NEW TECHNIQUES FOR PARALLEL SIMULATION OF DEVS AND CELL-DEVS MODELS IN CD++

By

Ezequiel Glinsky, B. Sc.

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of

the requirements of the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

Canada

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis

New techniques for parallel simulation

of DEVS and Cell-DEVS models in CD++

Submitted by Ezequiel Glinsky

In partial fulfillment of the requirements for the

Degree of Master of Applied Science

_____

Thesis Supervisor

Dr. Gabriel Wainer

_____

Chair, Department of Systems and Computer Engineering

Dr. Rafik A. Goubran

Carleton University

2004

ABSTRACT

DEVS is a sound formal modeling and simulation (M&S) framework based on generic dynamic system concepts. Cell-DEVS is a formalism for cell-shaped models based on DEVS. This work presents a new simulation technique for execution of DEVS and Cell-DEVS models in distributed environments. The parallel simulator is based on Time Warp, an optimistic synchronization protocol, and developed as a new simulation engine for CD++, a M&S toolkit that implements DEVS and Cell-DEVS theory. The presented technique uses a non-hierarchical approach that simplifies the structure of the simulator and reduces the communication overhead. In order to analyze the performance of our simulator, we introduce a synthetic benchmark to test DEVS-based simulators. The performance analysis shows reasonable overhead in comparison to other simulators. Using a distributed environment, our simulator outperforms other alternatives and achieves considerable speedups.

ACKNOWLEDGMENTS

I want to start by thanking my supervisor, Prof. Gabriel Wainer, for his support and guidance over the last years. Working with him has been a challenging yet rewarding experience.

I am grateful to several people at Carleton University who helped me in many ways, especially to Diane Berezowski, Laura Cohen, and Luc Lalande.

I have had a wonderful time with friends at the Graduate Students' Association. I will always keep those people, those moments, and that organization in my heart. In particular, I want to thank my dream team: Cathy Anstey, Glen Bornais, Robert Johnson, Phil Robinson, and Andrea Rounce.

Thanks to those who became my closest family in Canada. My biggest gratitude goes to Juanca and Leo for our *café mirón* and our Argentinean friendship, and to my favourite Chilean, Loreto. I am very grateful to many friends I met in Ottawa, especially to Abeer, the Martino family, Thierry, José Merseguer and my unforgettable *rumis* Nestor and Covy.

I am thankful to my dearest friends in Argentina: Alberto Siless, Leandro Resnik, Lisandro Icardi, Luciano Tirantte, Martin Dragovetzky, Patricio Donato, and their partners.

I love and admire my parents, Isabel Monzón and Gregorio Glinsky. I carry their love, support and encouragement as a gift. Thanks to my brothers, Adrian and Fernando, who are always with me.

This thesis is dedicated to the love of my life, Solange Epelman.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ACRONYMS

AMS	Automatic Manufacturing System
API	Application Program Interface
CORBA	Common Object Request Broker Architecture
CVDS	Continuous Variable Dynamic Systems
DEDS	Discrete Event Dynamic Systems
DEVS	Discrete Event System Specification
GVT	Global Virtual Time
HLA	High Level Architecture
LP	Logical Process
M&S	Modeling and Simulation
MPI	Message Passing Interface
OMG	Object Management Group
PDES	Parallel Discrete Event Simulation
P-DEVS	Parallel Discrete Event System Specification
UML	Unified Modeling Language

Chapter 1: INTRODUCTION

Modeling and simulation (M&S) methodologies have become essential for understanding, analyzing and developing a wide variety of systems. In the last centuries scientists and engineers have relied on the use of models to describe the properties of the systems under study. Most of these models were defined with mathematical representations, allowing mathematical analysis techniques. Nevertheless, these methods are unsuitable for many complex artificial applications developed in the last 50 years, such as traffic control systems, automated factories, computer architectures, or biomedical devices. These methods are not appropriate for studying various natural systems either, especially when the complexity or the required level of detail is high.

The development of computers has offered alternative methods; models can be executed using computer simulation, allowing users to experiment different conditions under risk-free environments. M&S is also frequently used for training and educational purposes, using models previously developed by experts within an application domain. Nowadays, M&S is a well-developed, well-proven approach to problem solving, which advances steadily as more computing power becomes available at less cost.

The M&S process begins with a problem that needs to be solved or understood. Figure 1 shows the basic entities in M&S and their relationships, as described in [Zei00]:

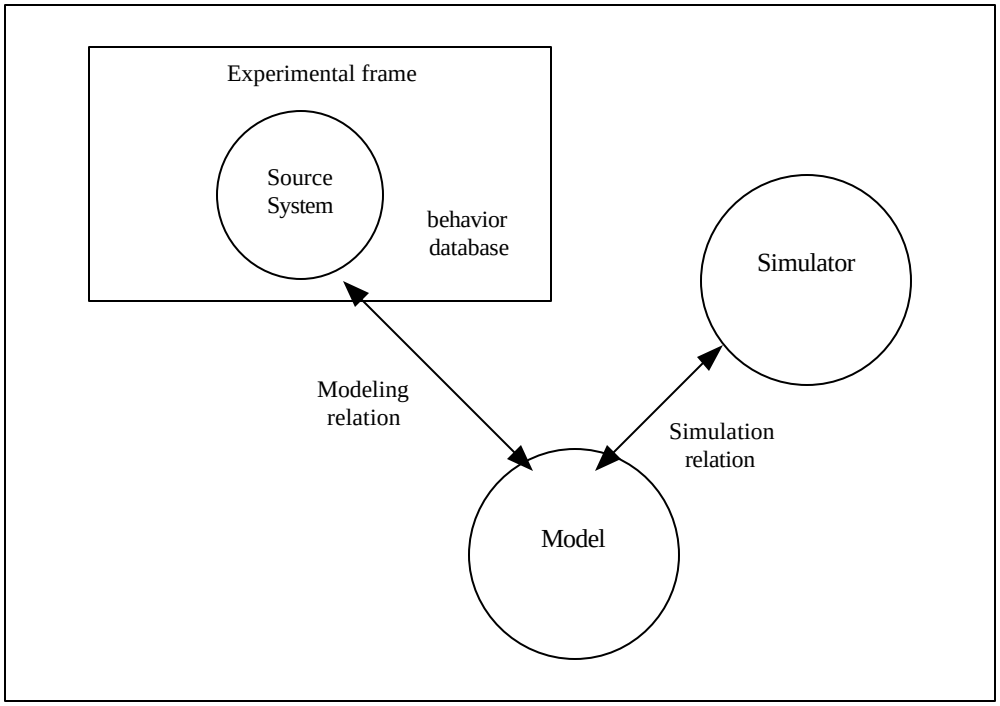


Figure 1: Basic entities in M&S and their relationships [Zei00]

The **source system** is the environment under analysis. The elements observed in the system and the conditions under which the system is observed establish the **experimental frame**. A **model** is an abstract representation of such system that is constructed using the acquired data. Generally, a model includes a set of instructions, rules, or mathematical equations to duplicate the behaviour of the actual system. A **simulator** is an agent capable of executing the model's instructions, and thus generating the model's behavior.

Figure 1 shows the two fundamental relationships that connect the basic entities. The **modeling relation** links the real system and the experimental frame with the model in terms of validity. It is concerned with how well the model behaviour agrees with the system behaviour under the conditions specified in the experimental frame. The

simulator relation links a model and a simulator. This relation deals with how faithfully the simulator executes the instructions of the model.

The separation between model and simulator enables to validate the model and to verify the correctness of the simulator independently [Zei99b]. This distinction results in simulation algorithms whose correctness has been rigorously established separately from the model.

Nowadays, several formalisms coexist and are being used to model and simulate different types of systems. In this work, we focus on the **DEVS** (Discrete Events systems Specification) formalism [Zei76, Zei00], which has been proven to be a universal formalism to represent **DEDS** (Discrete Event Dynamic Systems). DEVS was originally defined in the 1970's as a discrete-event M&S mechanism. DEVS is a sound formal framework based on generic dynamic systems concepts that supports provably correct, efficient, event-based simulation. The framework enables the construction of models in a hierarchical, modular fashion, allowing component reuse and reducing development and testing time. **Parallel DEVS** or **P-DEVS** [Cho94a] is an extension to DEVS that provides a better way to handle simultaneously scheduled events, while keeping all the major properties of the original formalism. Since P-DEVS eliminates serialization constraints existing in the original DEVS formalism, it enables more efficient execution of models in parallel and distributed environments.

The **Timed Cell-DEVS** formalism [Wai98] combines cellular automata [Wol86] with DEVS theory, allowing individual cells to be defined as basic DEVS models and

coupled together to form complete cell spaces. The formalism supports the definition of complex cell behavior with simple constructions.

Numerous tools that implement these formalisms have been used in a variety of industries and areas of expertise, including chemistry, biology, computer architectures, telecommunication networks, decision support systems, military applications, and transportation. **CD++** [Wai02] is a M&S tool that implements DEVS and Cell-DEVS theory. CD++ was revised and extended several times, and it currently supports stand-alone [Rod99], real-time [Gli02a], and conservative parallel simulation [Tro01a]. CD++ has been used to model a variety of applications. The propagation of forest fires has been simulated with CD++ [Ame01] using a Cell-DEVS model. Environmental and vegetation conditions determine spread and intensity of fire. Three main groups of parameters are specified: vegetation type (caloric content, mineral content and density), fuel properties (the type of vegetation is classified according to its size), and environmental parameters (wind speed, humidity and field slope). External factors are taking into consideration for the spread of the fire in the region, such as the influence of rain or the activity of firefighters. The movement of robots in an industrial plant has also been studied with CD++ [Ame01]. Robots follow predefined one-way routes at a given speed with the risk of colliding with other robots, in which case they apply a strategy to continue their way. When a robot reaches its destination, the carried load is delivered and the robot is taken off the floor. In this application, a DEVS component adds new robots to the simulation. CD++ was used to study a watershed using Cell-DEVS [Ame01], based on a model previously defined in [Zei96]. The equations that define the filtration of water through

each layer of the soil were used to specify the rules of the Cell-DEVS model. The simulation allows the study of the accumulation of water on a region after a period of rain. A computer processor was simulated using a DEVS model defined in CD++ [Wai01]. This model includes the specification of components such as memory, registers, and control unit. The reproduction of a marine germ was studied with CD++ [Ame03]. The concentration of bacteria over time was simulated using a Cell-DEVS model. The variation of temperature was simulated using a DEVS model. The behavior of ants also been studied with CD++ [Ame03]. The model analyzes how ants find an existing source of food and carry the food back to the anthill. CD++ has been used to study the movement of crowds in a metro station [Ame03]. The model shows how people try to reach the doors of a railroad car. It also shows the conflict when they collide with people trying to get out from the railroad car. Modeling and simulation of urban traffic has been studied using ATLAS [Dia01]. ATLAS is a specification language to study the flow of vehicles in a city, which gives modelers a simple means to describe intersections, streets, and other constructions, such as railways, traffic lights and parking spaces. The flow of cars and trucks was studied in detail in [Dav00a].

