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

2001

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

Autor
Alejandro Troccoli

Director
Dr. Gabriel Wainer

Abstract 3
1 Introduction 4
2 The Parallel DEVS formalism 7
2.1 Parallel DEVS Atomic Models ..... **Error! Bookmark not defined.**
2.2 Parallel DEVS Coupled Models ..... **Error! Bookmark not defined.**
3 Cellular Automata and the Parallel Cell-DEVS formalism 13
3.1 The Parallel Cell-DEVS formalism 13
3.2 Cell-DEVS Quantization 18
4 Abstract simulator for distributed Parallel-DEVS 20
4.1 Parallel DEVS Abstract Simulators 20
5 Parallel Simulation 31
6 CD++ 35
6.1 Atomic model definition 35
7 Parallel CD++ 37
7.1 Warped API 37
7.2 An overview of parallel CD++ 38
8 Results 40
8.1 An extended version of the GPT model 40
8.2 A heat diffusion model 42
8.3 A measure of model parallelism 47
9 A flow-injection Cell-DEVS model 51
9.1 Flow injection analysis 51
9.2 A Cell-DEVS model for flow-injection 52
9.3 Simulation results 56
10 Conclusions and further developments 58
11 References 59

Abstract

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

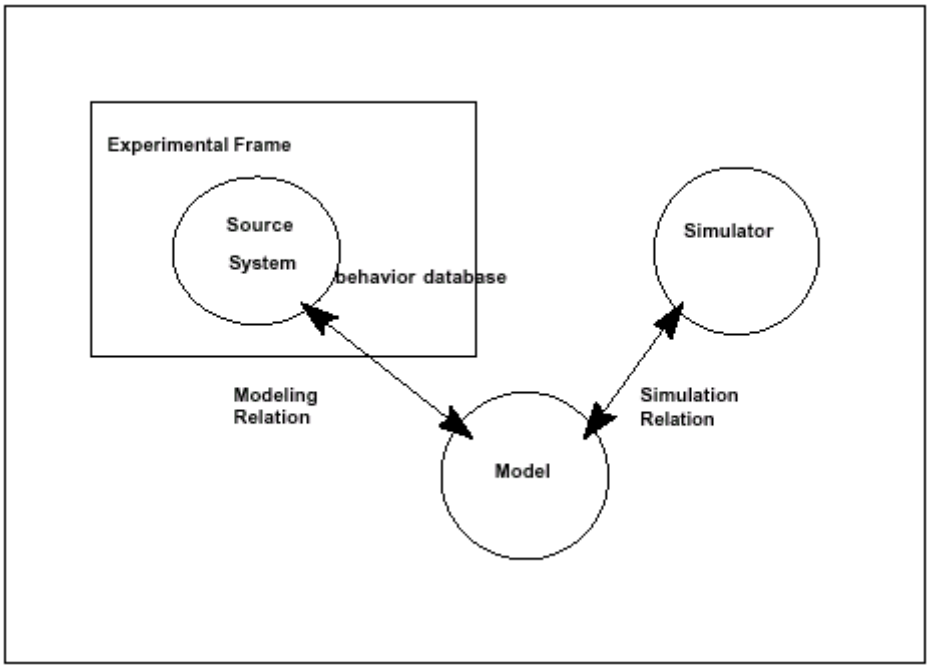


Figure 1 : The basic entities and their relationships [Zei00]

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

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

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

There exist at present quite a number of simulation techniques and paradigms. Among these, the **DEVS** formalism [Zei76] provides a framework for the construction of hierarchical models in a modular manner, allowing for model reuse and reducing development time and testing. In DEVS a model is specified as a black box with a state and 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 high level description of how the simulation of DEVS models should be executed by a **simulator**. Two kinds of **simulators** are defined, one for atomic and another one for coupled models, this latter known as a **coordinator**. These simulators progress through the simulation by exchanging messages as described by the abstract simulation mechanism.

Timed Cell-DEVS [Wai98] is a formalism based on DEVS for the simulation of cellular models. A cellular automaton is a lattice of cells, each of which has a value and a local rule that defines how to obtain a new value based on the current state of the cell and the values of neighboring cells. Cells are updated synchronously all at the same time. Timed Cell-DEVS defines a cell as a DEVS model and a cellular automaton as a coupled model, and introduces a new way of defining the timing of each cell which is more flexible than the existing synchronous approach. In Timed Cell-DEVS each cell defines its own update delay.

CD++ is a tool for the simulation of DEVS and Cell-DEVS models which has been used to simulate a variety of models including: traffic, forest fires, ants and watershed simulation. Simple models were easily handled by the tool, but the execution of complex models requires a computing power that stand alone computers do not provide. It was then proposed that parallel execution should be used.

Not only parallel execution was being demanded for Cell-DEVS but also for DEVS models. But the DEVS formalism suffered from serialization constraints that would not allow for a parallel implementation. Therefore, it was revised and the Parallel DEVS (P-DEVS) [Cho94a] formalism was proposed. The Cell-DEVS formalism was also revised [Wai00] and the Parallel Cell-DEVS formalism followed.

It is the aim of this work to modify CD++ to run Parallel Cell-DEVS on a distributed environment, providing a tool that will not only reduce execution times but also allow larger models. When P-DEVS was proposed, the abstract simulator was changed to implement the new semantics. This new simulator, 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 because it does not distinguish messages sent over the network from those sent between objects on the same process. Therefore, there was a need to extend the P-DEVS abstract simulator for distributed environments. This work addresses this issues by further specializing coordinators into master and slave.

For the new parallel version of CD++, a simulation kernel that would encapsulate all the lower level network communications was required. In parallel simulation, the execution is divided into a set of logical processes, each running on a different CPU. Logical process communicate with each using timestamped messages. For correct results to be obtained, a way of synchronizing the logical process for correct message processing must be defined. There are three approaches to synchronization between logical process: optimistic, pessimistic, and no synchronization at all (application level synchronization). A parallel simulation kernel must provide one of these.

During the design phase of parallel CD++, some research was done to evaluate existing simulation kernels and the Warped project was found. Warped is a project at the University of Cincinnati dedicated to the implementation of a simulation API to support different parallel simulation kernels. Two kernels are currently provided: an optimistic kernel that implements the TimeWarp protocol and a NoTime kernel that uses no synchronization. Further work was carried out at the Universidad de Buenos Aires, and a pessimistic kernel that complied with the Warped API was implemented. Having three different simulation kernels with the same API, Warped proved ideal for parallel CD++, which was therefore written to run on top of Warped and currently supports the TimeWarp and NoTime kernels. Switching between kernels is just a matter of setting the proper compilation arguments.

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. Chapter 2 presents the DEVS and Parallel DEVS formalisms and Chapter 3 the Cell-DEVS and Parallel Cell-DEVS counterpart. 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. After this presentation, chapter 6 will introduce CD++ and chapter 7 its parallel version, with special mention of implementation issues using the Warped kernel. Chapter 8 will show some results obtained, chapter 9 will show a chemical diffusion model so large that parallel execution is required, and then the conclusions will follow. A complete user's for parallel CD++ guide is also provided.

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

The original DEVS model is a structure:

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

where

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

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

The semantics for this definition are as follows. At any given time, a DEVS model is in a state $s \in S$ and in the absence of external events, it will remain in that state for a period of time as defined by ta(s). The ta(s) function can take any real value between 0 and ∞. A state for which ta(s) = 0 is called a **transient state**. On the other hand, if ta(s) = ∞, the system will stay in that state forever unless an external event is received. In such a case, s is called a **passive state**. Transitions that occur due to the expiration of ta(s) are called **internal transitions**. When an internal transition takes place, the system outputs the value $\lambda(s)$, and changes to state $\delta_{int}(s)$. A state transition can also happen when an external event occurs. In this case, the new state is given by δ_{ext} based on the input value, the current state and the elapsed time. Figure 2 illustrates this definition by specifying a model of a computer processor using DEVS.

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

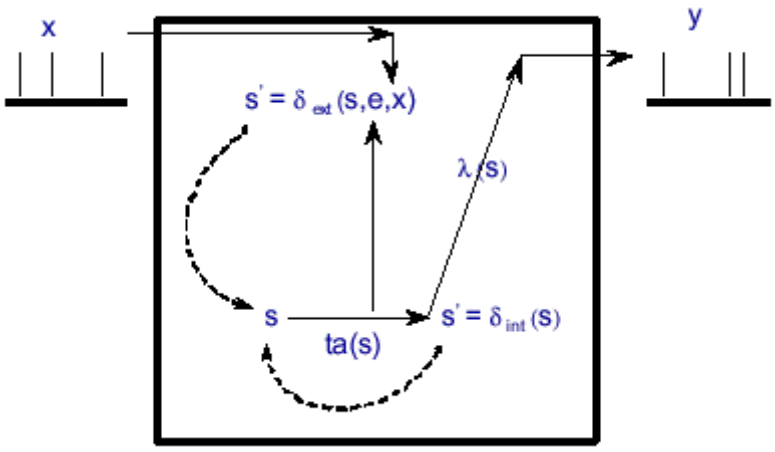
When the processing time has expired, a state transition will take place. At this time, the output function is called followed by the internal transition function. Continuing with our description,

$\lambda(busy) = jobId$

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

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

(a)



(b)

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

A *coupled model* is a structure:

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

where

D is a set of components.

for each *i* in *D*,

M_i is a component.

