

The following constants frequently used in the domains of the physics and the chemistry are available.

Constant Pi

Returns 3.14159265358979323846, which represent the value of π, the relation between the circumference and the radius of the circle.

Constant e

Returns 2.7182818284590452353, which represent the value of the base of the natural logarithms.

Constant INF

This constant represents to the infinite value, although in fact it returns the maximum value valid for a *Double* number (in processors Intel 80x86, this number is 1.79769×10^{308}).
Note that if, for example, we make $x + INF - INF$, where x is any real value, we will get 0 as a result, because the operator + is associative to left, for that will be solved:

$$(x + INF) - INF = INF - INF = 0.$$

Note: When being generated a numeric overflows taken place by any operation, it is returned *INF* or *–INF*. For example: power(12333333, 78134577) = *INF*.

Constant electron_mass
Returns the mass of an electron, which is 9.1093898 x 10^{-28} grams.

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

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

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

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

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

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

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

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

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

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

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

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

Constant Avogadro
Returns the Avogadro's number, defined as 6,0221367 x 10^{23} mols.

Constant amu
Returns the Atomic Mass Unit, defined as 1,6605402 x 10^{-27} kg.

Constant pem
Returns the ratio between the proton and electron mass, defined as 1836,152701.

Constant ideal_gas
Returns the constant of the ideal gas, defined as 22,41410 litres / mols.

Constant Faraday
Returns the Faraday's constant, defined as 96485,309 coulomb / mol.

Constant Stefan_boltzmann
Returns the Stefan-Boltzmann's constant, defined as 5,67051 x 10^{-8} Watt / (m^2 . $°K^4$)

Constant golden
Returns the *Golden Ratio*, defined as $\frac{1+\sqrt{5}}{2}$.

Constant euler_gamma
Returns the value of the Euler's Gamma, defined as 0.5772156649015.

4.4 Techniques to Avoid the Repetition of Rules

This section describes different techniques that allow to avoid repeating rules. This helps to make models more readable.

4.4.1 Clause *Else*

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

Figure 25 shows a short example where the *Else* clause is used. The default local transition for the cells in this model is *default_rule.* In addition, cell (13,13) defines a special funcion to be used when an external event arrives through port *In*. If none of the conditions for the rules that make this functions is satisfied, then the else clause sets *default_rule* as the function to be evaluated.

```
[demoModel]
type: cell
...
link: in  in@demoModel(13,13)
localTransition: default_rule
portInTransition: in@demoModel(13,13)    another_rule

[default_rule]
rule: ...
...
rule: ...

[another_rule]
rule: 1 1000 { portValue(thisPort) = 0 }
...
else: default_rule
```

Figure 25 : Example of the Else clause

The *Else* clause can point to any valid transition function. Care must be taken to avoid circular references, as in the example shown next.

```
[another_rule1]
rule: 1    1000 { portValue(thisPort) = 0 }
rule: 1.5  1000 { (0,0) = 5 }
rule: 3    1500 { (1,1) + (0,0) >= 1 }
else: another_rule2

[another_rule2]
rule: 1 1000 { (0,0) + portValue(thisPort) > 3 }
else: another_rule1
```

Figure 26 : A circular reference produced by a bad use of the clause Else

CD++ will detect the special case shown in Figure 27, where the *else* clause references the same function being defined.

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

Figure 27 : Example of a circular reference detected by the simulator

4.4.2 Preprocessor – Using Macros

CD++ has a preprocessor that will expand macros. If macros are not used, the preprocessor can be disabled using the command line argument **–b** to speed up model parsing.

Macros are usually defined in separate files that are included in the main .ma file be means of the preprocessor **#include** directive, which is of the form

#include(fileName)

where *fileName* is the name of the file that contains the definition of the macros. This file should be in the same directory where the main .ma file is.

More than one #include directive is allowed in the main .ma file, but no included files can have themselves the #include directive.

To define a macro, the directives **#BeginMacro** and **#EndMacro** are used.

A macro definition has the form:

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

Figure 28 : Definition of a macro

Macros can contain any valid text in any number of lines. The only restriction that applies is that they can not be used in the same file they are defined.