M&S has become a fundamental tool in a wide variety of fields. As a result, many of the simulated systems are becoming more and more sophisticated. As these systems become larger and more complex, the resources provided by a single-processor machine become, in many cases, insufficient to execute those systems. **Parallel and distributed simulation** (PADS) deals with the issues introduced by distributing simulations over multiple processors. **Parallel discrete event simulation** (PDES) studies the execution of

discrete event models in parallel or distributed computers. A PDES simulation advances by the occurrence of events that take place at discrete points in time. Fujimoto identified three major research communities involved in the field of parallel and distributed simulation [Fuj01]. The first group is the high performance computing community, whose work started in the late 1970's and 1980's. This group's main concern was to reduce execution time of applications by using multiple processors. Several synchronization algorithms developed by this community, such as Chandy-Misra-Bryant [Bry77, Cha79] and Time Warp [Jef85], introduced fundamental ideas that are still being applied. The second group is the defense community, mainly interested in integrating separate training simulations to facilitate interoperability and software reuse. The third group is the gaming and Internet community. Their efforts are mostly focused on developing realistic scenarios in distributed environments.

Parallel and distributed simulation can provide four major advantages [Fuj99]:

1. *Enabling execution of simulations that otherwise could not be performed.* Executing a large system after subdividing it in simpler, smaller parts enables shorter execution times. This enables using real-time simulations to support time-critical decision-making processes in cases where a single computer cannot achieve the required performance. Moreover, distributed environments allow the execution of larger, more complex simulations whose memory requirements exceed the resources available in a single computer.

2. *Geographical distribution.* It is possible to distribute the execution at different physical locations, which is particularly interesting for some applications where data or users are not located in a central location.
3. *Integrating simulators based on different platforms.* Simulations can be carried out using different computers, operating systems, and simulators.
4. *Fault tolerance.* When using multiple processors, it becomes possible to increase the tolerance to failures in a simulation; if a node fails, a surviving node may take over and continue the execution.

Synchronization is key when executing applications in parallel and distributed environments. A **logical process** (**LP**) is a basic entity in a simulation. An LP receives and generates timestamped events or messages to communicate with other LPs, which might execute in a different processor or machine. The synchronization mechanism ensures that each LP complies the **local causality constraint**, which requires that events should be processed in their timestamp order.

There are two main classes of algorithms for synchronization. **Conservative algorithms** offer a pessimistic approach. They avoid violating causality constraints at all times during the execution of a simulation. **Optimistic algorithms** allow some violations to happen, but provide a mechanism to detect and recover from these situations. Optimistic algorithms have two main advantages over conservative approaches: (i) they enable greater degrees of parallelism, and (ii) they do not rely on application-specific data to determine events that are safe to process, which is usually the case in conservative approaches.

A different way for improving simulation performance and reducing execution times deals with the structure of the simulator. **Hierarchical simulation** mechanisms incur in greater overheads due to an increased number of exchanged messages that travel up and down the entire structure. **Flat simulation** approaches have been implemented in distributed [Kim00a] and stand-alone [Gli02a, Gli02d] environments, aiming to reduce the overhead by simplifying the structure. It has been shown that flat simulation approaches outperform the hierarchical mechanisms in virtual-time and real-time simulators [Gli02b, Gli02c].

A hierarchical, conservative parallel simulation mechanism has been implemented in CD++ [Tro01a]. Results have shown that parallel simulations outperformed single-processor simulations for both DEVS and Cell-DEVS models in CD++ [Tro01b]. However, since it implements a pessimistic synchronization mechanism, the degree of parallelism and the corresponding speedups are bounded. Moreover, studies showing that hierarchical approaches worsen execution performance encourage the implementation of a flat distributed simulator.

1.1 CONTRIBUTION

This dissertation presents the design and implementation of a new technique for optimistic simulation of Parallel DEVS and Cell-DEVS models in distributed environments. Our simulation methodology is based on the Parallel DEVS abstract simulator [Cho94b] and the Time Warp synchronization mechanism [Jef85]. The Time Warp algorithm allows simulation objects to process events optimistically, assuming

events sent from remote LPs will not cause rollbacks in the future. If a remote LP sends such a message, simulation objects have to rollback optimistically processed events and continue the normal execution of the model from that point.

We introduce two new classes of DEVS processors that carry out the simulation efficiently across multiple machines. The proposed simulation algorithms use a flat simulation approach that eliminates the need for intermediate coordinators. Consequently, it reduces the overhead of message passing, improving the overall performance of the simulation.

The new simulation technique is implemented in the CD++ toolkit, and its efficiency is measured using DEVS and Cell-DEVS models. Moreover, instead of restricting our efforts on testing individual models, we developed DEVStone, a synthetic benchmark to study the performance of DEVS-based simulators. Different factors were considered in order to create such synthetic models, which resemble real world applications. We focused on the main factors that have a significant impact on performance. Our benchmark supports generating models that have different structure, size and behavior. Two parameters define the general structure and size of a DEVStone model, namely depth and width. The depth of the model describes the number of levels in the overall structure. The width of the model specifies the number of components in each intermediate level. DEVStone supports three different types of models. Each model type is characterized by the number and complexity of interconnections between inner components. Finally, models execute different workloads that represent the real world processes to be performed by components of the application.

Using DEVStone, it is possible to assess the performance of a simulator using a large set of models with diverse characteristics. Since the structure and behavior of the models are known, it is also possible to compare the performance of different simulators. In order to obtain meaningful results, we use different types of models focusing on issues that can impact the execution performance. For example, we study the effect of executing models that are predominantly wide (i.e., a large number of models per level) or deep (i.e., a large number of levels in the modeling hierarchy). Using a heterogeneous test set, it is possible to analyze the performance of the simulators under different scenarios.

The accuracy and reliability of DEVStone relies on the automatic generation and execution of a large pool of diverse models, which provides a robust test set. This enables an analysis of performance with relation to the characteristics of a category of models of interest. The scope of this work is the execution of medium to large size models, defined as models composed of a minimum of 30 components. DEVStone can be used to assess the efficiency of DEVS simulation engines for these types of models, and it provides a common metric to compare the results using different tools.

We conduct a performance analysis using DEVStone to study the overhead of the new mechanism. A comparison between its performance and other engines provided by CD++ is provided. Although the overhead associated with synchronization tasks implemented by our simulator can be considerable, it presents good performance results when compared with more simple techniques. Based on these tests, we observed overheads for DEVS models is in the range of 2.5% to 5%. For Cell-DEVS models, we

present a performance analysis on distributed environments using models with different size and partition strategies showing significant execution speedups.

1.2 THESIS ORGANIZATION

This work is organized as follows. Chapter 2 introduces the DEVS and Cell-DEVS formalisms, as well as some general concepts on parallel and distributed simulation with special focus on Time Warp synchronization. A survey on existing DEVS-based tools is presented. We also review the design of the CD++ simulator and provide some examples of DEVS and Cell-DEVS models. Chapter 3 discusses the simulation mechanisms provided by CD++ and other tools. Then, we introduce basic ideas about our new simulation technique. In Chapter 4, we present the design of our flat, optimistic simulator for DEVS and Cell-DEVS. We introduce the new algorithms that carry out distributed simulation, providing sample scenarios to better understand how they work. Chapter 5 discusses the implementation issues related to the distributed CD++ simulator. Chapter 6 introduces a synthetic benchmark for DEVS-based tools. A performance analysis of the new simulator that uses DEVStone and other models is presented. Finally, Chapter 7 provides our conclusions and future work.

Chapter 2: DISCRETE EVENT MODELING AND SIMULATION TECHNIQUES

This chapter provides background information about the DEVS and Cell-DEVS formalisms and their extensions. The two main synchronization approaches for distributed simulation are also discussed, focusing on the optimistic alternative chosen for this work. We survey several M&S tools based on DEVS. Then we review the design of the CD++ simulator and some examples of models are given. Finally, this chapter presents Warped and MPI, which are used to implement the optimistic CD++ simulator.

2.1 DEVS AND PARALLEL DEVS FORMALISMS

Systems whose variables are discrete and where time advance is continuous are known as DEDS (Discrete Event Dynamic Systems), as opposed to CVDS (Continuous Variable Dynamic Systems) which, in general, can be described by differential equations. Simulation mechanisms for DEDS systems assume that changes of state will take place upon the occurrence of an event. Formally, an event is defined as a change of state that occurs at a specific point of time $t_i \in R$.

DEVS (Discrete Events systems Specification) [Zei76, Zei00], a formalism for modeling and simulating DEDS systems, defines a way to specify systems whose states change either upon the reception of an input event or due to the expiration of a time delay. It allows hierarchical decomposition of the model by defining a way to couple existing DEVS models.

A real system modeled using DEVS can be described as a composition of *atomic* and *coupled* components. An *atomic* model is defined by:

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, l, ta \rangle$$

where

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

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

S is the set of sequential states;

$\delta_{ext}: Q \times X \rightarrow S$ is the external state transition function;

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

$\delta_{int}: S \rightarrow S$ is the internal state transition function;

$\lambda: S \rightarrow Y$ is the output function;

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

A DEVS model is in a state $s \in S$ at any given time. In the absence of external events, it remains in that state for a lifetime defined by $ta(s)$. A transition that occurs due to the consumption of time indicated by $ta(s)$ is called an internal transition. When $ta(s)$ time expires, the system outputs the value $\lambda(s)$ and then changes to a new state given by $\delta_{int}(s)$. On the other hand, an external transition occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by $\delta_{ext}(s, e, x)$ where s is the current state, e is the time elapsed since the last transition and $x \in X$ is the external event that has been received.

The time advance function can take any real value between 0 and ∞. A state for which *ta(s) = 0* is called a transient state. In contrast, if the *ta(s) = ¥* then *s* is said to be a passive state, in which the system will remain perpetually unless an external event is received.