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 when there are more than one components schedule for an internal transition at the same time. The first model to make its internal transition will produce and output that may be translated to an external event being received by another component model that is already scheduled for an internal transition at that time. But then, should this second model process the external transition first with $e = ta(s)$ or should the internal transition take place first and then the external transition with $e = 0$? The way the DEVS formalism solves this is by the use of the *select* function. Only one model of the group of imminent models will be allowed to be with $e = 0$. The other imminent models will be divided in two groups: those that receive the external output from this model, and the ones that do not receive this output. The first group will execute their external transitions functions with $e = ta(s)$ and

the second group will be among the group of imminent models for the next simulation cycle, which may require again the use of the select function to decide which model will execute first.

This tie-breaking approach is a potential source of errors since the serialization 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.

A P-DEVS model is described as a set of basic and coupled models. In addition, the model's interface was also revised. 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.

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 be seen as another basic model. Therefor, 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 value*s;

$Y_M = \{(p,v)|\ p \in OPorts, v \in Y_p\}$ 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.
- 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 $e \leq t$a(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 i.e.,

$((d, op_d), (e, ip_d)) \in IC$ 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 3.

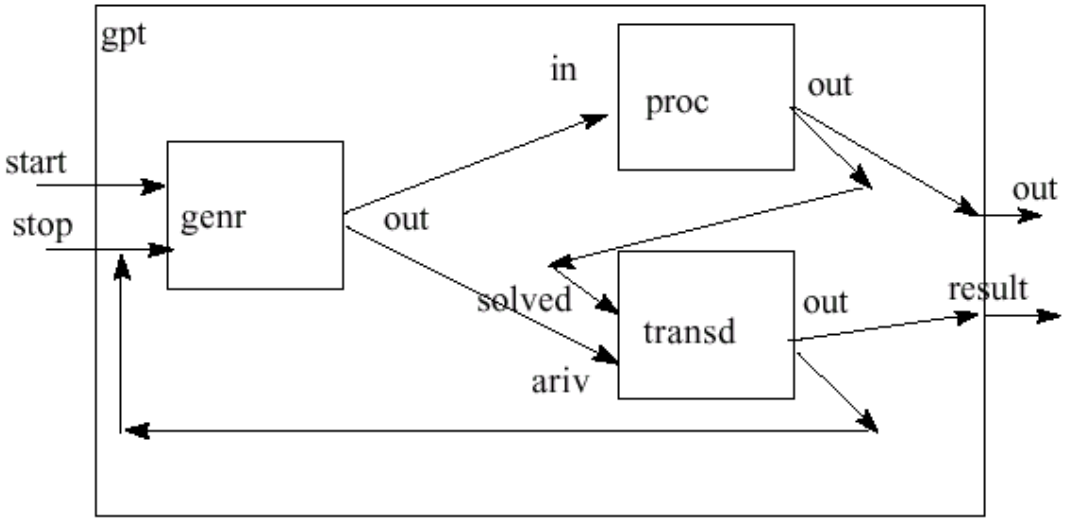


Figure 3 : The GPT coupled model. [Zei00]

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).

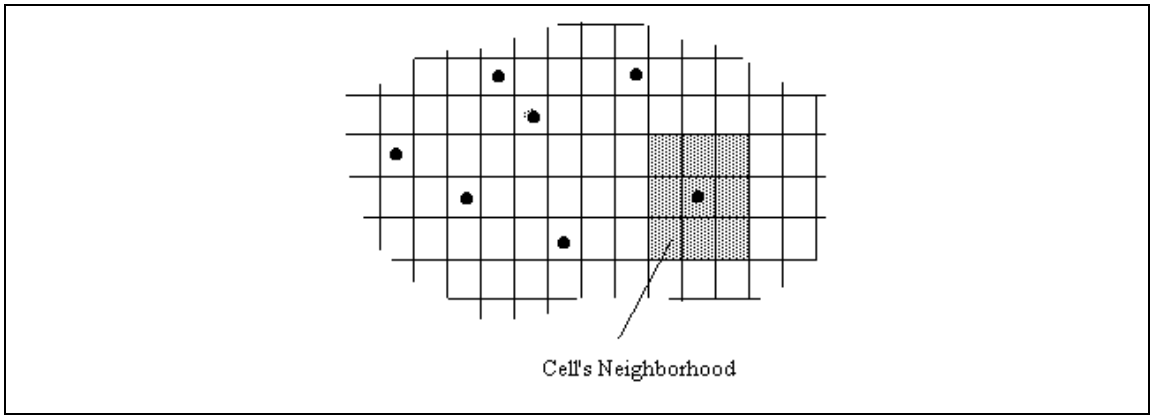


Figure 4 : Sketch of a Cellular Automaton [Wai00]

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.

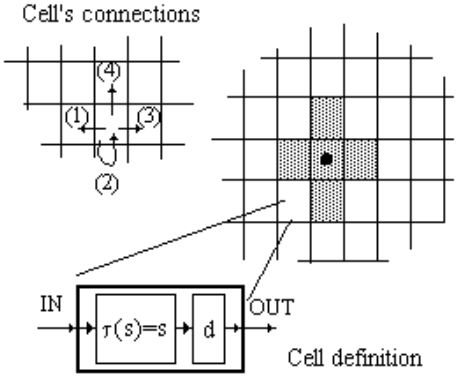


Figure 5 : Informal definition of a Cell-DEVS model [Wai98]

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 [Wai00]:

- 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 $\in$ S is the status value for the cell,

s' $\in$ S is an intermediate status value for the cell;

phase $\in$ {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

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

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

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

$\delta_{int}: \theta \rightarrow S$;

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

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

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

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

$\lambda: S \rightarrow Y^b$; and

$D: \theta \text{ x } N \text{ x } d \rightarrow R_0^+ \cup \infty$.

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 $\boldsymbol{\tau_{con}}$, 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.

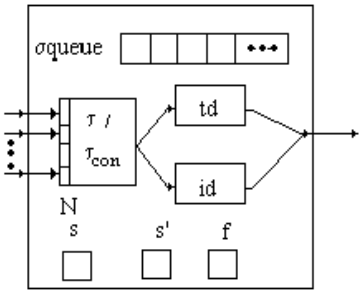


Figure 6 : Cell's definition [Wai00]

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.

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

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 [Wai00], 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}:

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

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

λ:

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

δ_{ext}:

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

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

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

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

3.3 Cell-DEVS Quantization

Recently, a theory of quantized models was developed (Paper Gabriel). 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 7. 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.

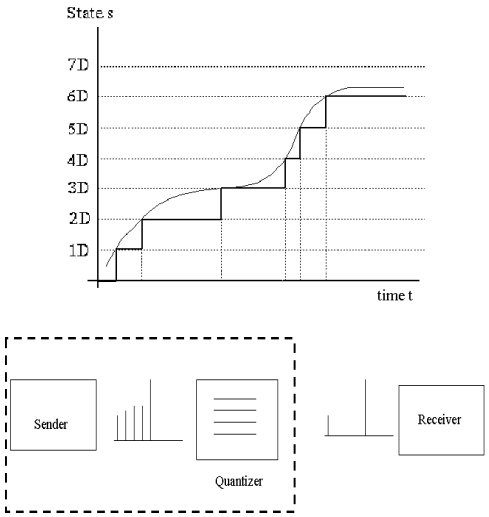


Figure 7 : Quantization (Zeigler et al 1999)

In (Paper Gabriel) 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 [Paper Gabriel] 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}:

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

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

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

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

where

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

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 simulation. This simulation mechanism is implemented by abstract simulators. 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. In a distributed environment, there is considerable communications overhead which can not be ignored. Therefore, the abstract simulator should restrict the number of messages over the network to a minimum.

As a result, a abstract simulator for distributed environments was developed and will be now presented.

4.1 Parallel DEVS Abstract Simulators

The simulation is carried out by DEVS processors. As in the existing definition of the abstract simulator [Cho94b], the DEVS processors will be specialized into two different simulation engines, **simulator** and **coordinator**. Basically, the role of the *simulator* is to invoke an atomic's model transition and external event functions. On the other hand, a *coordinator* is paired with a coupled model and has the responsibility of translating its children' output events and of keeping the time of the next imminent/s dependants.

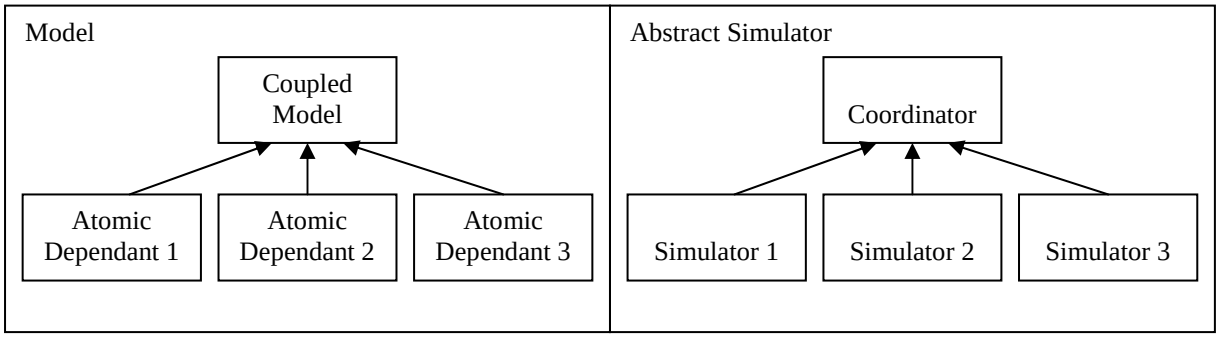


Figure 8 : Correspondence between the model and the DEVS processors

As it is shown in Figure 8, every coordinator has a set of child DEVS processors. When a simulation is run in distributed fashion, each machine will run one logical process which will host one or more DEVS processors. Under these assumptions, a coordinator's children need not be executing on the same logical process. If the correspondence between models and DEVS processors is one to one, then every coupled model is associated to only one coordinator. Then every message sent to child processors running on a different CPU will require inter-process communication. Figure 9(a) illustrates this case. It shows a coordinator sending a message to its 8 children distributed on two CPUs. Four inter-process messages are required for the four children running on processor 1.