To expand a macro, the **#Macro** directive should be used in the place were the macro shoudl be expanded. A #macro directive is of the form

#Macro(*macroName*)

An included file can contain any number of macro definitions. Any text in these files that is outside the macro definitions is ignored. If a required macro is not found, an error will be reported.

An *#include* directive can be placed at any line of the .ma file, as long as the macros therein defined are used after the *#include.*

A macro can not make use of another macro.

Within a .ma file, the preprocessor allows comments. Comments begin with a **%** . All text between the **%** and the end of the line is ignored.

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

Figure 29 : A .ma file with comments

Section 9.5 shows a model where macros are used.

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

5 Supporting files

5.1 Defining initial cell values using a .val file

Within the definition of a cellular model, the *InitialCellValue* parameter defines a file name with the initial values for the cells. This is a plain text file. Each line of the file defines a value for a different cell. The format of this file is shown in Figure 30.

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

Figure 30 : Format of the file used to define the initial values of a cellular model

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

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

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

It is not necessary to define an initial value for each cell. If no value is defined in this file, then the value defined by the parameter *InitialValue* will be used.

Figure 31 shows a short fragment of a .val file for a cellular space of 4 dimensions.

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

Figure 31 : Example of a file for the definition of the initial values for a Cellular Model

5.2 Defining initial cell values using a .map file

If the *InitialMapValue* parameter is used, then the initial values for a cellular model are specified in a .map file. This file contains a map of cell values, as shown in Figure 32.

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

Figure 32 : .map file format

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

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

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

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

5.3 External events file

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

HH:MM:SS:MS PORT VALUE

where:

HH:MM:SS:MS	is the time when the event will occur.
Port	is the name of the port from which the event will arrive.
Value	is the numerical value for the event. Can be a real number or the undefined value (**?**).

Example:

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

Figure 33 : File with external events

5.4 Partition file

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

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

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

A line starts with a machine number (machine numbers start at 0) followed by a space, a colon and a list of names separated by spaces. Different lines may start with the same machine number.

The list of names following a machine number is the list of atomic instances that will be hosted by that machine. For cellular models, a single cell may be specified or a range of cells may be given. A cell range is described with name of the coupled cell model followed by the first cell in the range, two dots, and the last cell in the range.

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

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

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

Figure 34 : Partial definition of the heat diffusion model

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

```
0 : generadorCalor generadorFrio
0 : superficie(0,0)..(32,32)
1 : superficie(0,33)..(32,65)
2 : superficie(0,66)..(32,99)
3 : superficie(33,0)..(65,32)
4 : superficie(33,33)..(65,65)
5 : superficie(33,66)..(65,99)
6 : superficie(66,0)..(99,32)
7 : superficie(66,33)..(99,65)
8 : superficie(66,66)..(99,99)
```

Figure 35 : Valid partition for the heat diffusion model over 9 machines

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

6 Output Files

6.1 Output events

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

HH:MM:SS:MS PORT VALUE

Following is a small example of an output file.

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

Figure 36 : Example of an Output file

6.2 Format of the Log File

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

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

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

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

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

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

[superficie]
type : cell
width : 5
height : 5
...
```

Figure 37 : Partial definition of the heat diffusion model

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

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

Figure 38 : Calor.log

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