The following figure shows the description of states and variables in DEVS models:

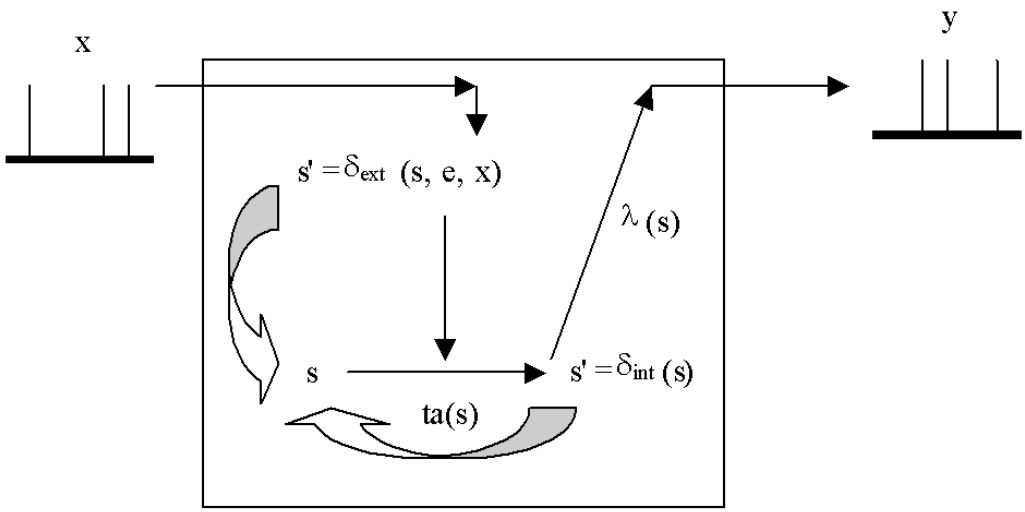


Figure 2: DEVS semantics

A DEVS *coupled model* is composed of several atomic or coupled submodels. It is formally defined by:

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

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, and the following constraints apply to the components, which are also DEVS models:

for each $d \in D$

$M_d = (X_d, Y_d, S, d_{ext}, d_{int}, d_{con}, l, ta)$ is a DEVS basic structure,

where

$X_d = \{(p,v) | p \in IPorts, v \in X_p\}$,

$Y_d = \{(p,v) | p \in OPorts, v \in Y_p\}$, and the couplings are subject to the following conditions:

external input couplings (EIC) connect external inputs to component inputs, $EIC \subseteq \{((N, ip_N), (d, ip_d)) | ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$

external output couplings (EOC) connect component outputs to external outputs, $EOC \subseteq \{((d, op_d), (N, op_N)) | op_N \in OPorts, d \in D, op_d \in OPorts_d\}$

internal couplings (IC) connect component outputs to component inputs, $IC \subseteq \{((a, op_a), (b, ip_b)) | a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$

Direct feedback loops are not allowed, i.e., no output port of a component may be connected to an input port of the same component. Formally,

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

The values sent from a source port must be within the range of accepted values of a destination port (range inclusion constraint). Formally,

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

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

$\forall$ *((a, op$_a$), (b, ip$_b$)) Î IC : Y$_{opa}$ Í X$_{ipb}$*.

select is the tie-breaker function, where *select*: subset of D $\rightarrow$ D, such that for any non-empty subset E, *select* (E) $\in$ E.

A coupled model groups several DEVS into a compound model that can be regarded, due to the *closure property*, as a new DEVS model. The closure property guarantees that the coupling of several class instances results in a system of the same class [Zei00]. This property allows hierarchical model construction.

In addition, each coupled model has its own input and output events, as defined by the X and Y sets. When external events are received, the coupled model has to redirect the inputs to one or more components. Similarly, when a component produces an output, it may have to map it as an input to another component, or as an output of the coupled model itself. Mapping between ports is defined by the *Z* function.

Multiple components can be scheduled for an internal transition at the same time in a coupled component, and ambiguity may arise. If the first component to execute its internal transition produces an output that maps to an external event for another component that is already scheduled for an internal transition, then it is not clear which transition this second component should execute first. Two alternatives exist: to execute the external transition first with *e = ta(s)* and then the internal transition, or else to execute the internal transition first followed by the external transition with *e = 0*. By the *select* function, the DEVS formalism enables a simple way to solve this ambiguity. The function defines an order over the components so that only one component of the group

of imminent models is allowed to have $e = 0$. The other imminent models are divided in two groups: those that receive an external output from this model, and the rest. The former will execute their external transition functions with $e = ta(s)$, the latter will be imminent during the next simulation cycle which may require again the use of the *select* function to decide which model will execute first. This strategy for tiebreaking is rigid and, in addition, it introduces serialization in the execution of components. The serialization introduced by this approach becomes visible when the *select* function has to be used to determine the priority in which the components have to be executed. For example, the *select* function is used to determine which atomic component has priority over the rest to execute its internal transition function when many interconnected atomic models are imminent.

Parallel DEVS or **P-DEVS** [Cho94a] is an extension to DEVS that provides a more flexible way of dealing with these ambiguities. Atomic models provide an additional confluent function to specify collision behavior for events that might be scheduled simultaneously. Since serialization constraints existing in the original DEVS formalism are now eliminated, P-DEVS permits increased degrees of parallelism that can be exploited in parallel and distributed environments. Consequently, Parallel DEVS was the formalism chosen as the foundation for this work.

P-DEVS models are described very much like DEVS models. An atomic Parallel DEVS model is defined as:

$$M = \langle X_M , Y_M , S, d_{ext} , d_{int}, d_{con}, l , ta \rangle$$

where

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

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

S is the set of sequential states;

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

$\delta_{int}: S \rightarrow S$ is the internal state transition function;

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

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

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

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

There are two main differences between a basic DEVS and a basic Parallel DEVS model. First, the external transition function uses a bag of events instead of a single event. This allows multiple events to be processed simultaneously. Since external events received by the component are added to a bag, X_M^b, external transition functions can combine the functionality of a number of external transitions into a single one. Second, the model specification includes a confluent transition function (δ_{con}). When a collision between the internal and external functions occurs, the confluent function determines the new state of the model.

The semantics of P-DEVS are similar to those of DEVS. A basic model is in a state *s* at any given time. In the absence of external events, the model remains in that state for a lifetime period defined by ta(*s*). When that time expires, an internal transition takes place; the system outputs the value $l(s)$ and then it changes to the state specified by

$d_{int}(s)$. If one or more external events $E = \{x_1 .. x_n / x \in X_M\}$ occurs before ta(s) expires, i.e., while the system is in total state (s, e) with $e <$ ta(s), the new state will be given by the model's external transition function, $d_{ext}(s,e,E)$. P-DEVS allows a better way to deal with collisions. External and internal transitions are in conflict when external events E are received when e = ta(s). In such cases, the new state of the model can be given by $d_{ext}(d_{int}(s),e,E)$ or $d_{int}(d_{ext}(s,e,E))$. Hence, modelers have a flexible way of indicating the appropriate behavior for each model in the confluent function (d_{con}), which is triggered in case of collisions.

In P-DEVS, coupled models are defined as in DEVS without the need for a *select* function. Formally, a coupled model is defined as:

$$\text{CM} = <X, Y, D, \{M_d \mid d \hat{I} D\}, EIC, EOC, IC>$$

The definitions for the set of input and output events (X and Y), components (D and M_d), and couplings (EIC, EOC, and IC) follow the specifications of DEVS coupled models presented earlier in this chapter.

If multiple components in a coupled model are imminent, all their outputs are first collected and mapped to their influencees. Then, the corresponding transition function is executed for every model.

2.2 MODELING CELL SPACES

Different formalisms have been used to capture the behavior of systems that can be represented as cell spaces. Examples of such systems can be found in many fields, from chemistry to engineering, from physics to social sciences. Cellular Automata

[Wol86] is a well-known formalism that describes this type of systems. A cellular automaton is an infinite regular *n*-dimensional lattice whose cells can take one finite value. States in the lattice are updated according to a local rule in a simultaneous, synchronous way. The cell states change in discrete time steps using a local transition function that considers the current state of the cell and a finite set of nearby cells (called the neighborhood of the cell).

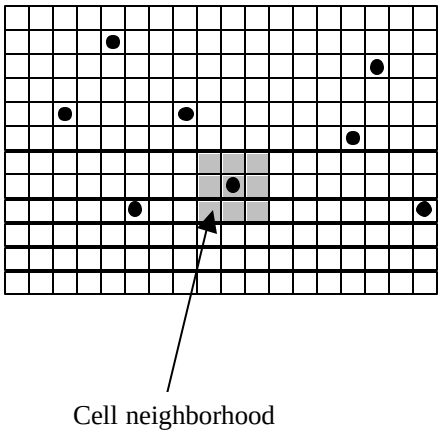


Figure 3: Sketch of a cellular automaton [Wai00]

2.2.1 The Timed Cell-DEVS formalism

The Timed Cell-DEVS formalism [Wai98] uses the DEVS paradigm to define a cell space where each cell is defined as a DEVS atomic model. As a result, it is possible to build discrete event cell spaces improving their definition by making the timing specification more expressive. A Cell-DEVS atomic model is defined in [Wai98] as:

TDC = < X, Y, I, S, ? , N, d, δ_{int}, δ_{ext}, t , ? , 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;

T 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;

t is the local computation function;

? is the output function; and

D is the state 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 *t*. A delay function is associated with each cell, deferring the output of the new state to the neighbor cells. This activation of the local computation is carried by the δ_{ext} function.

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

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

where

X_{list} is the input coupling list;

Y_{list} 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 its neighborhood. Nevertheless, 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* (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. The *select* function serves the same purpose as in the original DEVS models: to tiebreak among imminent components.

The use of the *select* function in Cell-DEVS introduces similar problems to those introduced by DEVS, namely lack of parallelism exploitation and possible inconsistency

with the real system. In addition, the timed Cell-DEVS has another restriction: only one input can arrive from each input port. Such restriction disallows zero-delay transitions and external DEVS models sending two simultaneous events to the same cell [Wai00]. Forbidding zero-delay transitions and the limitation of only one event per external model is very restrictive, and led to an extension of the formalism.

Parallel Cell-DEVS is a revision of the Cell-DEVS formalism that eliminates such restrictions [Wai00]. The author shows two important properties: i) Parallel Cell-DEVS models are equivalent to parallel DEVS models, and ii) closure under coupling for parallel Cell-DEVS models also holds, i.e., a coupled parallel Cell-DEVS model is equivalent to a basic parallel Cell-DEVS model. An implementation of Parallel Cell-DEVS was presented in [Tro03].

2.3 DEVS-BASED TOOLKITS FOR M&S

Several tools have been implemented based on DEVS theory and its extensions, reflecting the level of interest from the community. Some of the existing DEVS M&S toolkits are listed next.