If 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 9(b) illustrates this case. For the same coupled model, there are two coordinators, one in logical process 0 and another 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. Upon receiving a message from a child, a coordinator could forward this message to all the coordinators for the model. This would require all coordinators to know about each other. For instance, if coupled model *A* is a child of coupled model *B*, then *B´s* coordinators would have to interact with *A´*s coordinators. If handled uncarefully, this

communication can turn out producing a big number of inter-process messages. In such a scenario, a way of keeping the number of inter-process messages to a minimum is to have only one of the coordinators to receive messages from or route messages to the parent´s model coordinator. This specialized coordinator will be known as a **master coordinator** and all other model coordinators will be **slaves**. The master coordinator for model *A* will then be the only one that can receive or send messages to *B´*s local coordinator.

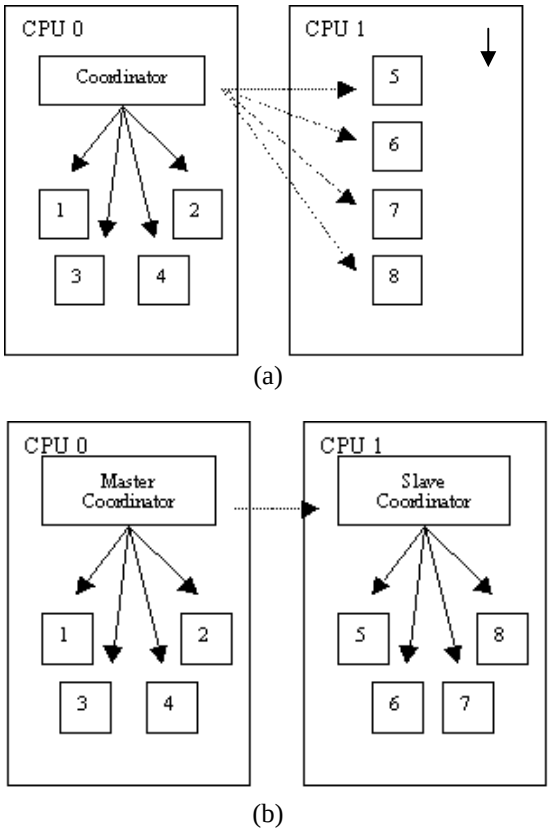


Figure 9 : (a) A single coordinator sending a message to all its child processor. Dashed lines = interprocess messages. (b) A master- slave pair sending messages to all their children processors.

When master and slave coordinators are used, DEVS 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 must be defined. This relationship 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*.

The simulation advances as a result of the exchange of messages between parent and child DEVS processors. Every message is a pair of the form *(type, time)* and can belong to one of two categories: synchronization messages and content messages. The synchronization messages are (@ , *t*), (*, *t*), and (*done*, *t*) and the contents messages are (*y*, *t*) and (*q* , *t*).

The synchronization messages (@ , *t*), (*, *t*) are sent from a parent DEVS processor to its imminent children. A (@ , *t*) is used to tell the children to send their outputs and (*, *t*) tells the children to invoke

their transition function (whether it corresponds to execute an external, internal or confluent transition). All outputs produced by a model are translated to (*y*, *t*) messages between a child DEVS processor and its parent. Finally, those external messages that arrive from outside the system or that are generated as a result of an output message being sent to an influencee are sent as (*q* , *t*) messages.

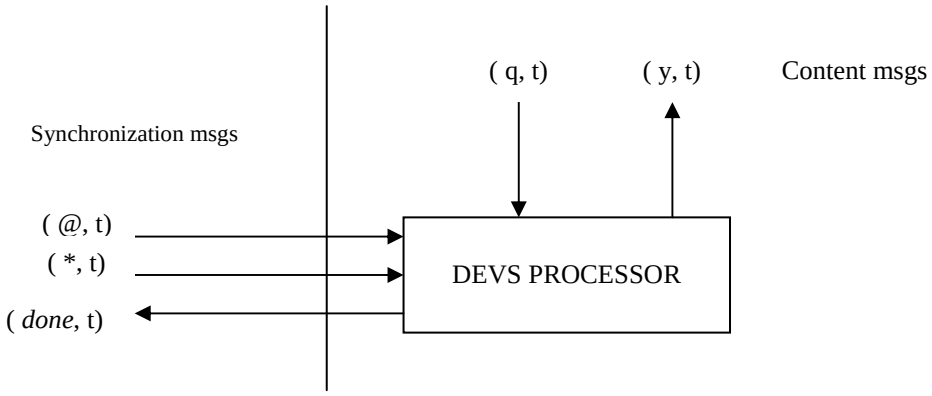


Figure 10 : Messages a DEVS processor receives and sends

It is assumed that any two messages sent from the same source to the same destination will preserve their original ordering.

The complete behavior of a DEVS processor is described 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

When a simulator receives a (@ , *t*) it executes the atomic model's λ function and sends the output to the parent coordinator.

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 indicates a model's transition function must be executed. The transition function to be executed will depend on t and the content's of the queue. If $t < t_N$, then it is not the time for an internal transition, and it must be the case that the queue 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, is responsible for the simulation of a coupled model. It translates output events to input events and keeps track of the imminent models. Each coordinator has a set of child processors which correspond with the coupled model components. For a master coordinator the set of child processors is made by the set of slave coordinators, the set of local child simulators and the set of local child master coordinators. A DEVS processor is local if it is executing on the same processor.

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 DEVS processor
S 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

For describing the behavior of a *master coordinator* upon receiving an output message, two cases need to be distinguished:

an output message (*y* , *t*) received from a child *i* that is not 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. This is done to reduce the number of messages sent across the network. 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 and it will be the responsibility of the *slave coordinator* to 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 $t_L \le t \le t_N$

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 the output events are routed down to child processors, if the message is to be forwarded to a *slave coordinator* the *z* translation will not be applied. Instead, the original *q* message will be 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.

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.

SLAVE 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

As it can be noticed, there is no difference on how both *master* and *slave coordinators* handle a (@, t). However, the set of child processor of a *slave coordinator* is different. For a *slave coordinator* the set of child processors is made by the set of local child *simulators* and the set of local child *master coordinators* only.

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 parent 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 parent coordinator

end if

end if

end when

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

lock the *bag*

Add event q to the *bag*

unlock the *bag*

end when

SLAVE COORDINATOR

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

if $t_L \le t \le t_N$

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 parent 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

This abstract simulator mechanism will be able to handle both, Parallel DEVS and Parallel Cell-DEVS models because the latter one is a specialization of the first one.

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 DEVS processors.

Logical processes (LPs) talk to each other through time-stamped events that move the simulation forward. Events must be processed in the order defined by their timestamps for correct results. It does not always suffice to have a queue on each logical process and advance the simulation by processing the first event on the queue, ignoring the other LPs. Such case is illustrated in Figure 10 which 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 is generator for LP 1, D, 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.

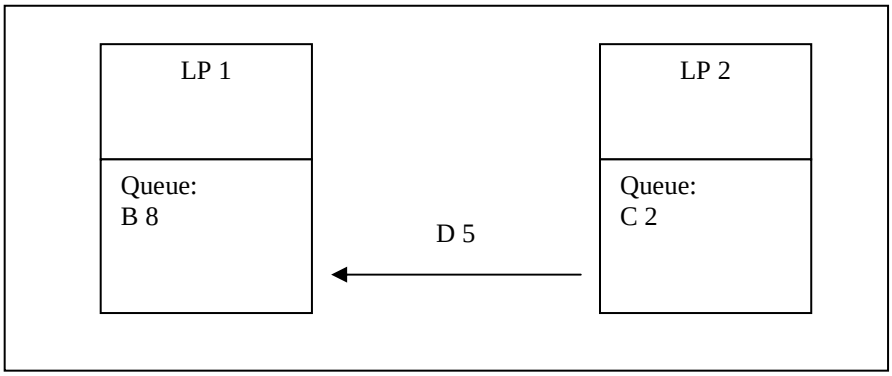


Figure 11 : Execution of the first queued message does not always guarantee correct results.

Then, 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. Special consideration will be given to this approach later because the Parallel-DEVS abstract simulator presented in the previous chapter does provide by itself a synchronization mechanism.

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 equalt 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 10, where there are two logical processors 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 time stamp 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.