```
0 I / 00:00:00:000 / top(29) para superficie(01)
0 D / 00:00:00:000 / superficie(0,0)(02) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,1)(03) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,2)(04) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,3)(05) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(0,4)(06) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,0)(07) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,1)(08) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,2)(09) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,3)(10) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(1,4)(11) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,0)(12) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,1)(13) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,2)(14) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,3)(15) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(2,4)(16) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,0)(17) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,1)(18) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,2)(19) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,3)(20) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(3,4)(21) / 00:00:00:000 para superficie(01)
0 D / 00:00:00:000 / superficie(4,0)(22) / 00:00:00:000 para superficie(01)
```

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

Figure 39 : Fragment of calor.log01

6.3 Partition Debug Info

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

Figure 41 shows a fragment of a partition debug file generated when running the model described in Figure 37 with the partition shown next.

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

Figure 40 : Partition for the heat diffusion model of Figure 37

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

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

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

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

...

Model: superficie(3,0)
```

```
  Machines:
         Machine: 1   ProcId: 18 < local >  < master >

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

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


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

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

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

...

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

Figure 41 : Partition debug information file calor.pardeb01 (LP 1)

6.4 Output generated by the Parser Debug Mode

When the simulator is invoked with the option **–p**, the debug mode for the parser is activated. In debug mode, the parser will write the parse tree as it reads the rules. All tokens that are successfully

processed are shown and if there is a syntax error, the place were the error was detected is specified.

Figure 42 shows the output generated for the *Game Life* model as implemented in section 9.1.

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

Figure 42 : Output generated in the Parser Debug Mode for the Game of Life

6.5 Rule evaluation debugging

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

Figure 43 shows a fragment of the output generated for the Game of the Life model of section 9.1 Line numbers have been added to make the following explanations clear.

The first two lines indicate the beginning of a new evaluation. Line 2 begins the evaluation of the first rule for the first cell. Each evaluated argument is listed with the partial result for the expression. Line 2 shows the evaluation of the cell reference (0,0), which turned out to be 0. In line 3, the integer constant 1 is evaluated, which is later compared to 0, evaluating to 0 (false). *BinaryOp* indicates that a binary operation is being performed. The operator name will be included between brackets, as well as the value of each of the operands. Line 13 shows the final result for the condition of the rule, which was false in this case.

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

Figure 43 : Fragment of the output generated by the debug mode for the Evaluation or Rules

7 Utility programs

7.1 Drawlog

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

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

```
drawlog -[?hmtclwp0]

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

Figure 44 : Help shown by DrawLog

–?: similar to **–h**.

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

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

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

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

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

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

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

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

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

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

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

Figure 45 shows two different ways of starting drawlog. The first uses a log file as input. The second one, instead, takes its input from the standard input.

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

or

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

Figure 45 : Examples for the invocation to DrawLog

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

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

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

7.1.1 Bidimensional cellular models

A 2 dimensions model will be displayed as a matrix of values. Figure 46 shows a fragment of the output generated by DrawLog for a two-dimensional model of size (10, 10). The number width has been set to 5 and the precision to 1.

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

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

Figure 46 : Fragment of the output generated for a bidimensional cellular model

7.1.2 Three dimensional models

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

Figure 47 shows the output of *Drawlog* when used to draw a cellular space of size (5,5,4) with a number width of 1, a precision of 0 and zero values not displayed.

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

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

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

Figure 47 : Fragment of the output generated for a three–dimensional cellular model

7.1.3 Cellular models of more than 3 dimensions

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

Figure 48 shows a fragment of the output generated by *DrawLog* for a model of size (2, 10, 3, 4).

```
Line : 506 - Time: 00:00:00:000
  (0,0,0,0) = ?
  (0,0,0,1) = 0
  (0,0,0,2) = 9
  (0,0,0,3) = 0
  (0,0,1,0) = 21
  ...    ...    ...
  ...    ...    ...
  (1,9,1,0) = 0
  (1,9,1,1) = 4.333
  (1,9,1,2) = 0
  (1,9,1,3) = -2
  (1,9,2,0) = 6
  (1,9,2,1) = 0
  (1,9,2,2) = 7
  (1,9,2,3) = 0


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

Figure 48 : Fragment of the output generated for a model with dimension 4

7.2 Parlog

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

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

Usage:

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

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

Figure 49 : Parlog command line options

–h : Displays help.

-? : Displays help.

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

Figure 50 shows the output generated by parlog with a model running in for machines.

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

Figure 50 : Parlog output for a 4 machines partition.

7.3 Logbuffer

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

To run logbuffer use,

logbuffer [-b]

-bn Sets the size of the buffer. The default size is 200.

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

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

Figure 51 : Logbuffer receives a message with timestamp 3 and then two messages with timestamp 2. Logbuffer sorts and sent in the correct order.

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

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

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

Figure 52 : Running pcd with logbuffer.

7.4 Random Initial States – MakeRand

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

Usage:

```
makerand -[?hmcs]

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

Figure 53 : MakeRand command line options

–**?**: similar to –**h**.

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

–**c**: Name of the cellular model. This parameter is required because the size of the model needs to be known.