- ADEVS [Nut04] provides a C++ library based on DEVS, which developers can use to build their own models, and supports integration with other simulation environments.
- DEVS-C++ [Zei96] is a DEVS-based modeling and simulation environment written in C++, which implements parallel execution and supports large-scale systems.

- DEVS-Scheme [Zei93] is a knowledge-based environment for modeling and simulation based on the DEVS formalism, supporting real-time simulation.
- DEVS/Grid [Seo04], a JAVA-based simulator for Grid computing infrastructures, was developed focusing on performance and scalability. It supports cost-based model partitioning, remote simulator activation, and dynamic coupling restructuring.
- DEVS/HLA [Zei99a] is based on the High Level Architecture (HLA) [HLA00]. It was used to demonstrate how an HLA-compliant DEVS environment could improve the performance of large-scale distributed modeling and simulation.
- DEVSCluster [Kim00b, Kim04] is a multi-threaded, CORBA-based simulator for DEVS models that supports simulation in heterogeneous network environments.
- DEVSJAVA [Sar98] is a DEVS-based modeling and simulation environment written in Java. It provides classes for the users to implement their own DEVS models.
- DEVSim++ [Kim94] is an object-oriented software to simulate DEVS models, which was implemented in C++. The tool defines basic classes that can be extended by users to define their own atomic and coupled DEVS components.
- GALATEA [Dav00b] is a simulation platform that offers a language to model multi-agent systems using an object-oriented architecture. The tool describes a real system as a set ofinteracting agents.
- JAMES [Him04] implements DEVS theory to model and simulate agent systems. The toolkit supports software-in-the-loop simulation to test agents in virtual environments.

- JDEVS [Fil02a] is a DEVS modeling and simulation environment written in Java. It allows general purpose, component-based, object-oriented, visual simulation of models.
- PyDEVS uses the ATOM3 tool [Del02] to construct DEVS models and to create the code to be executed. Models are represented as a state graph used to generate Python code and then interpreted by PyDEVS.
- SimBeams [Pra99] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, result output analysis and visualization using DEVS.

The majority of the existing toolkits support stand-alone simulation. Some of them, such as DEVS-C++, DEVS/HLA, DEVSCluster, D-DEVSim++ [Kim96] (an extension to DEVSim++), and DEVSJAVA allow distributed execution of DEVS models. The middleware technology that enables parallel and distributed simulation varies from tool to tool. Some of these technologies are:

- CORBA (Common Object Request Broker Architecture) [OMG02], an open standard promulgated by the Object Management Group (OMG),
- HLA (High Level Architecture) [HLA00], a standard specifically designed for distributed simulations, and
- MPI [Don96], a message passing interface standard designed for high performance communication on parallel and distributed environments.

Some of the approaches exploit the specific parallelism existing in DEVS by implementing a pessimistic approach. In such cases, a unique global scheduler is in

charge of synchronizing all nodes; only events with identical timestamp can be processed. As a result, the global scheduler often becomes a bottleneck that prevents achieving higher degrees of parallelism and speedups in a simulation [Kim96]. On the other hand, optimistic approaches give nodes more freedom to process events. In such cases, causality errors can occur but a mechanism to detect and recover from them has to be incorporated. Some efforts in optimistic simulation of DEVS models are summarized next.

DEVS-Ada/Tw [Chr90] was the first attempt to combine DEVS and Time Warp over a multiprocessor environment. However, the implementation imposes two important constraints. First, all models mapped in the same processor are treated as an indivisible logical process. In case of a rollback, the associated cost can be considerable because all the information of the LP has to be restored. Second, models can be divided only at the top level of the hierarchy, imposing a major restriction on users when determining partition boundaries. The second constraint makes the approach inflexible in terms of partition strategies, as it is not possible to divide a model at lower levels of its hierarchy. For example, a system composed by two coupled models can only be partitioned in two processors (one machine running each coupled model), regardless of the internal structure of its coupled models.

The Distributed Optimistic Hierarchical Simulation (DOHS) scheme combines DEVS and Time Warp, implemented in D-DEVSim++ [Kim96]. This alternative presents a more general approach for distributed optimistic execution of DEVS models, while addressing the two major restrictions introduced by DEVS-Ada/TW: DOHS rollback

mechanism allows simulation objects to be rolled back individually, and it supports model partition at any level of the hierarchy.

DEVSCluster [Kim00b, Kim04] is an object-oriented, multi-threaded, distributed simulator that implements a combination of Time Warp and DEVS simulation based on the ideas presented in [Kim96]. However, instead of using the classic message passing approach, DEVSCluster uses CORBA-based method invocation for advancing the simulation. In [Kim04], the authors present a non-hierarchical approach for more efficient distributed simulation.

A risk-free optimistic synchronization mechanism is proposed in [Zei97b], focused on applications that interact with geographically distributed real-world components. In this approach, only safe outputs are sent (avoiding propagation of rollbacks to remote processors so that rollbacks can always be kept local). This mechanism is well suited for shared memory multiprocessor platforms, but has limitations in distributed heterogeneous architectures.

DEVS/P2P [Che04] is an implementation of a distributed DEVS simulator over a layered peer-to-peer network system. The proposed algorithm does not require a coordinator for scheduling purposes; simulators solve synchronization issues by themselves following a decentralized mechanism. Nodes use peer discovery functions to find the location of remote resources.

2.4 THE CD++ TOOLKIT

CD++ [Rod99, Wai02, Tro03] is a M&S toolkit that implements the original and Parallel DEVS and Cell-DEVS formalisms. The tool was built as a hierarchy of classes in C++, where each class corresponds to a simulation entity using the basic concepts defined in [Zei76, Zei00].

There are two basic abstract classes: *Model* and *Processor*. The former is used to represent the behavior of the atomic and coupled models, while the latter implements the simulation mechanisms. Figure 4 shows the CD++ class hierarchy.

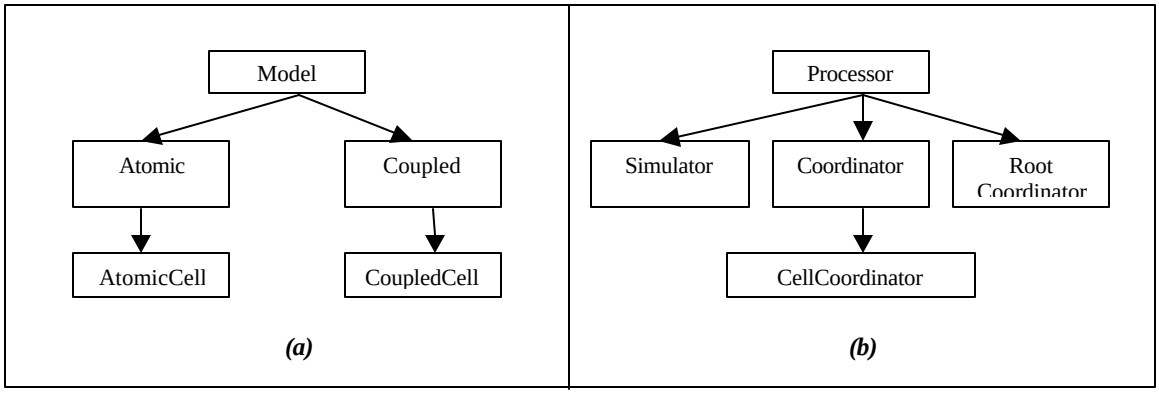


Figure 4: CD++ (a) Model hierarchy, (b) Processor hierarchy

The *Atomic* class implements the behavior of an atomic component. The *Coupled* class implements the mechanisms of a coupled model. For cellular models, special atomic models are used to represent the cells. To do so, *AtomicCell* and *CoupledCell* are defined as subclasses of *Atomic* and *Coupled* respectively. *AtomicCell class* extends the behavior of the atomic models, to define the functionality of the cell space. In contrast, *CoupledCell* handles a group of atomic cells.

A *simulator* object manages an associated atomic object, handling the execution of its δ_{int}, δ_{ext}, δ_{con} and λ(s) functions. A *coordinator* object manages an associated coupled object. Only one *root coordinator* exists in a simulation. It manages global aspects of the simulation. It is involved with the topmost-coupled component, which has the highest level in the model hierarchy. Moreover, the *root coordinator* maintains the global time, and it starts and stops the simulation process. Lastly, it receives the output results that must be sent to the environment.

The simulation process is message driven; processors exchange messages to advance the execution of the model. Each message contains information to identify the *sender* and the *receiver*. A *time-stamp* for the message and an associated *value* are also included in the packet. Two main categories of messages exist: synchronization and content messages. These categories consist of several types of messages.

Synchronization messages:

@	*Collect message*
*	*Internal message*
done	*Done message*

Content messages:

q	*External message*
y	*Output message*

Processors have internal variables to keep the time of the simulation:

t_L	*Time of last transition*

t_N *Time of next transition*

and a *bag* to store external messages.

The tool provides a specification language that allows describing coupling of models, initial values and external input events. Atomic models are developed under C++, which provides a great flexibility and computing power to the modeler. Each new atomic model must inherit from the *Atomic* class in order to extend their basic behavior.

New atomic models are written in C++ and have to be derived from the class *Atomic*. The methods that determine the behavior of an atomic model are:

- *initFunction*, which is executed when the simulation starts, and usually initializes the model variables,
- *externalFunction*, which is executed when an external event is received,
- *internalFunction*, which is executed when an internal transition is scheduled, and
- *outputFunction*, which generates the output of the model and is executed before the internal transition function.

CD++ provides functions that can be used from the atomic models, including:

- *holdIn(state, time)*. It is used to specify that the model must remain in a state for the specified time.
- *passivate()*. When this function is called, the model enters in passive mode (i.e., ta = ∞), and only external events can change its state.
- *sendOutput(p, v).* This function sends an output message with a value of *v* through the output port *p*.

- *state()*. It returns the current state of the model.