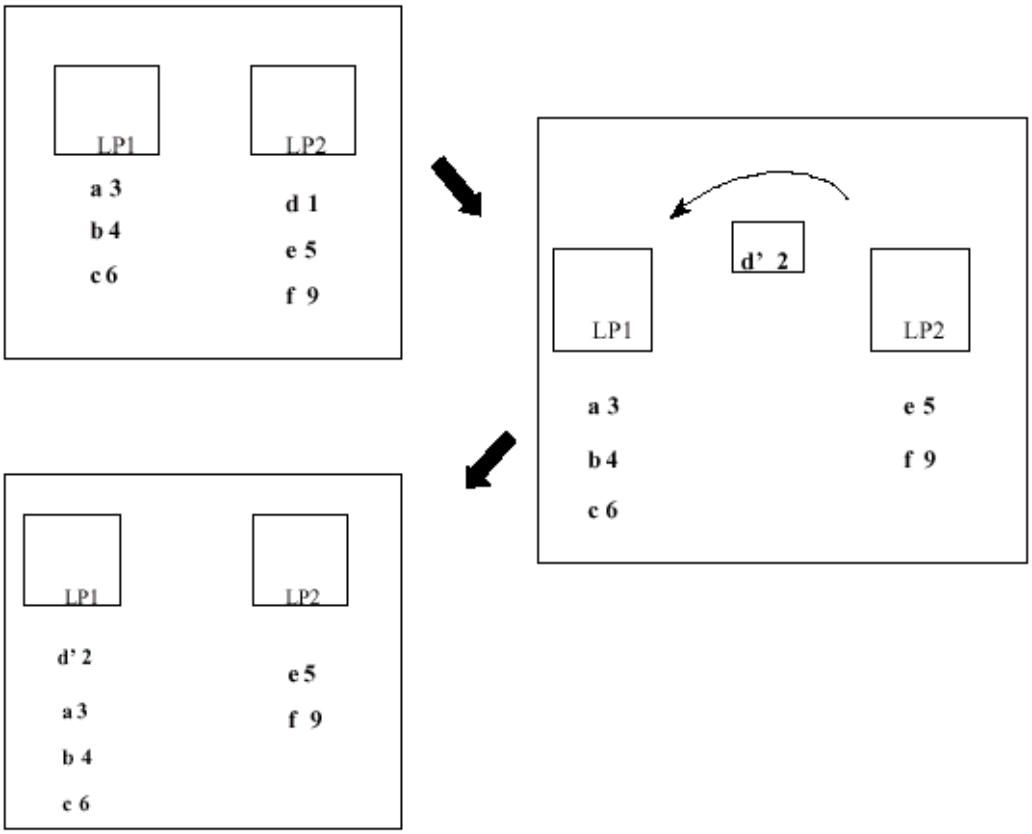


Figure 12 : LPs with conservative synchronization [Zei00]

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 time in the future up to which it is safe to process messages. Each LP must then carry a lookahead for determining the time up to which it is safe to process time-stamped inputs. In Figure 12, 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 process 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. An improvement on these 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 13 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.

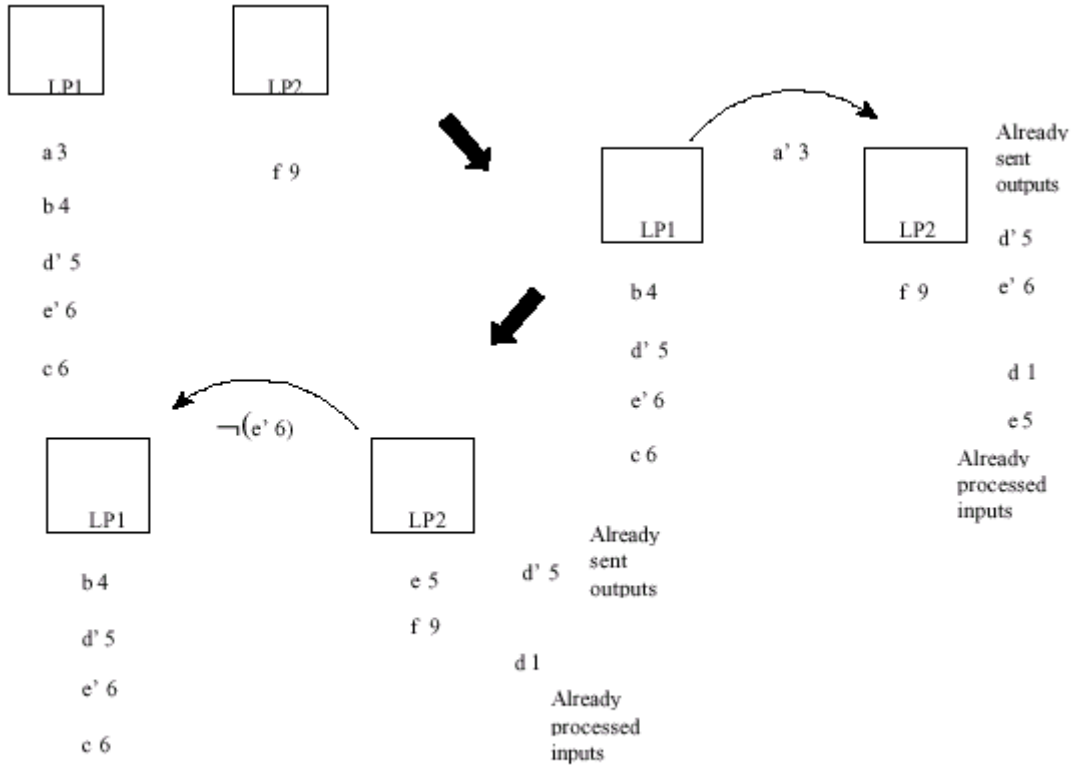


Figure 13 : Event processing in an optimistic scheme

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. 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, which is 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 [Jeff].

5.3 Synchronization for the DEVS abstract simulator

Parallel CD++ will run the abstract simulator described in Section 4. The DEVS processor (root coordinator, simulator, slave coordinator or master coordinator) will be the simulation objects that will run on the available LPs and a suitable synchronization mechanism should be chosen.

When analyzing the behavior of the simulator and master and slave coordinator, it can be seen that upon receiving any of the *(*,t),(@, t), (done,t) , (y,t)* or *(q, t)* messages, any other message that is sent will have the same timestamp *t*. The root coordinator is the only DEVS processor that will cause the time to advance by sending a new message with the time of the next imminent model or external event. 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 updated.

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 LP. An LP will be allowed to receive multiple messages with a timestamp equal to its LVT. The only constraint that needs to be placed is that the two or more events sent from a source object S to a destination object D should preserve the same ordering upon arrival to D.

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

Proof
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 application itself. However, as it will be seen Section 7, parallel CD++ will be implemented over a simulation kernel that provides all three synchronization mechanisms. This will allow future abstract simulators to take advantage of the TimeWarp protocol and send events in the current simulation cycle with a timestamp in the future.

6
CD++

CD++ implements the DEVS theory. It allows to define models according to the original DEVS formalism (Wainer *et al*. 2000, Rodríguez and Wainer 1999). A set of independent applications related with the tool allow the user to have a complete toolkit to be applied in the development of simulation models.

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 class hierarchy programmed in C++. Coupled and Cell-DEVS models need not be programmed. The tool provides a specification language that defines the model's coupling, including the initial values and external events, and the local transition rules for Cell-DEVS models.

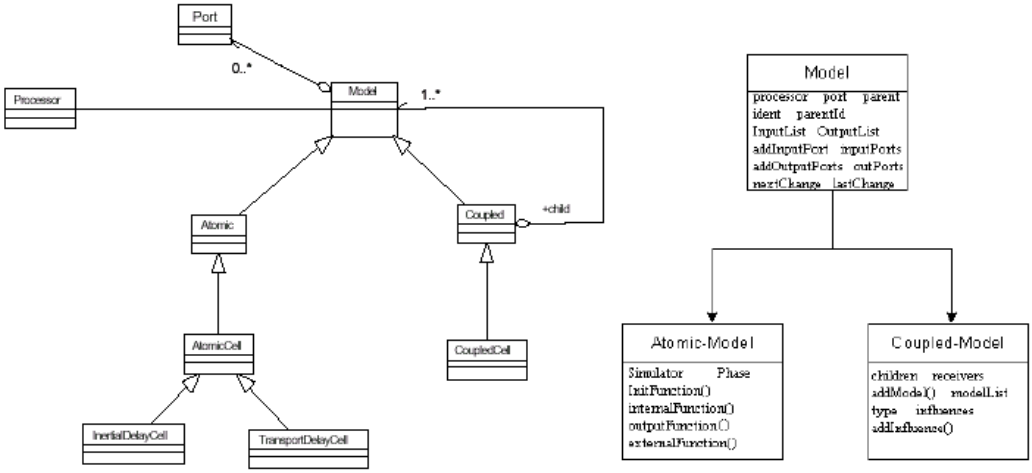


Figure 14 : CD++ Models and Processors.

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. Coupled models are defined using a specialized specification language. Following, we explain how to incorporate atomic and coupled models to be simulated.

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 *passiv*e. 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.registerNewAtomic*s(). 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 *tim*e. 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 suitable layered architecture had to be chosen.

It was decided that *Parallel CD++* should be built on top of a modified version of Warped [8]. The Warped project is an attempt to make a freely available simulation kernel for parallel and distributed environments. 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 [Jeff]; the second is an unsynchronized kernel. In addition, a sequential kernel is also provided for running standalone simulation. Further efforts were done at the University of Buenos Aires to develop a conservative kernel [Sul].

For the distributed simulation kernels, Warped uses MPI for the message passing. The complete layered architecture is shown in Figure 15.

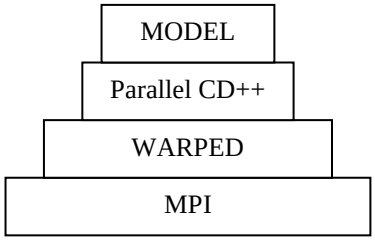


Figure 15 : Parallel CD++ layered architecture

7.1 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 16 shows the Warped API.

The Warped kernel presents an interface to the application that is based on Jefferson's original paper on TimeWarp. 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.

```
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 16 : Warped API

7.2 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 17).