–**s**: Specifies the type of initial state to be created:

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

–**s1–n**: Indicates that the model initially will have *n* cells with value 1 (distributed randomly according to an uniform distribution) and the rest of the cells will have the value 0. If *n* is bigger to the quantity of cells of the model, then an error will occur and the initial state will not be generated.
For example, if we have a 40x40 cellular and we want 75% of the cells (1200 cells) to have an initial value of 1, and the remaining cells an initial value of 0, then **–s1–1200** should be used.

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

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

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

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

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

Usage:

```
toMap -[?hmci]

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

Figure 54 : Command line arguments for toMap

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

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

–**c**: Name of the cellular model.

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

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

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

8 Coding new atomic models

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

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

8.1 Defining the state of a model

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

The AtomicState class declaration is shown below.

```
class AtomicState : public ModelState {
public:

      enum State
      {
            active,
            passive
      } ;


      State st;

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

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

};
```

Figure 55: The AtomicState class.

To access the current state the function

ModelState* getCurrentState()

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

An assignment operator and a copy constructor need to be provided for Warped to work properly. In addition, the method getSize should be overridden to return the size of the class.

8.2 Defining a new atomic model

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

```
class Atomic : public Model
{
public:
  virtual ~Atomic();     // Destructor


protected:

  //User defined functions.
  virtual Model &initFunction() = 0;
  virtual Model &externalFunction ( const MessageBag & );
  virtual Model &externalFunction( const ExternalMessage & );
  virtual Model &internalFunction( const InternalMessage & ) = 0 ;
  virtual Model &outputFunction( const CollectMessage & ) = 0 ;
  virtual Model &confluentFunction ( const InternalMessage &, const MessageBag & );
  virtual ModelState* allocateState();
  virtual string className() const

  //Kernel services
  void nextChange(Vtime);
  Vtime nextChange();
  void lastChange(Vtime);
  Vtime lastChange();

  Model &holdIn( const AtomicState::State &, const VTime & ) ;
  Model &sendOutput(const VTime &time, const Port & port , BasicMsgValue *value)
  Model &sendOutput(const VTime &time, const Port & port , Value value)
  Model &passivate();



  //State functions

  virtual ModelState* getCurrentState() const;
  virtual ModelState* getCurrentState() ;

  //State shortcuts
  Model &state( const AtomicState::State &s )
  { ((AtomicState *)getCurrentState())->st = s; return *this; }

  const AtomicState::State &state() const
  {return ((AtomicState *)getCurrentState())->st;}

};       // class Atomic
```

Figure 56: The Atomic Class

The class atomic provides a set of services and requires a set of functions to be redefined. The services are functions that allow the model to tell the simulator the current state and duration. These are:

- **holdIn(state, VTime)**

  Tells the simulator the model will remain in the state *state* for a period of V*Time time.* It corresponds to the ta(s) function of the DEVS formalism.

- **passivate()**

  Sets the next internal transition time to infinity. The model will only be activated again if an external event is received.

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

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

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

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

- **nextChange():**

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

- **lastChange():**

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

- **state():**

  Returns the current model's phase.

- **getParameter(modelName, parameterName)**

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

The new class should override the following functions:

- **virtual Model &initFunction()**

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

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

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

The interface for the MessageBag class is shown below.

```
class MessageBag {

public:

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

        MessageBag &add( const BasicPortMessage* );

        bool portHasMsgs( const string& portName ) const;

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

        int size() const

        MessageBag& eraseAll();

        const VTime& time() const;

};
```

Figure 57: MessageBag class

- **virtual Model &internalFunction(const InternalMessage &)**

  This method corresponds to the δ_{int} function of the DEVS formalism.

- **virtual Model &outputFunction(const CollectMessage &)**

  This function is called before δ_{int}. It should send all the output event. Each output event is sent using the function sendOutput defined below.

- **virtual Model &confluentFunction (const InternalMessage &, const MessageBag &)**

  It corresponds to the δ_{conf} function of the DEVS formalism. By default, it is set to:

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

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

  //Call the external function
  externalFunction( extMsgs );

  return *this;

}
```

- **virtual string className()**

Returns the name of the atomic class.

8.3 Defining the output values

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

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

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

  BasicMsgValue(const BasicMsgValue& );

};

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

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

Figure 58: The BasicMsgValue and RealMsgValue classes

The user needs to define the following functions:

- **virtual int valueSize() const;**

  Returns the size of the class. It should be set to:

  ```
  return sizeof( className);
  ```

- **virtual string asString();**

  Returns a string that is used in the log file to log the value sent or received.