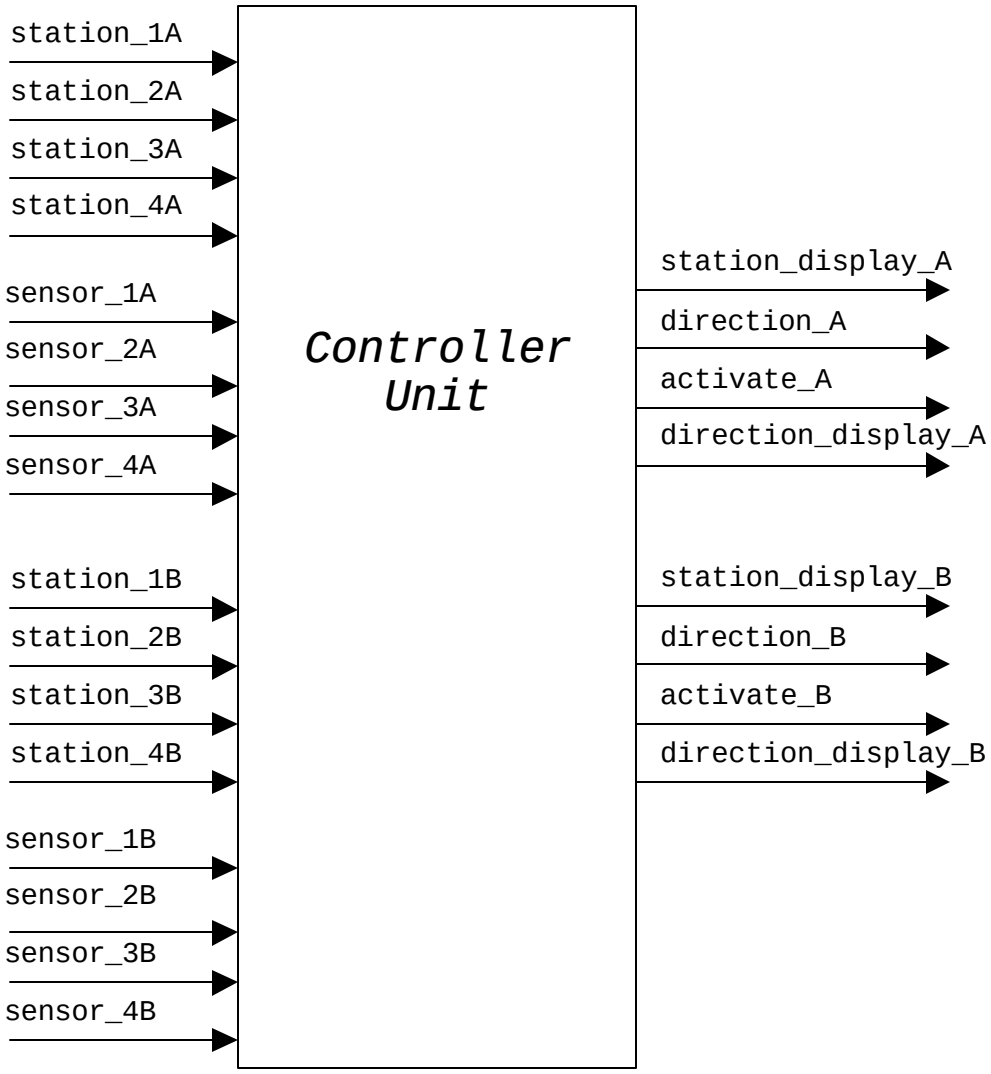


Figure 5: Diagram of atomic model Controller Unit

Figure 5 shows the scheme of an atomic model. This is a controller unit used in an automated manufacturing system (AMS) built with CD++ [Gli04]. An AMS is formed by dedicated stations that perform tasks on products being assembled, and conveyors that

transport the products to/from those workstations. The controller is connected to other components: sensors, a scheduler, conveyor belts, and a digital display. Input ports (e.g., station_1A and sensor_2A, which are connected with the scheduler) and output ports (e.g., station_display_A, which is connected to the digital display, and direction_A, which is connected with the engine of the conveyor belt) allow the controller unit to communicate with those components.

Figure 6 and Figure 7 show the specification of the controller unit in CD++. The constructor of the class, *ControllerUnit::ControllerUnit*, creates the input and output ports of the model. The initialization function, *initFunction*, initializes some of the controller's variables (e.g., current station, next required station, activation of sensors, direction of the conveyor belt) for both production lines (A and B). The external transition function specifies the behavior of the controller unit upon the reception of events from the sensors and the scheduler. For example, upon the activation of a sensor, the conveyor belt A has to be stopped when the requested station is reached. The method *holdIn()* is called to trigger an internal transition function after the time indicated as a parameter (in this case, since the specified time is zero, then the internal function is executed immediately).

```
ControllerUnit::ControllerUnit
( const string &name ) : Atomic( name ),
station_1A( addInputPort( "station_1A" ) ),
station_2A( addInputPort( "station_2A" ) ),
station_3A( addInputPort( "station_3A" ) ),
...
sensor_3B( addInputPort( "sensor_3B" ) ),
sensor_4B( addInputPort( "sensor_4B" ) ),

station_display_A( addOutputPort( "station_display_A" ) ),
...
direction_display_B( addOutputPort( "direction_display_B" ) ),
{ }

Model &ControllerUnit::initFunction()
{
  req_station_A = 1;
  curr_station_A = 1;
  sensors_enabled_A = 1;
  direction_A = 0;
  ...
  req_station_B = 1;
  curr_station_B = 1;
  sensors_enabled_B = 1;
  ...
}

Model &ControllerUnit::externalFunction
( const ExternalMessage &msg )
{
  if (sensors_enabled_A)
  {
    if( msg.port() == sensor_1A )
    {
      if (req_station_A == 1)
      {
        stop_engine_A = 1;
        ...
        holdIn( active, VTime::Zero );
      }
    if( msg.port() == sensor_2A )
    ...
    if( msg.port() == sensor_3A )
    ...
  ...
  }
}
```

Figure 6: Specification of atomic model Controller Unit in CD++ (part 1)

Figure 7 contains the internal and output functions for the controller unit. The output function is executed before the internal transition function. Following the same example, when the engine of the conveyor belt A has to be stopped, a value of 0 is sent via the port *activate_A* using the method *sendOutput*. The internal transition function enables and disables the sensors depending on the values of the current and requested stations for each production line and it passivates the model (i.e., sets the next internal transition time to infinity).

```
Model &ControllerUnit::internalFunction
( const InternalMessage &msg )
{
  if ( !sensors_enabled_A && (req_station_A==cur_station_A) )
  {
    sensors_enabled_A = 1;
  }
  ...
  passivate();
  return *this ;
}

Model &ControllerUnit::outputFunction
( const InternalMessage &msg )
{
...
  if (stop_engine_A == 1)
    sendOutput( msg.time(), activate_A, 0) ;
  if (stop_engine_B == 1)
    sendOutput( msg.time(), activate_B, 0) ;
...
}
```

Figure 7: Specification of atomic model Controller Unit in CD++ (part 2)

CD++ allows users to combine multiple basic models (i.e., atomic or coupled) into a coupled model using a specification language that follows DEVS definitions. Using this specification language, it is possible to define external input couplings, external output couplings and internal couplings, and components that form the model.

Let us continue with the previous example of the automated factory. The entire AMS, formed by a scheduler, a controller unit, a display controller, and two conveyor belts, can be seen as a new coupled model. This coupled model is composed of atomic (e.g., controller unit) and coupled components (e.g., conveyor belt), as outlined in Figure 8.

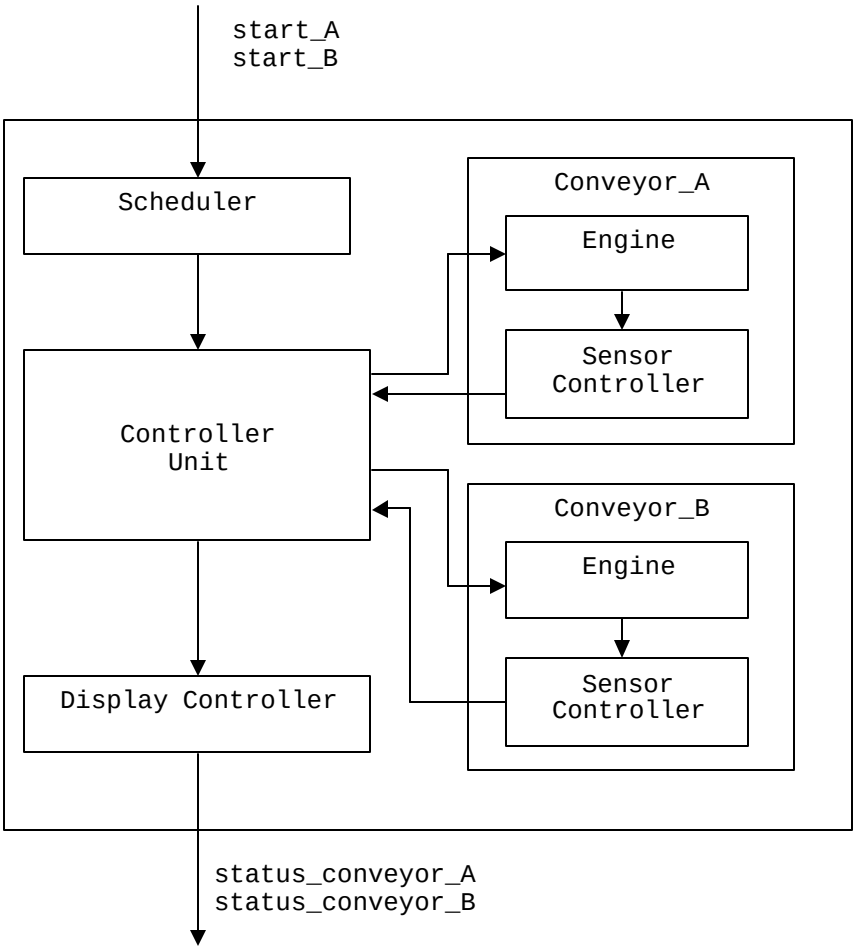


Figure 8: Diagram of the coupled model AMS

Figure 9 shows the specification of the AMS coupled model in CD++.

```
components:  conveyor_A      conveyor_B         scheduler@Scheduler
components   cu@CU           dis@Display
in   : start_A start_B
out  : status_conveyor_A
out  : status_conveyor_B
link : start_A    start_A@scheduler
link : start_B    start_B@scheduler
...
link : sensor_1@conveyor_A   sensor_1@cu
link : sensor_2@conveyor_A   sensor_2@cu
...
link : dir_display_A@cu    dir_display_A@dis
link : status_conv_A@cu    status_conv_A@dis
link : dir_display_B@cu    dir_display_B@dis
link : status_conv_B@cu    status_conv_B@dis
...
[conveyor_A]
components:  sb@SensorController   eng@Engine
in   : activate  direction
out  : sensor_1 sensor_2 sensor_3 sensor_4
link : activate    activate@eng
link : direction   direction@eng
link : sensor_1@sb sensor_1
...
link : current_pos@eng sensor_triggered@sb
...
[conveyor_B]
components:  sb@SensorController   eng@Engine
...
```

Figure 9: Specification of coupled model AMS in CD++

The components for the top model follow the architecture shown in Figure 8. Here, *conveyor_A* and *conveyor_B* are coupled components, whereas *cu*, *scheduler*, and *dis* are atomic. The top model input ports, *start_A* and *start_B*, are used to trigger the production cycle for lines A and B. The output ports, *status_conveyor_A* and *status_conveyor_B*, provide information about the state of products in each line. The keyword *link* defines connections between components. For example, the *start_A* in the top model port is connected to the *start_A* port in the *scheduler*, and the *sensor_1* port of *conveyor_belt_A* is connected to the port *sensor_1* of the controller unit (*cu*).