```
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 17 : The Atomic class

In addition, Parallel CD++ provides a way of allowing the user to define the data carried by output and external events. Originally, in CD++, this was restricted to real numbers.

```
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 18 : 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 for each atomic model the machine on which it will run. 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.

8
Results

The aim of making CD++ to run in parallel is to have a tool that will reduce the simulation time. Therefore, the results to be presented in this chapter will show how execution time of different models changes with different configurations. 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 set results were 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.

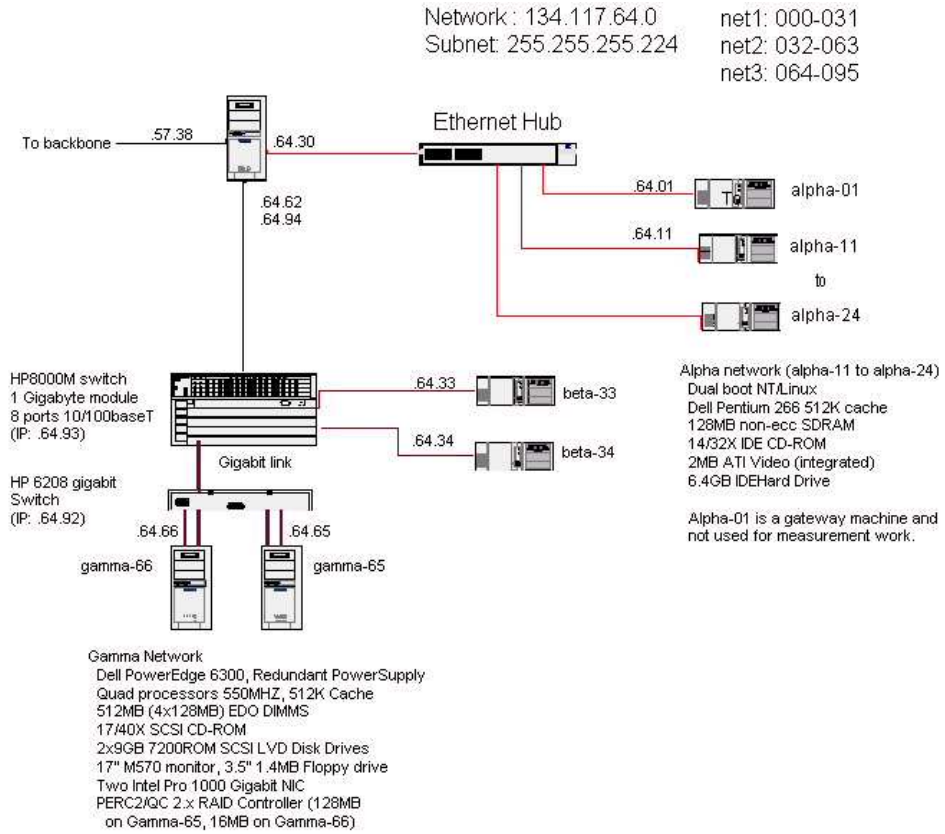


Figure 19 : The RADS measurement networks

8.1 An extended version of the GPT model

Parallel CD++ was first tested with an extended version of the Generator-Producer-Transducer model (GPT). The GPT model simulates a processor receiving jobs and calculates its throughput and load. It consists of a generator, a queue, a processor and a transducer, as shown in Figure 20. The generator outputs jobs periodically. When a new job 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 it 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; the transducer will compute the turnaround time and update the throughput and cpu usage values, which it will output periodically.

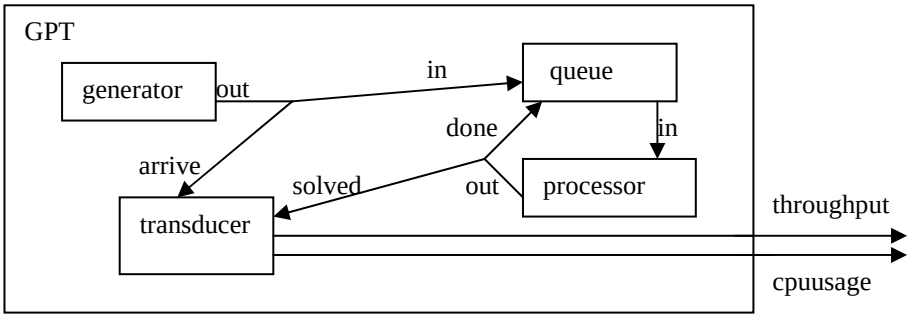


Figure 20 : The GPT model

The definition of this model for Parallel CD++ is shown in Figure 21.

```
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 21 : Definition of the GPT model

The extended version of the GPT model consists of several copies of the GPT model just shown. Tests were conducted with 12, 48 and 96 copies, running on 1 to 12 machines. The results of running this model are shown in Figure 22. All random variables that were present in the model definition were eliminated to obtain comparable results.

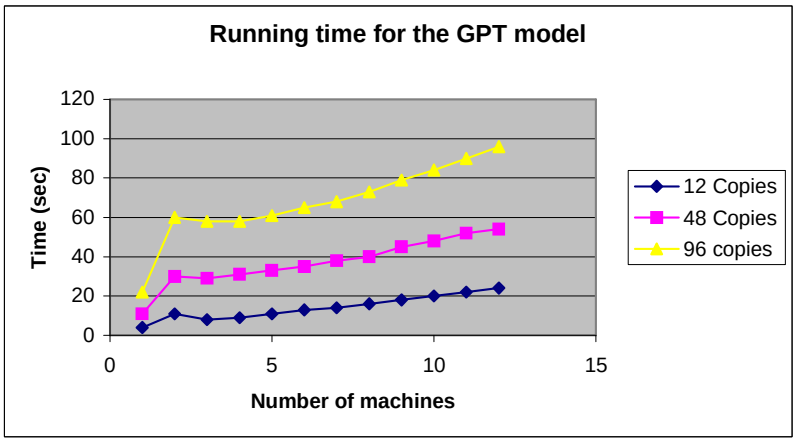


Figure 22 : Execution time in seconds of 12, 48, and 96 copies

As it can be seen, the execution times for this model did not behave as expected. As more machines are added, the execution time increases. To determine the causes for such a behavior further tests were conducted. To verify if the communications overhead was being the cause for such an increase in the running time, the model was rewritten to include a delay in the external function. 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, that do confirm this hypothesis, are shown in Figure 23.

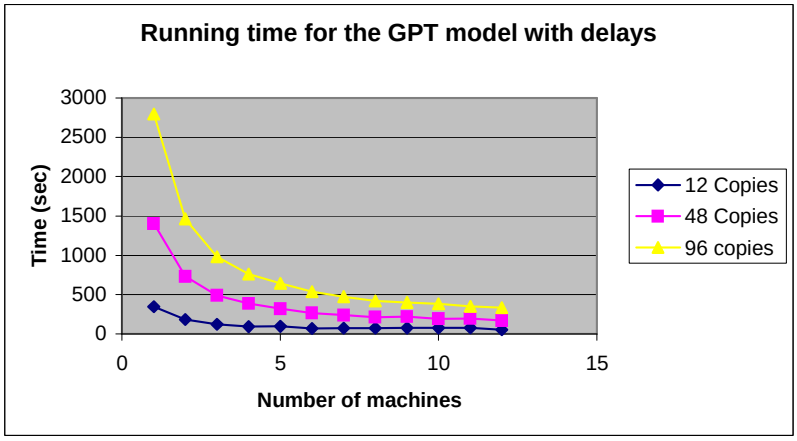


Figure 23 : Execution time in seconds of 12, 48, and 96 copies of the GPT model with delays on 1 to 12 machines, for a simulation virtual time of 10 minutes. Results show the minimum time of three runs independent runs.

8.2 A heat diffusion model

Parallel CD++ was also tested with a heat diffusion model. In this model, a surface is represented by a 50 x 50 cellular automaton, each cell containing a temperature. In each simulation cycle, the temperature of a cell is updated to the average of the values of the neighborhood. In addition, a heat generator is connected to the cells (25, 25) and (10, 10), generating temperatures in the range [24, 40] with uniform distribution. Also, a cold generator that creates temperatures in the range [10, 15] with uniform distribution, has been connected to the cells (10, 40) and (40, 40). Both generators create values after *x* seconds, where *x* follows an exponential distribution with mean 50 seconds. When any of the generators outputs a new value, the cell to which it is connected will take that value. For testing purposes, the random number generators were disabled to obtain comparable results.

The definition of the model using the language provided by the tool is showed in Figure 24. The top model and its components are defined between lines 1 and 4. Between lines 6 and 26, the model representing the surface is defined. It is composed of a cellular automata of 50x50 cells with an initial temperature of 24° C. In the lines 28 and 29 the local transition function is defined.