- **virtual BasicMsgValue * clone();**

  Returns a pointer to a new copy of the message value. The function that receives the pointer will own it and afterwards delete it.

- **BasicMsgValue(const BasicMsgValue&)**

  A copy constructor is required.

8.4 Example. A queue model.

A queue is a device of temporary storage that uses a FIFO (First In First Out) mechanism. Our model of a queue will hold any type of user defined value. The queue will have three input ports and one output port. Values to be stored will be received through the input port *In* and will later be sent through the port *Out*. The input ports *start-stop* and *next* will serve to regulate the flow of values through the output port. Figure 59 shows the structure of our model of a queue.

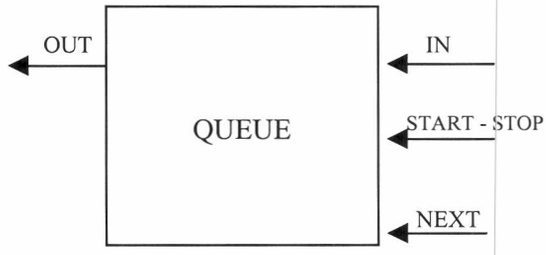


Figure 59: Structure of a Queue

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

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

After this brief description, we are ready to begin writing our model. First, we need to define a class to store the state of the queue. The queue will have two state variables: a list of elements and a boolean to store the enabled/disabled status. Figure 60 lists the Queue state class declaration and definition.

Once the state class has been defined, we are ready to implement the model itself. The Queue class declaration is shown in Figure 61.

```
class QueueState : public AtomicState {

public:

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

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

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

        ElementList::const_iterator cursor;

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

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

        return *this;
        }

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

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

Figure 60 : QueueState class

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

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

  ModelState* allocateState()
  {      return new QueueState;}

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

  VTime preparationTime;

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

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

  void enabled (bool val)
  {  ((QueueState*)getCurrentState())->enabled = val; }
};       // class Queue
```

Figure 61: The Queue class declaration

The initFunction has to set the initial state for the queue, as shown in Figure 62. The elements of the list will be erased and the enabled will be set to true.

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

Figure 62: initFunction for the Queue model

The externaFunction will be activated every time one or more events are received. For the queue model, this function will have to insert into the queue all values received through port *In*, schedule an output if a value is received through the port *next* and enabled or disable the queue if an event is received through port *start-stop*, as detailed in Figure 63. It is important to notice that it is the modeler's responsibility to set which message will have the highest priority when more than one is received. For our queue model, it can be seen from Figure 63 that the *start-stop* messages will have higher precedence than the *done* and *in* messages.

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

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

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

        if ( portHasMsgs( "in" )
        {

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

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

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


        }

}
```

Figure 63: External transition function for the queue model

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

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

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

Figure 64: Methods for the Output Function and the Internal Transition of the Queue

The sendOutput function will delete the pointer it receives, so all memory previously allocated to store the queue values will be reclaimed.

If we wanted to use the queue for a network model, the queue would store IP packets. Then an IP packet class derived from BasicMsgValue should be defined.

Figure 65 lists the definition of the IPPacket class. The only restriction that needs to be placed in classes derived from BasicMsgValue is that they do not contain any pointers.

```
class IPPacket : public BasicMsgValue
{
public:

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

  IPPacket();
  virtual ~IPPacket();

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

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

  IPPacket(const IPPacket& );

};
```

Figure 65: IP Packet Definition

9 Appendix A – Examples

9.1 Game of Life

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

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

The implementation of this model in *CD++* is as follows:

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1,-1) life(1,0) life(1,1)
initialvalue : 0
initialrowvalue :  1  00010001111000000000
initialrowvalue :  2  00110111100010111100
initialrowvalue :  3  00110000011110000010
initialrowvalue :  4  00101111000111100011
initialrowvalue : 10  01111000111100011110
initialrowvalue : 11  00010001111000000000
localtransition : life-rule

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

Figure 66 : Implementation of the Game of Life

9.2 A bouncing object

The following is the specification of a model that represents an object in movement that bounces against the borders of a room. This example is ideal to illustrate the use of a non toroidal cellular automata, where the cells of the border have different behavior to the rest of the cells.

For the representation of the problem, 5 different values are used for the states of each cell, these values are:

0 = represents an empty cell.
1 = represents the object moving toward the south east.
2 = represents the object moving toward the north east.
3 = represents the object moving toward the south west.
4 = represents the object moving toward the north west.

The specification of the model is:

```
[top]
components : rebound

[rebound]
type : cell
width : 20
height : 15
delay : transport
defaultDelayTime  : 100
border : nowrapped
neighbors : rebound(-1,-1)              rebound(-1,1)
neighbors :                rebound(0,0)
neighbors : rebound(1,-1)               rebound(1,1)
initialvalue : 0
initialrowvalue : 13     0000000000000000010
localtransition : move-rule
zone : cornerUL-rule { (0,0) }
zone : cornerUR-rule { (0,19) }
zone : cornerDL-rule { (14,0) }
zone : cornerDR-rule { (14,19) }
zone : top-rule { (0,1)..(0,18) }
zone : bottom-rule { (14,1)..(14,18) }
zone : left-rule { (1,0)..(13,0) }
zone : right-rule { (1,19)..(13,19) }

[move-rule]
rule : 1 100 { (-1,-1) = 1 }
rule : 2 100 { (1,-1) = 2 }
rule : 3 100 { (-1,1) = 3 }
rule : 4 100 { (1,1) = 4 }
rule : 0 100 { t }

[top-rule]
rule : 3 100 { (1,1) = 4 }
rule : 1 100 { (1,-1) = 2 }
rule : 0 100 { t }

[bottom-rule]
rule : 4 100 { (-1,1) = 3 }
rule : 2 100 { (-1,-1) = 1 }
rule : 0 100 { t }

[left-rule]
rule : 1 100 { (-1,1) = 3 }
rule : 2 100 { (1,1) = 4 }
rule : 0 100 { t }

[right-rule]
rule : 3 100 { (-1,-1) = 1 }
rule : 4 100 { (1,-1) = 2 }
rule : 0 100 { t }

[cornerUL-rule]
rule : 1 100 { (1,1) = 4 }
```

```
rule : 0 100 { t }

[cornerUR-rule]
rule : 3 100 { (1,-1) = 2 }
rule : 0 100 { t }

[cornerDL-rule]
rule : 2 100 { (-1,1) = 3 }
rule : 0 100 { t }

[cornerUR-rule]
rule : 4 100 { (-1,-1) = 1 }
rule : 0 100 { t }
```

Figure 67: Implementation of the Rebound of an Object

9.3 Classification of raw materials

The aim of this example is to show the use of special behavior that can be given to a cell when an external event arrives through an input port. We have a model that represents the packing and classification of certain raw material that contains 30% of carbon approximately. The model is made of a machine that loads 100 grams fractions of that substance in a carrying band. One a fraction reaches the end of the band, it is processed by a packager that takes these fractions until a kilogram is obtained. Then, the packed substance is classified. If each packet contains 30 ± 1 % of carbon, it is classified as of first quality; otherwise, it will be of second quality.

The model uses the atomic model *Generator* that generates values (in this case always the value 1) each *x* seconds (where *x* has and Exponential distribution with average 3). These values are passed to the carry band, represented by a cellular mode. At the end of the band, another cellular model makes the packaging and selection.

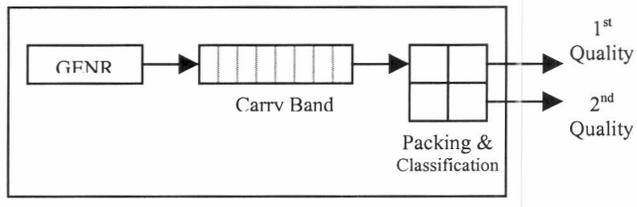


Figure 68: Coupling structure for the Classification of Substances

The following is the specification of the model:

```
[top]
components : genSubstances@Generator queue packing
out : outFirstQuality outSecondQuality
link : out@genSunstances in@queue
link : out@queue in@packing
link : out1@packing  outFirstQuality
link : out2@packing  outSecondQuality
```

```
[genSubstances]
distribution : exponential
mean : 3
initial : 1
increment : 0