Cell-DEVS models are also defined using a built-in specification language. Users specify different parameters of the system such as size of the model, cell neighborhood, type of borders (wrapped or non-wrapped), type of delay (transport or inertial), and the rules that determine the behavior of each cell. Figure 10 shows the specification for the popular "life" game [Gar70] as a Cell-DEVS model in CD++ [Wai02].

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
localtransition : conrad-rule
neighborports : value

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

Figure 10: Specification of Cell-DEVS model life in CD++

This life model is defined as a 20x20 wrapped Cell-DEVS model with transport delays and 3x3 neighborhood. The behavior of each cell is defined by the rules of the model. Rules have the form of *VALUE DELAY {CONDITION}*; when the *CONDITION* is satisfied, the cell state becomes *VALUE* and then it is *DELAY*ed for the specified time. In this case, the survival of a cell that is active (or alive) depends on the number of active cells within its neighborhood. If the number of active cells, determined by

statecount(1,~value), is three or four the cell remains alive (specified by the first rule), otherwise it dies (specified by the second rule). The third rule specifies that an inactive cell becomes active if the number of active cells in its neighborhood is three. In this model, the delay is 100 milliseconds for every rule.

CD++ provides several operations, such as Boolean (AND, OR, NOT, IMP, and EQV), comparison (=, !=, <, >, <=, and >=), and arithmetic, as well as numerous functions, such as trigonometric, rounding, truncation, logarithmic, minimum, and maximum.

Nowadays, CD++ is the only simulation tool that implements Parallel Cell-DEVS, although there are numerous tools that support the execution of cellular automata, for example MJCell [Woj04], Cellsprings [Ell04], Trend [Cho02], SpaSim [Mor02], and JCASim [Fre01]. Some of these tools support parallel execution to reduce simulation time. CD++ enables visualization of Cell-DEVS in 2D and 3D using different shapes and colors to better understand the results of a simulation [Wai03].

The algorithms for simulator, coordinator, and root coordinator implemented in CD++ can be found in [Tro03], and are based on those presented in [Cho94b].

2.5 PARALLEL AND DISTRIBUTED SIMULATION

Parallel discrete event simulation (PDES) is focused on the execution of discrete event simulations in distributed environments. In parallel and distributed simulations, the execution of a system is subdivided in smaller, simpler parts that run on different processors or nodes. Each of these subparts is a sequential simulation, which is

usually referred to as a logical process (LP). A logical process groups one or more simulation objects running in a node.

Simulator objects communicate with each other exchanging timestamped messages or events to advance the simulation. Objects located on different LPs have to traverse the boundaries of the LPs to interact with each other in an activity known as inter-LP communication. Those running on the same LP can also interact with each other, in this case without crossing the boundaries of any LP, by means of intra-LP communication.

For example, let us consider the distributed simulation of a system described earlier: an automated manufacturing system formed by workstations, conveyor belts, and loading and storing subsystems. Products move through the factory using the conveyor belts. Items have to be loaded, workstations must perform actions (e.g., polishing, varnishing, cutting, painting) on them and, lastly, completed items are stored. The simulation of such a system is distributed across two logical processes, as outlined in Figure 11: one to control the workstations and conveyor belts (LP_1) and the other for the loading and storing subsystems (LP_2).

When an item is loaded and ready to be processed, the loading system (running on LP_2) places it on the corresponding conveyor belt (running on LP_1). This event requires inter-LP communication (shown in Figure 11 with a dashed line from *Conveyor Belt* to *Workstation B*); a message has to be sent from LP_2 to LP_1. In contrast, interactions among workstations and conveyor belts require only intra-LP communication (shown in Figure 11 with a solid line). Simulation objects that interact frequently should be placed in the

same LP, since intra-LP communication usually requires less time than inter-LP communication. In contrast, simulation objects that seldom interact should be placed in different machines to take advantage of parallelism [Rao98].

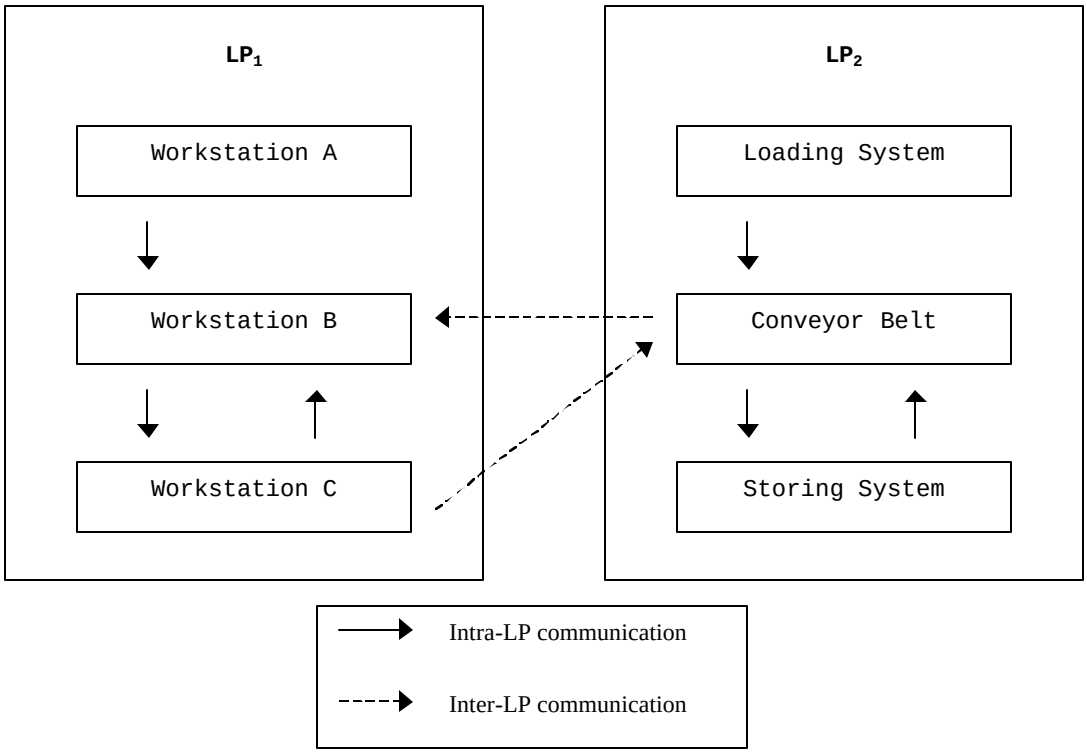


Figure 11: Automated Manufacturing System partitioned in two LPs

Let us extend our example. If one wants to reduce the execution times, it may seem reasonable to concurrently execute events received on different LPs in order to exploit parallelism. A possible scenario of two LPs processing events is shown in Figure 12. Consider two events: E_{200} with timestamp 200 received in LP_2 node and E_{300} with timestamp 300 received in LP_1. Suppose that there are no unprocessed events before 200 (in LP_2) and before 300 (in LP_1). In this situation, it might seem reasonable to process E_{200} and E_{300}. Now, suppose that the execution of E_{200} in LP_2 generates a new event E_{250}

with timestamp 250, which is sent to LP_1 (shown in Figure 12 with a dashed line from LP_2 to LP_1). When LP_1 receives the event E_{250}, it was already processing the event E_{300} with timestamp 300. Although it was received later, the event E_{250} happens before E_{300} and therefore should have been processed first. For example, E_{250} may represent a signal that requires immediate attention and affects the results of processing E_{300}.

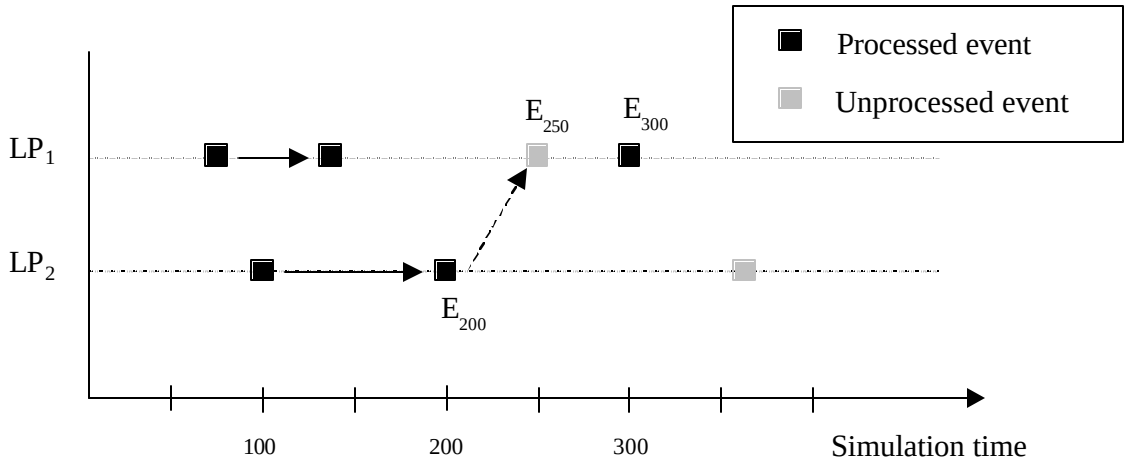


Figure 12: Violation of local causality constraint in a distributed simulation

The local causality constraint [Fuj99] addresses this type of situations:

Local causality constraint. A discrete-event simulation formed by logical processes that interact by exchanging timestamped messages obeys the local causality constraint if and only if each LP processes events and messages in nondecreasing timestamp order.

This brings us to a fundamental issue in parallel and distributed simulations known as synchronization. Simulations in distributed environments rely on

synchronization algorithms that either avoid or deal with local causality constraints. The goal of a synchronization algorithm is to ensure that the distributed simulation of the system yields the same results as the sequential case.

Notice that the synchronization mechanism does not need to guarantee that all events are always processed in their timestamp order, but the final results must coincide with the results obtained by sequential simulation [Fuj99].

There are two major classes of synchronization strategies: conservative and optimistic. Next, we briefly describe each of them with more focus on the latter, which is used in this work.