Lines 31 and 32 define the transition function upon receiving an external event from the heat generator, and lines 34 and 35 for transition triggered by external events coming from the cold generator. Lines 37 to 47 define the distribution parameters for the generators.

```
01   [top]
02   components : surface generatorHeat@Generator generatorCold@generator
03   link : out@generatorHeat inputHeat@surface
04   link : out@generatorCold inputCold@surface
05
06   [surface]
07   type : cell
08   width : 50
09   height : 50
10   delay : transport
11   defaultDelayTime  : 100
12   border : wrapped
13   neighbors : surface(-1,-1) surface(-1,0) surface(-1,1)
14   neighbors : surface(0,-1)  surface(0,0)  surface(0,1)
15   neighbors : surface(1,-1)  surface(1,0)  surface(1,1)
16   initialvalue : 24
17   in : inputHeat inputCold
18   link : inputHeat in@surface(25,25)
19   link : inputHeat in@surface(10,10)
20   link : inputCold in@surface(40,40)
21   link : inputCold in@surface(10,40)
22   localtransition : heat-rule
23   portInTransition : in@surface(25,25)  setHeat
24   portInTransition : in@surface(10,10)  setHeat
25   portInTransition : in@surface(40,40)  setCold
26   portInTransition : in@surface(10,40)  setCold
27
28   [heat-rule]
29   rule : { ((0,0) + (-1,-1) + (-1,0) + (-1,1) + (0,-1) + (0,1) + (1,-1) + (1,0) + (1,1)) / 9 } 10000 { t }
```

```
30
31   [setHeat]
32   rule : { uniform(24,40) } 1000 { t }
33
34   [setCold]
35   rule : { uniform(-10,15) } 1000 { t }
36
37   [generatorHeat]
38   distribution : exponential
39   mean : 50
40   initial : 1
41   increment : 0
42
43   [generatorCold]
44   distribution : exponential
45   mean : 50
46   initial : 1
47   increment : 0
```

Figure 24 : Definition of the heat diffusion model

Figure 25 shows a model partition for running the heat diffusion model on 4 machines. There are a total of 252 simulators that have to be assigned to 4 CPUs. Line 1 defines the location for the simulators associated to the generatorHeat and generatorCold atomic models. Lines 2 to 5 set where the simulators for the cells of the surface model will be running.

```
01   0 : generatorHeat generatorCold
02   0 : surface(0,0)..(24,24)
03   1 : surface(25,0)..(49,24)
04   2 : surface(0,25)..(24,49)
05   3 : surface(25,25)..(49,49)
```

Figure 25 : 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.

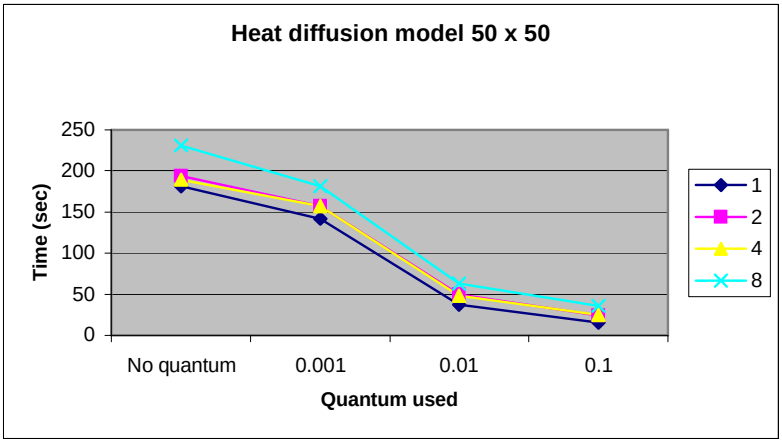


Figure 26 : Simulation execution time (seconds) for the heat diffusion model

The graph shows that:

- As the quantum is increased there is a reduction in the execution time.
- When the same quantum is used, adding more machines does not reduce the simulation time.

The results shown in Figure 26 were rearranged to display the number of machines in the X axis and the new graph is shown in Figure 27. It is clear that, as more machines are added, the execution time increases. When this behavior was observed for the extended GPT model, adding a delay to the external transition function produced the expected results. For cellular models, another way of increasing the computing load is to increase the model size. So the heat diffusion model was rewritten as a 100x100 cellular model to asses if the execution time would behave differently. The results are shown in Figure 28.

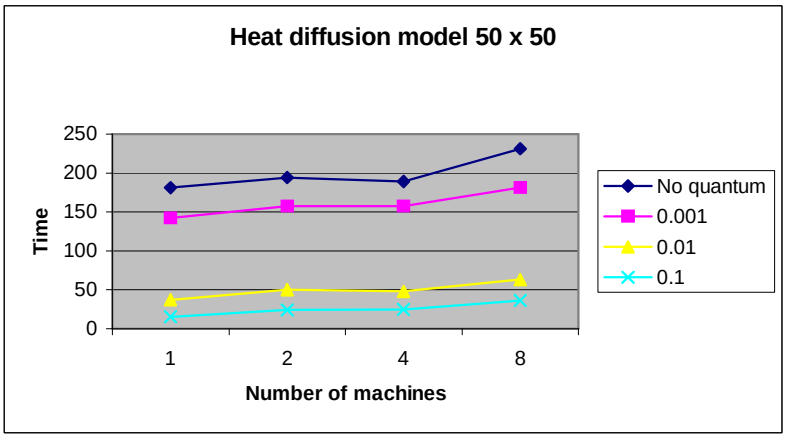


Figure 27: Simulation execution time for a 50x50 heat diffusion model.

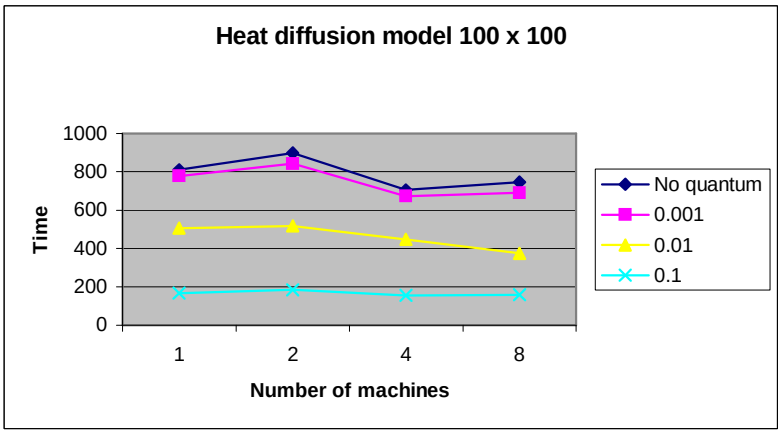


Figure 28: Simulation time for a 100x100 heat diffusion model

The 100 x 100 cellular model showed a different behavior. Taking for instance, the executions without quantum, it can be observed that the smallest execution time was achieved with 4 machines. The execution time for 2 machines was the greatest, and using 8 machines the performance was worse than using 4.

After these results, the abstract simulator of section 4 was studied thoroughly to determine the causes for such a behavior, specially for the time increase observed from 4 machines to 8 machines. As a result, the revised abstract simulator that is described in the next section was obtained.

9
A revised abstract simulator.

For cellular models, 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 shown in Figure 29.

#Machines	#Cells
1	0
2	698
4	890
8	1274

Figure 29: Number of cells with remote neighbors when different partitions are used.

When a cell sends an output, 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 sent as a *(q , t)* message to a model that is running on a different logical process, it just forwards the *(y , t)* message to the master coordinator who will then forward it to the corresponding recipients. Thus, an output message will whose final recipient is a logical process that is not the one running the master coordinator, will make to 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 30 shows how an output message from cell (25,0) is forwarded to cell (25,49). The dashed lines represent messages sent over the network.

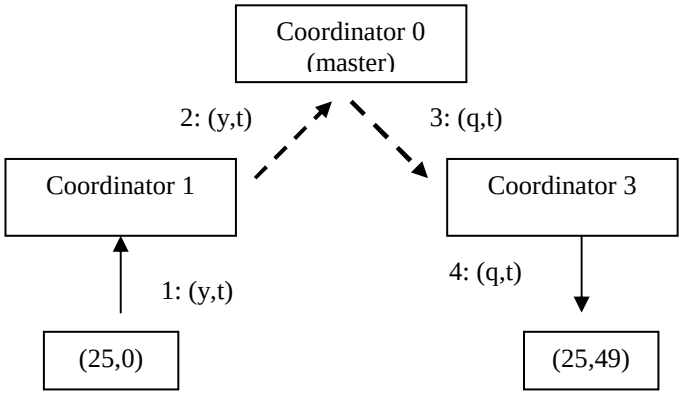


Figure 30 : Master - Slave coordinator output relaying.

This way of relaying messages between logical processes has a negative impact on:

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

From the numbers in Figure 29, it can be seen that for 8 machines, if all cells have an output to send the master coordinator for the coupled cellular model will receive 1274 messages. All but one eight of these output messages will then be forwarded to a different logical process.

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 parent coordinator, without going through the master coordinator. In this way, the relaying is avoided, as shown in Figure 31.

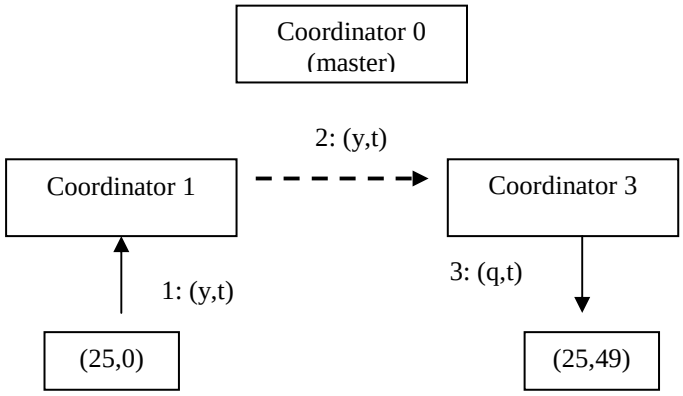


Figure 31: Revised output relaying.

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 mentions that there are three stages to be completed when a component is imminent:

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 30, 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 no longer know who the active slave coordinators are. As Figure 31, 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 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 32.