[queue]
type : cell
width : 6
height : 1
delay : transport
defaultDelayTime  : 1
border : nowrapped
neighbors : queue(0,-1) queue(0,0) queue(0,1)
initialvalue : 0
in : in
out : out
link : in in@queue(0,0)
link : out@queue(0,5) out
localtransition : queue-rule
portInTransition : in@queue(0,0)  setSubstance

[queue-rule]
rule : 0           1 { (0,0) != 0 and (0,1) = 0 }
rule : { (0,-1) }  1 { (0,0) = 0 and (0,-1) != 0 and not isUndefined((0,-
1)) }
rule : 0        3000 { (0,0) != 0 and isUndefined((0,1)) }
rule : { (0,0) }   1 { t }

[setSubstance]
rule : { 30 + normal(0,2) } 1000 { t }

[packing]
type : cell
width : 2
height : 2
delay : transport
defaultDelayTime  : 1000
border : nowrapped
neighbors : packing(-1,-1) packing(-1,0) packing(-1,1)
neighbors : packing(0,-1)  packing(0,0)  packing(0,1)
neighbors : packing(1,-1)  packing(1,0)  packing(1,1)
in : in
out : out1 out2
initialvalue : 0
initialrowvalue :  0      00
initialrowvalue :  1      00
link : in in@ packing(0,0)
link : in in@ packing(1,0)
link : out@ packing(0,1) out1
link : out@ packing(1,1) out2
localtransition : packing-rule
portInTransition : in@packing(0,0) add-rule
portInTransition : in@packing(1,0) incQuantity-rule

[packing-rule]
rule : 0   1000 { isUndefined((1,0)) and isUndefined((0,-1)) and (0,0) = 10
}
rule : 0   1000 { isUndefined((-1,0)) and isUndefined((0,-1)) and (1,0) =
10 }
rule : { (0,-1) / (1,-1) }  1000 { isUndefined((-1,0)) and
isUndefined((0,1))
                        and (1,-1) = 10 and abs( (0,-1) / (1,-1) - 30 ) <=
1 }
```

```
rule : { (-1,-1) / (0,-1) } 1000 { isUndefined((1,0)) and
isUndefined((0,1))
                        and (0,-1) = 10 and abs( (-1,-1) / (0,-1) - 30 ) >
1 }
rule : { (0,0) } 1000 { t }

[add-rule]
rule : { portValue(thisPort) + (0,0) } 1000 { portValue(thisPort) != 0 }
rule : { (0,0) }                        1000 { t }

[incQuantity-rule]
rule : { 1 + (0,0) } 1000 { portValue(thisPort) != 0 }
rule : { (0,0) }      1000 { t }
```

Figure 69: Implementation of the Model to Classify Substances

The cellular model **queue** that represents the carry band makes use of the **portInTranstition** clause. As it was mentioned earlier, this clause is used to set the rule that will be evaluated when an external event is received by the cell through the specified port. This clause is then used again in the definition of the model *Packing* set the behavior of the cells upon the reception of a raw material from the carry band.

9.4 Game of Life – 3D

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

Figure 70 shows the model definition and Figure 71 lists the contents of file "3d-life.val" that contains the initial values for the cell.

```
[top]
components : 3d-life

[3d-life]
type : cell
dim : (7,7,3)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors : 3d-life(-1,-1,-1) 3d-life(-1,0,-1) 3d-life(-1,1,-1)
neighbors : 3d-life(0,-1,-1)  3d-life(0,0,-1)  3d-life(0,1,-1)
neighbors : 3d-life(1,-1,-1)  3d-life(1,0,-1)  3d-life(1,1,-1)
neighbors : 3d-life(-1,-1,0)  3d-life(-1,0,0)  3d-life(-1,1,0)
neighbors : 3d-life(0,-1,0)   3d-life(0,0,0)   3d-life(0,1,0)
neighbors : 3d-life(1,-1,0)   3d-life(1,0,0)   3d-life(1,1,0)
neighbors : 3d-life(-1,-1,1)  3d-life(-1,0,1)  3d-life(-1,1,1)
neighbors : 3d-life(0,-1,1)   3d-life(0,0,1)   3d-life(0,1,1)
neighbors : 3d-life(1,-1,1)   3d-life(1,0,1)   3d-life(1,1,1)
initialvalue : 0
initialCellsValue : 3d-life.val
localtransition : 3d-life-rule