2.5.1 Conservative Simulation

Conservative approaches, also known as pessimistic approaches, were the first synchronization algorithms proposed for distributed simulations. The general idea behind them is that no local causality errors shall ever happen upon processing an event. In other words, an event is processed in a node if it can be guaranteed that no other event with smaller timestamp will be received in the future. Therefore, a situation like the one illustrated in Figure 12 can never happen.

The Chandy-Misra-Bryant (CMB) algorithm [Bry77, Cha79, Mis86] is a well-known conservative algorithm developed in the late 1970s. Since the requirement introduced by conservative algorithms introduces deadlocks, the original CMB algorithm was extended with *null* messages (which are exchanged among LPs) to deal with this situation. Null messages indicate a lower bound of the subsequent messages that will be

sent by an LP, and allow advancing the simulation, while breaking the deadlock. A well-known problem of the CMB algorithm is that larger numbers of null messages may lead to poor simulation performance: the communication overheads can become considerable high.

Other conservative algorithms based on CMB detect and recover from deadlocks [Cha81] instead of avoiding them. A different algorithm, proposed by Chandy and Sherman, relies on more detailed event-related information and can achieve better performance in many cases [Cha89]. Unfortunately, newer algorithms require more application specific data to exploit greater degrees of parallelism.

Although many conservative algorithms are currently found in real-world applications, they have two main disadvantages [Fuj90]:

i) It is not possible to take advantage of the concurrency available in the application, since they have to adhere to the local causality constraint at all times.

ii) The simulation program has to be specifically designed to exploit concurrency, leading to a complex, tedious design process. In relation to this, small changes in the application may worsen the performance of the simulation in a great way, since changes may affect data used for efficient conservative simulation.

2.5.2 OPTIMISTIC SIMULATION

Instead of avoiding violations to the local causality constraint, like conservative algorithms do, optimistic algorithms allow some causality errors to occur but provide means to recover from them, which in the end leads to correct results.

Optimistic approaches address the two fundamental disadvantages of conservative algorithms:

i) They can exploit higher degrees of concurrency by advancing the simulation optimistically. These approaches assume that causality errors will not arise. If a causality error occurs, the optimistic algorithm has to detect and recover from that situation.

ii) Optimistic algorithms are less dependent on application specific data than conservative approaches, leading to more flexible, transparent applications.

Time Warp [Jef85] is the most popular optimistic synchronization algorithm. Time Warp provides a mechanism that allows LPs to recover from causality errors. An event that is received with a timestamp smaller than one or more of the events that have been already processed in a logical process is known as a *straggler* event, and represents a violation to the local causality constraint. Upon the reception of a straggler event, the LP recovers from the causality error by undoing the effects of the events already processed, in an activity known as *rollback*.

It might be necessary to perform two actions in case of a rollback. First, the state of the object has to be restored to a time smaller or equal to the straggler's timestamp. Second, the process may have sent messages to other LPs in states that are now being

undone. Therefore, it is necessary to inform objects that those events should not be processed (leading to potential rollbacks in those nodes if the events were already processed).

In relation to the first point (the restoration of previous states) Time Warp has a mechanism that periodically stores states of the objects. There are two main techniques in Time Warp that deal with how to rollback state variables:

i) Copy state saving is a strategy that generates a copy of all the state variables within the LP. In case of rollback, it is necessary to retrieve all the variables for the required time, which can be easily accessed. In general, copy state saving is useful for applications that often modify most of the variables.

ii) Incremental state saving, in contrast, saves a copy of individual variables that changed as a result of processing the event. This requires less memory and, potentially, less overhead for storing state variables at each step. However, a rollback requires going back through all the intermediate steps to retrieve all the changes made to state variables. This strategy can be more efficient in scenarios where variables are rarely modified.

In order to deal with messages that should not have been sent, Time Warp uses a mechanism of negative messages or *anti-messages*. Jefferson borrowed the terminology from physics, where matter and anti-matter particles annihilate each other and disappear.

When an object sends a message, a negative message is created and kept. A negative message is a duplicate of the positive (original) message with a flag indicating it is actually an anti-message. In case of a rollback, the LP sends anti-messages to the

corresponding LPs as a means of "unsending" the original one. If the original message has not been processed yet in the receiving node, the anti-message simply annihilates it and both messages are removed from the pending queue. If the original message has already been processed in the receiving LP, the anti-message produces a rollback, which may also generate anti-messages to be sent to other LPs.

Nevertheless, most applications often perform input/output operations that cannot be "undone" or rolled back. Moreover, the Time Warp mechanism as described before has vast requirements of memory for state saving purposes. Both problems are addressed by the concept of *Global Virtual Time* (GVT), a fundamental concept in Time Warp.

The GVT is a lower bound on the time of any future rollback that might occur. Thus, the application has a guarantee that events occurred prior to the GVT will never be rolled back. The consequences of having a GVT are twofold. First, input/output operations with timestamps lower than the GVT can be committed, as it is possible to know that they will never be rolled back. Second, the state information prior to the GVT is no longer needed, since those states will not be restored, and thus memory can be released. The computation of the GVT is fundamental for efficient execution of Time Warp simulations.

Let us focus on one LP to understand how GVT can be computed. The only activity that can trigger a rollback is the reception of a (positive or negative) message in the past of a logical process. Events can be generated only by unprocessed or partially processed events. Consequently, one can compute the GVT as the minimum timestamp among all messages (positive and negative) that are unprocessed or partially processed in

all LPs. A more formal definition of global virtual time is provided by Fujimoto in [Fuj99]:

The Global Virtual Time at wallclock time T (GVT_T) is defined as the minimum time stamp among all unprocessed and partially processed messages and anti-messages in the system at wallclock time T.

There have been modifications to the Time Warp algorithm that try to provide better performance. For example, different mechanisms for state saving were presented in [Ron96, Wes96], and different error-handling mechanisms are discussed in [Nic97].

2.6 THE WARPED TOOL

Warped [Mar97] is a public domain simulation kernel developed at the University of Cincinnati, which provides an implementation of the Time Warp algorithm [Jef85]. We use the services provided by the Warped middleware to implement the optimistic distributed simulator presented in this work. Different Time Warp optimizations are supported in the middleware [Mar96], and the interface for the application developer hides most of the implementation issues. Warped also provides a sequential kernel.

Warped is written in C++ and uses the MPI message passing standard for communication. MPI [MPI95, Don96] is a message passing interface standard designed for high performance communication on parallel and distributed environments. MPI was designed with three main goals: portability, efficiency and functionality.

There are commercial and public domain implementations of MPI. In this work, we use MPICH [Gro96], a freely available implementation that has been ported to

different platforms, including Linux, Unix and Microsoft Windows. Figure 13 shows the layout of how simulation objects and logical processes communicate in Warped [Mar96]. Simulation objects within the same LP exchange messages using direct communication, whereas those running in different LPs use MPI communication services.

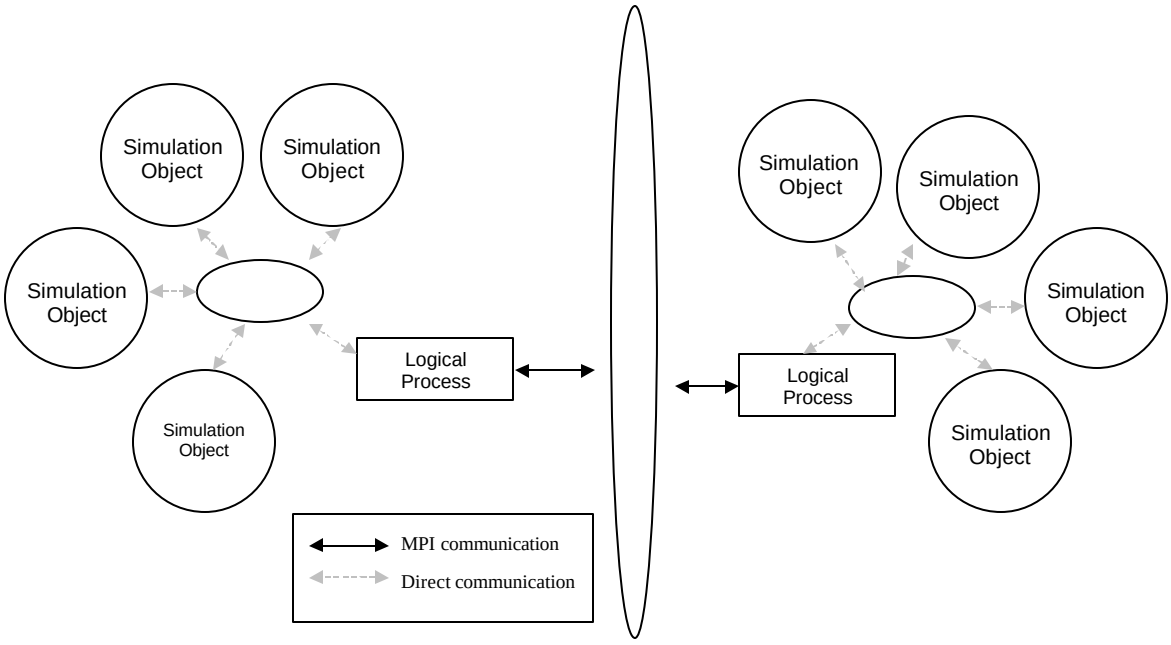


Figure 13: Structure of LPs and simulation objects in Warped [Mar96]

Warped presents an application program interface (API) that allows the definition of simulation objects, their states, and the messages that can be exchanged by those objects. Warped also provides a simple definition of time (which can be redefined by the user) and functions to perform consistent I/O operations.

Figure 14 shows a summary of some classes that form Warped's API.

```
class TimeWarp {
    TimeWarp();
    virtual ~TimeWarp();
    virtual void initialize();
    virtual void finalize();
    virtual void executeProcess() = 0;
    BasicEvent* getEvent();
    ...
}

class BasicState {
    BasicState* copyState( BasicState*);
}

class BasicEvent {
    int size;
    Vtime sendTime;
    Vtime recvTime;
    int sender;
    int dest;
}

class LogicalProcess {
    int getNumObjects();
    LogicalProcess(int, int, int);
    int getTotalNumberOfObjects() const;
    int getLPid();
    void simulate(VTime);
    void calculateGVT();
    ...
}
```

Figure 14: Summary of Warped API [Mar97]

The user can define one or more types of simulation objects, which have to be derived from a basic *TimeWarp* class. The *TimeWarp* class has three main three methods that specify how to initialize the object (*initialize*), what to execute during a simulation cycle (*executeProcess*), and how to finalize its execution (*finalize*). These objects have states (which are also defined by the user and derived from a *BasicState* class) associated with them. The kernel provides two main methods to the user: one to receive events (*getEvent*), and one to send events (*sendEvent*). Different events can be defined by the user deriving them from a basic class, *BasicEvent*. An event always contains information

about its size, the time at which it was sent and received, and the address of the sender and receiver.