In Figure 32, 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 and so the master coordinator sends a *(done, t)* and receives a *(*,t)* that is forwarded to coordinator 1 and 3. Coordinator 3 can receive the *(*,t)* message before the *(y,t)* message, producing incorrect results. The problem here is that there is no knowledge of when all the sorting of output messages has concluded. The abstract simulator of section 4 did not have this problem because all output messages to remote models went through the master coordinator. Since the master coordinator always forwarded all outputs before sending a *(*,t)* message, and because MPI guarantees that messages between two logical process are received in the same order they were sent, the problem was avoid.

A correct abstract simulator would delay the *(done, t)* messages (number 3 in Figure 30) until all outputs have been received. A first approach might lead to having a coordinator acknowledge a *(y,t)*, but this again, leads to an enormous number of messages being sent.

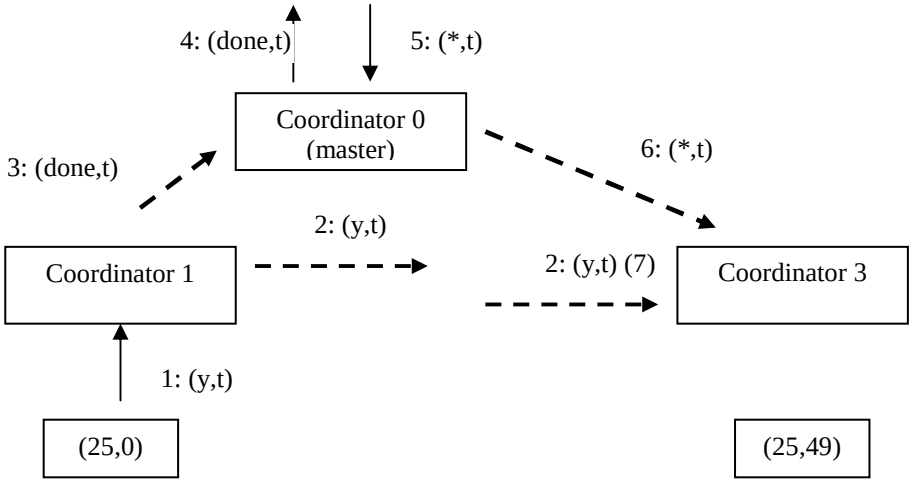


Figure 32: A (*,t) message is received before a (y,t) message

Instead, the following approach will be taken:

1. When a master coordinator receives a *(@, t)*, it will forward it to all its slave coordinators, whether they are imminent or not.
2. When a slave coordinator receives a *(@, t)*, it will send all of its *(y, t)* to the other coordinators.
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.

8.3 A measure of model parallelism

To have a better understanding of the factors that contribute to a reduction of a model's simulation time as more machines are used, a measure of a model level of parallelism was developed. Basically, this measure should have 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 at 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 measure 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 network, the execution time for each simulation cycle will be given by

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

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)*. Knowing the cycle time, the CPU usage at each LP can be obtained by dividing the used time by the cycle time, which is given by

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

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. All LP's CPU usage measures can be averaged to give a measure for the whole system. This measure 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 measure was used with the heat diffusion model. The results are shown in figure 26. As it is shown, the parallelism for the execution without quantum is keeps over 0.88 for all partitions, but falls considerably when a quantum is used. This helps to explain why for the execution using quantization adding more machines did not reduce the execution time. Adding more machines does not necessarily mean that there will more simultaneous execution.

As a subproduct, the total number of * messages sent during the whole was calculated. The results are shown in figure 27. Here it is interesting to see that using quantization does indeed reduce the number of messages sent. For a discussion on quantization to be complete, it remains to analyze the error incurred in each case, but that is not within the scope of this work.

Last, the evolution of the level of parallelism through the whole execution time is shown for the heat diffusion model when 2, 4 and 8 LPs are used. Figure 28 shows the results for the execution without quantization and figure 29 shows the results when a quantum of 1 is used. In the first case, the level of parallelism increases steadily till it reaches a value near to 1 at time 20 sec. On the other hand, the level of parallelism is constantly changing.

# Machines	No quantum	Q = 0.1	Q = 0.5	Q = 1
1	1,00	1,00	1,00	1,00
2	0,95	0,68	0,71	0,70
4	0,93	0,58	0,62	0,62
8	0,88	0,47	0,49	0,48

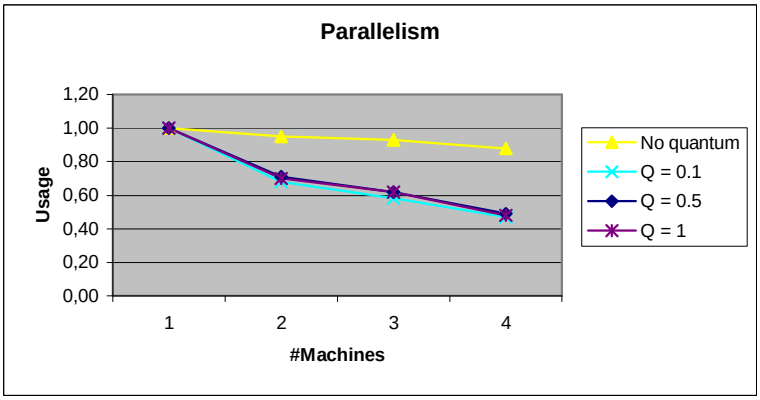


Figure 26 : Parallelism of the heat diffusion model with and without quantum

# Machines	No quantum	Q = 0.1	Q = 0.5	Q = 1
1	107148	17920	12357	11649
2	107201	17964	12409	11689
4	107307	18057	12486	11766
8	107522	18206	12637	11908

Figure 27 : Number of * messages sent when executin the heat diffusion model with and without quantum

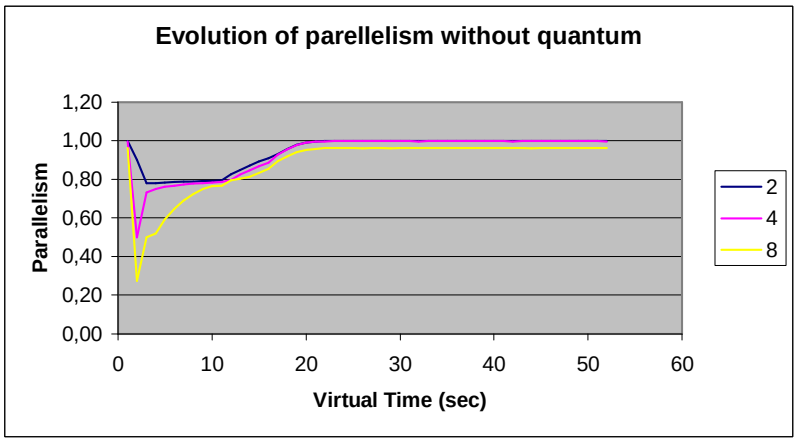


Figure 28 : Evolution of the parallelism for the heat diffusion model when no quantum is used. Results shown correspond to execution on 2, 4 and 8 LPs.

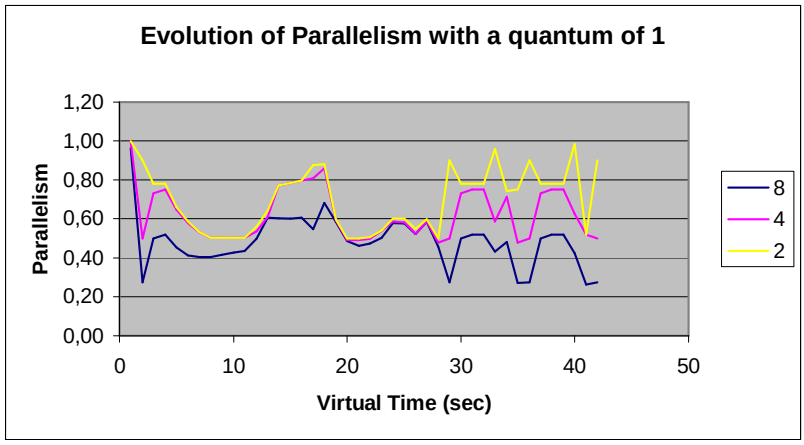


Figure 29 : Evolution of the parallelism for the heat diffusion model when a quantum of 1 is used. Results shown correspond to execution on 2, 4 and 8 LPs.

9
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.

9.1 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 33 presents a simple flow-injection apparatus.

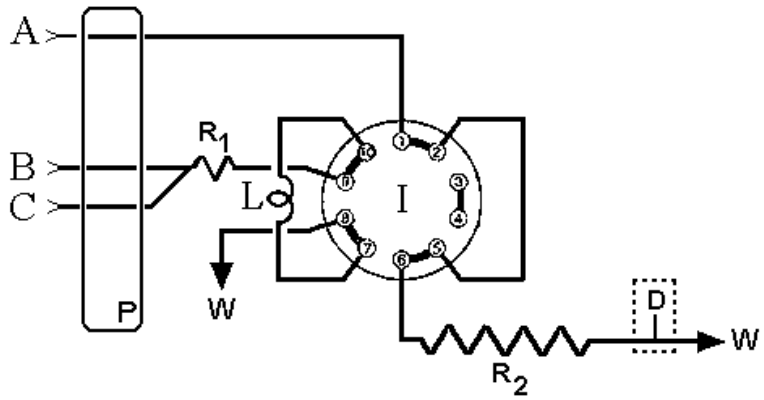


Figure 33 : A FIA manifold.

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 [1], 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 [1] 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.