[3d-life-rule]
rule : 1 100 { (0,0,0) = 1 and (truecount = 8 or truecount = 10) }
rule : 1 100 { (0,0,0) = 0 and truecount >= 10 }
rule : 0 100 { t }
```

Figure 70: Implementation of the Game of Life – 3D

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

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

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

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

9.5 Use of Macros

The following example shows how macros can be used to write a new version of the *Game of the Life* for a 4 dimensional space. Macros can be defined in external files that are included in the main .ma file. More than one macro definition is may be included per file, but no macro can make use of an existing macro. A macro is defined between the *#BeginMacro* and a *#EndMacro* directives. All other text is ignored. The next figures show the contents of the four files that are used to completely define the new model.

```
#include(life.inc)
#include(life-1.inc)

[top]
components : life

[life]
type : cell
dim : (2,10,3,4)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors :                 life(-1,-1,0,0) life(-1,0,0,0) life(-1,1,0,0)
neighbors : life(0,-8,0,0) life(0,-1,0,0)  life(0,0,0,0)  life(0,1,0,0)
neighbors :                 life(1,-1,0,0)  life(1,0,0,0)  life(1,1,0,0)
initialvalue : 0
initialCellsValue :  life.val
localtransition : life-rule

[life-rule]
% Comment: Here starts the definition of rules
rule : 1                100 { #macro(Heat) or #macro(Rain) }
rule : 0                100 { (0,0,0,0) = ? OR (0,0,0,0) = 2 }
#macro(rule1)   % Another comment: A macro is invoked
rule : 1                100 { (0,0,0,0) = (1,0,0,0) AND (0,0,0,0) > 1 }
#macro(rule2)
```

Figure 72: Implementation of the Game of Life with 4 dimensions and using macros

```
(0,0,0,0) = ?
(1,0,0,0) = 25
(0,0,1,0) = 21
(0,1,2,2) = 28
(1, 4, 1,2) = 17
(1, 3, 2,1) = 15.44
```

Figure 73: File life.val that contains the initial values for the Game of Life in 4D

```
This is a comment: The macro Rule3 assigns the value 0 if the cell's value
is 3, and 4 if the cell's value is negative.

#BeginMacro(rule3)
rule : 0 100 { (0,0,0,0) = 3 }
rule : 4 100 { (0,0,0,0) < 0 }
#EndMacro


#BeginMacro(rule1)
rule : 0 100 { (0,0,0,0) + (1,0,0,0) + (1,1,0,0) + (0,-8,0,0) = 11 }
#EndMacro

#BeginMacro(Heat)
(0,0,0,0) > 30
#EndMacro
```

Figure 74: File life.inc that contains some macros used in the Game of Life 4D

```
#BeginMacro(Rule2)
rule : 0 100 { (0,0,0,0) = 7 }
rule : { (0,0,0,0) + 2 } 100 { t }
#EndMacro

#BeginMacro(Rain)
(0,-8,0,0) > 25
#EndMacro
```

Figure 75: File life–1.inc that contains the remaining macros for the Game of Life 4D

10 Appendix B – The preprocessor and temporary files.

When the preprocessor is used to resolve macros (by default the preprocessor is enabled), it will create a temporary file for the model with all macros expanded and all the comments erased. This temporary file is then passed to the simulator for its interpretation. If the use of the preprocessor with the parameter **-b** is disabled and macros are used, the model will not be processed correctly.

The name of the temporary file is the value returned by the instruction *tmpnam* of the *GCC*. The directory where the temporary files are located will be selected according to the following criteria:

1. When CD++ is compiled, the name of directory defined by *P_tmpdir* <*stdio.h*> will be used, unless this is the root directory.

   In *Linux* this variable usually has the value: "**/TMP**", while in the version of the *GCC* 2.8.1 for *Windows*–32 bits, this variable references to the root directory of the disk unit that is in use.

2. If *P_tmpdir* points to the root directory, then the name defined by the environment variable **TEMP** will be used.

3. If no **TEMP** variable is defined, then the value of the environment variable **TMP** will be used.

4. Finally, if the **TMP** is neither defined, the current directory will be used.