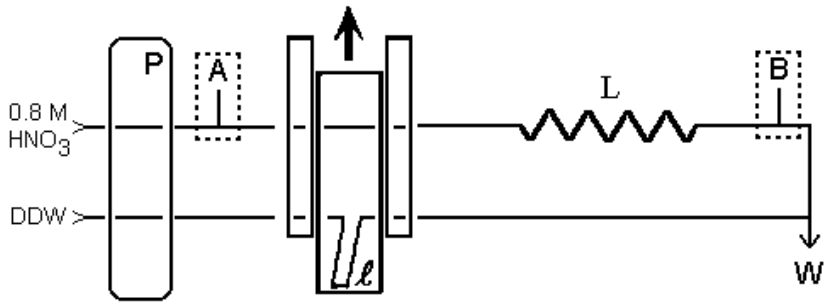


Figure 34 : FIA manifold for continuously monitoring. P = pump; l = loop; L = reactor; W=waste; A, B = detection points. Punctual detection: suitable detector in point B; integrated detection: Pt wires located at points A-B. [1]

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 35 shows the characteristic conductivity curve obtained by such a system.

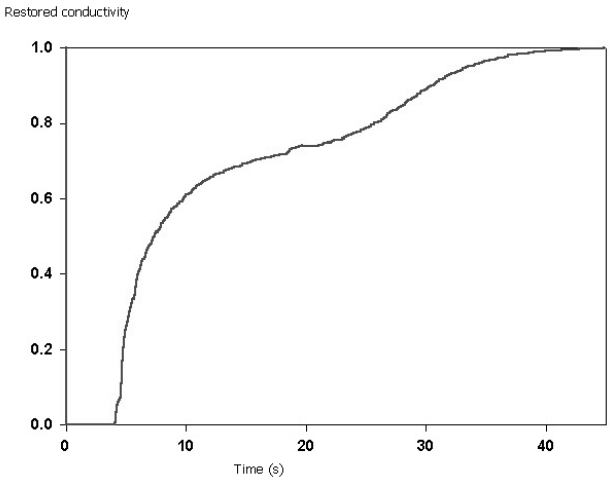


Figure 35 : Characteristic conductivity curve [1]

9.2 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.

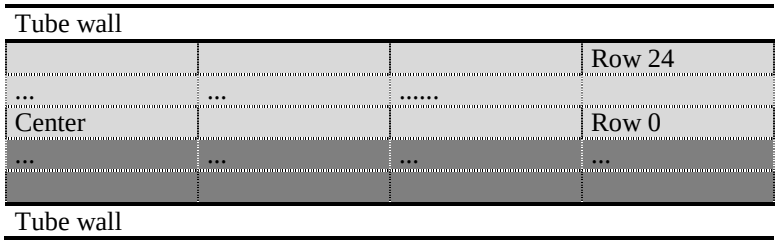


Figure 36 : Correspondence between the cell-space and the actual tube

Figure 36 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 37 : 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 38 : Definition of the FIA coupled cell model

Figure 38 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 39.

```
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 39 : 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. NCD++ 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 40 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 40 : 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 41 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 41 : External coupling of the FIA Cell-DEVS model.

9.3 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 42. 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.

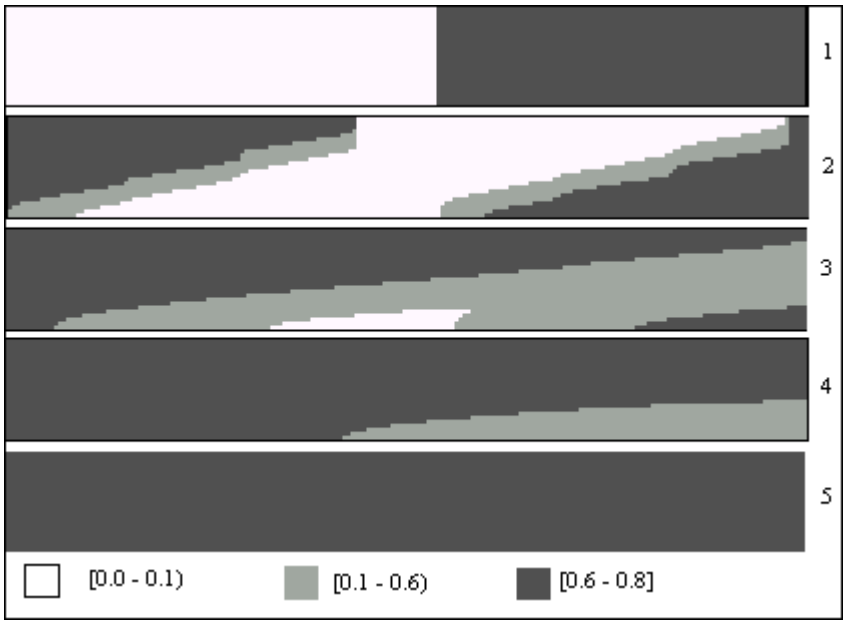


Figure 42 : Different execution stages of the FIA model. (1) At time 0 the sample (white), has been injected. The other half of the tube contains the carrier solution (dark gray). (2,3,4) The convective transport makes the sample disperse faster at the middle of the tube than near the walls. (5) The 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.

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

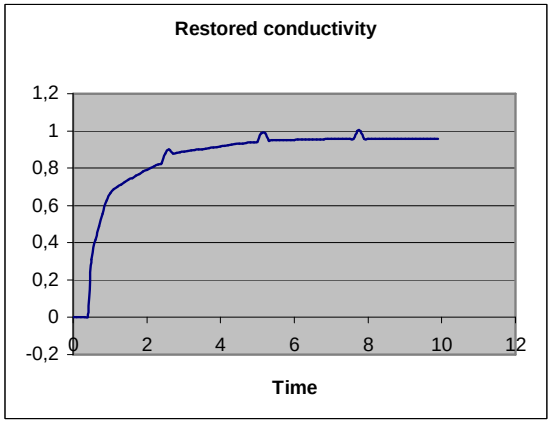


Figure 43 : Conductivity curve obtained

Figure 43 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 to simulate the whole FIA manifold.

10
Conclusions and further developments

An abstract simulator for distributed Parallel DEVS and Cell-DEVS has been presented. Distributed environments have a communications overhead that can be quite significant and that was not taken into account in previous works, which assumed parallel simulation on a shared-memory system. The extension of the Parallel-DEVS abstract simulator here presented keeps to a minimum the number of messages sent across machines. This was possible by assigning each coupled model one *master coordinator* and zero, one or more *slave coordinators.* Messages that have to cross a processor boundary are always sent between *master* and *slave coordinators,* which then forward the received messages to their local dependants.

This mechanism was implemented into Parallel CD++, a tool for running these models on a network of computers. Several tests were done to measure the execution time of a model on different configuration of machines. It was expected that a simulation on a distributed environment would take less time as more machines are used, but this was not always the case. Adding more machines to the simulation does reduce the execution time if there is simultaneous execution and if each simulation cycle takes a significant computing time that will compensate for the overhead due to messages being sent across the network. A model's level of parallelism was measured through a given formula that evaluated the amount of activity on each LP at every simulation cycle. Through this measure, it was shown that those simulations which experienced a reduction in the simulation time as more machines were added had a high level of parallelism.

Finally, a flow injection model using Cell-DEVS was presented. This model is still being developed and Parallel CD++ will be required for a simulation of a full scale scenario which consists of more than 50,000 thousand cells.

There are quite a few topics for further research:

- A new abstract simulator that will allow for out of order execution of events. The current abstract simulator forces all LPs to run at the same virtual time. This constraint may reduce the parallelism. A new abstract simulator may simplify this constraint and allow for better performance. For this new mechanism the Warped TimeWarp kernel will be required. The TimeWarp protocol by itself has a lot of performance improvements options which will affect DEVS and Cell-DEVS simulation on different ways. These should be also studied.

- Parallel CD++ requires the model partition to be defined before the simulation is run. If a partition is not chosen wisely, the load may turn out to be unbalanced among the available machines. A dynamic load balance mechanism might be implemented to allow for run-time balancing of the load.

- Though most of CD++ has been changed to obtain Parallel CD++, the Cell-DEVS rule evaluation mechanism has been left unmodified. These mechanism should now contemplate the new timing constraints presented in Wai00.

11
References

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

[Cho94a] ALEX C. CHOW; BERNARD P. ZEIGLER. Parallel DEVS: A parallel, hierarchical, modular modeling formalism. In *Winter Simulation Conference Proceedings*, Orlando, Florida, 1994. SCS.

[Wai] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modelling and simulation of cell spaces ". In "Discrete Event Modeling & Simulation: Enabling Future Technologies", to be published by Springer-Verlag. 2001.

[Wai00] WAINER, G. “Improved cellular models with parallel Cell-DEVS”. In *Transactions of the SCS.* June 2000.

[Cho94b] ALEX C. CHOW, DOO H. KIM; BERNARD P. ZEIGLER. "Abstract Simulator for the parallel DEVS formalism". *AI, Simulation, and Planning in High Autonomy Systems.* Dec., 1994

[Zei90] BERNARD P. ZEIGLER. *Object Oriented Simulation with Hierarchical, Modular Models.* Academic Press, San Diego, California, 1990.

[Wai99] RODRIGUEZ, D.; WAINER, G. "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Multiconference on Computer Simulation*. 1999.

[Mar97] MARTIN, D.; MCBRAYER, T.; RADHAKRISHNAN, R.; WILSEY, P. "TimeWarp Parallel Discrete Event Simulator". *Technical Report. Computer Architecture Design Laboratory, University of Cincinnati*. December 1997.