The *LogicalProcess* class groups one or more simulation objects sharing a *GVTManager* (in charge of calculating the global virtual time), a *CommManager* (dealing with inter-LP communication), and a *Scheduler* (in charge of scheduling the events received in the queue).

A performance analysis considering different Time Warp optimizations (such as Lower Time Stamp First scheduling, periodic/dynamic checkpointing, and lazy cancellation) implemented in Warped is discussed in [Rad96]. Communication overhead affects the performance of Time Warp simulations; see [Raj98] for a discussion on different alternatives of middleware for Warped communication and an implementation of a new technique.

Chapter 3: ENABLING NEW TECHNIQUES FOR PARALLEL SIMULATION OF DEVS AND CELL-DEVS MODELS

The widespread use of M&S in different application domains is leading to execution of larger and more complex systems, which often translates into more memory and processor requirements. Higher level of detail required by some applications also impacts on the memory and processor requirements. Furthermore, simulation results are frequently expected in short periods of time. For example, consider the creation of virtual worlds with human interaction, where the scenario has to evolve as fast as in real life, or critical on-line decision-making processes where results are needed in real-time [Fuj99]. Nowadays, several M&S tools coexist and try to respond to these needs by providing more efficient simulation mechanisms.

Our work is focused on the design and implementation of a new simulation technique for CD++, a M&S tool for DEVS and Cell-DEVS models. CD++ was originally developed as a stand-alone simulator, and later revisions provided real-time capabilities [Gli02a, Gli02d] and allowed distributed execution of Parallel DEVS and Parallel Cell-DEVS models [Tro01a, Tro03]. Parallel CD++ was the first attempt to reduce simulation time in CD++ by means of distributed execution of models. Distributed simulation with Parallel CD++ has shown speedups in the execution of both DEVS and Cell-DEVS models in comparison to the stand-alone simulator [Tro01b]. Its parallel approach is based on a pessimistic algorithm that exploits the parallelism inherent to the

DEVS formalism. Under that scheme, a single *root coordinator* acts as a global scheduler for every node participating in the simulation. Thus, events with the same timestamp can be processed simultaneously by those nodes.

As explained in Chapter 2, a simulation advances by the exchange of messages between simulators (in charge of atomic models) and coordinators (in charge of coupled models). Parallel CD++ introduced two different types of coordinators (*master* and *slave*) to reduce inter-process communication and, therefore, to alleviate overall communication overheads [Tro01a].

Most existing DEVS tools use a hierarchical simulator creating a one-to-one correspondence between model components and simulation objects. As a result, the simulator structure resembles the structure of the model. Since the simulation advances by exchanging messages between simulation objects, the communication costs associated with this structure can be considerable. Flat simulation mechanisms try to reduce the overhead in communication costs (i.e., try to reduce the number of exchanged messages) by simplifying the underlying simulator structure, while keeping the same model definition and preserving the separation between model and simulator. Studies have shown that flat simulators can outperform hierarchical mechanisms [Kim00a, Gli02a, Gli02d, Kim04]. In some cases, reductions of up to 40% of execution time have been reported [Gli02a, Gli02d]. Although the stand-alone and real-time versions of CD++ support both alternatives, the previous version of parallel CD++ [Tro01a] only supported the hierarchical mechanism.

This work addresses the need for efficient, fast execution of large, complex Parallel DEVS and Cell-DEVS models. We introduce a new technique for optimistic distributed simulation of such models in CD++. The technique combines the Time Warp synchronization mechanism and the Parallel DEVS and Cell-DEVS abstract simulators. In our approach, the hierarchy of the simulation objects is flattened to reduce the communication overheads.

There are two main differences when comparing our new approach with the previous parallel simulation technique available in CD++, namely the use of a non-hierarchical mechanism, and the optimistic protocol for distributed synchronization.

The use of a non-hierarchical mechanism in our work addresses some of the performance issues discussed in [Gli02b, Gli02c] when analyzing different simulation techniques. Those studies have shown that the hierarchical nature of the previous Parallel CD++ technique results in a significant number of messages exchanged in each simulation cycle, which ultimately worsens the performance of the simulator. The work presented in [Tro01a] presents a way to reduce the communication overheads by introducing two specialized DEVS coordinators. However, it has been shown that the communication overhead of the CD++ hierarchical mechanism is, in some cases, still significantly high [Gli02b, Gli02c]. A flat simulation technique, which was implemented in the stand-alone and real-time versions of CD++ [Gli02d], outperformed the hierarchical one for both DEVS and Cell-DEVS models [Gli02b]. Other studies also suggest the use of a flat simulator to reduce communication overheads and to improve performance of DEVS simulation. The original idea of a non-hierarchical DEVS

simulator was presented in [Kim00a], along with results showing its efficiency over hierarchical approaches. Another tool that implemented flat simulation is DEVSCluster [Kim04]. Benchmarking experiments showed that the non-hierarchical structure used by DEVSCluster outperformed the hierarchical mechanism implemented in D-DEVSim++ [Kim04]. When designing DEVS/Grid [Seo04], the authors acknowledged that coordinators become a bottleneck in hierarchical simulation approaches. Considering all these previous results, our work proposes a non-hierarchical simulator for Parallel DEVS and Cell-DEVS models.

Our new simulation technique uses an optimistic synchronization protocol, as opposed to the conservative approach implemented in Parallel CD++ [Tro01a]. The pessimistic approach exploited the specific parallelism existing in Parallel DEVS, but prevents from achieving higher degrees of parallelism because of its conservative nature. Only events that have identical timestamps can be executed simultaneously in the participating nodes. We also introduce the first implementation of an optimistic simulator for Cell-DEVS models and the first that supports flat distributed simulation of such models.

The Cell-DEVS formalism allows higher precision and speedups than traditional cellular automata [Wai00], but previous work has shown that the execution of Cell-DEVS models can be very demanding in terms of memory and computation time [Tro03, Gli02b]. Executing Cell-DEVS models with our simulator aims at reducing execution time and allowing access to more memory space.

The implementation of our optimistic distributed simulator is also important when considering other DEVS tools. In the previous chapter we surveyed many simulators based on the DEVS formalism. A few of them have capabilities for optimistic distributed simulation, and we pointed out limitations imposed by some of them, which we circumvent in this work. DEVS-ADA/Tw [Chr90] introduced the first technique that combined DEVS and Time Warp for distributed optimistic simulation. As we discussed earlier, DEVS-ADA/Tw is not flexible in the way that users can partition models. It is only allowed to partition models at the topmost level of the hierarchy. Additionally, since DEVS-ADA/Tw treats logical processes as indivisible objects, the cost of a rollback can be significantly large. We take a different approach from simulators that implement risk-free synchronization, such as [Zei97b], which have the additional limitation of inapplicability in heterogeneous platforms.

Chapter 4: OPTIMISTIC PDES OF DEVS MODELS

The optimistic distributed simulator for Parallel DEVS and Cell-DEVS introduced in this work is developed as an extension to the original CD++ tool [Rod99, Wai00, Wai02], which supported stand-alone simulation. We follow a layered-architecture design of a previous implementation of a conservative parallel simulator developed in CD++ [Tro01a].

Figure 15 outlines the architecture of our simulator. The topmost layer represents the model, which is executed by simulation algorithms implemented in CD++. The tool is built on top of Warped [Mar97], an object-oriented middleware written in C++ that implements Jefferson's Time Warp synchronization algorithm [Jef85]. Warped, in turn, uses MPICH [Gro96], a freely available implementation of MPI [MPI95, Don96], a message passing standard for high-performance communication on parallel and distributed environments.

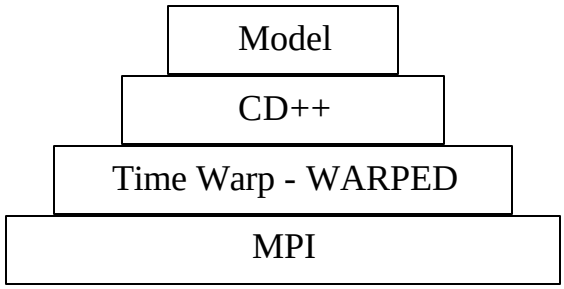


Figure 15: Layered architecture of the CD++ optimistic simulator

In Chapter 2, we have shown the fundamental classes implemented in the CD++ toolkit, which can be divided in two major groups: classes that inherit from the basic

model class, and those that inherit from the basic *processor* class. This reflects the clear distinction between the model and its simulator, a fundamental advantage of the DEVS formalism which allows users to build their models independently from the implementation of the underlying simulator. Since we are interested in the simulation mechanism implemented in CD++, our work takes advantage of this separation of concerns by focusing on the processors' class hierarchy. In contrast, all classes inheriting from *model* remain unchanged from those described in Chapter 2.

Two new classes are introduced, both inheriting from the *processor* class: *flat coordinator* and *node coordinator*. Additionally, we modify two existing classes, *simulator* and *root coordinator*, which also inherit from *processor*. Figure 16 shows the resulting UML class diagram for the processors.

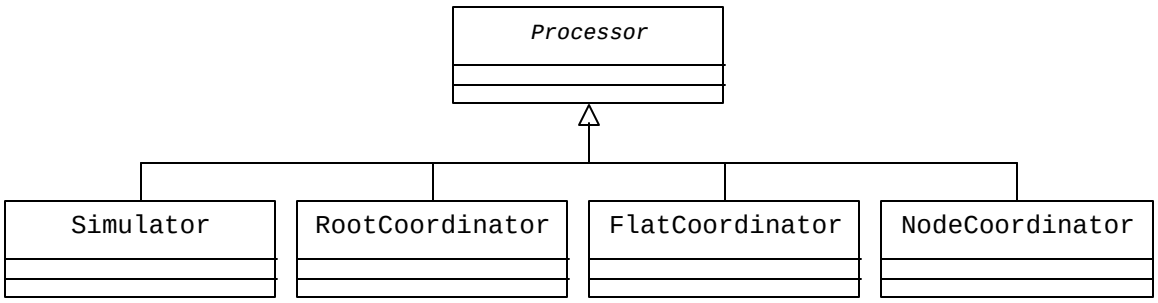


Figure 16: New processors' class hierarchy in CD++

Another important difference between the class diagram shown in Figure 16 and the one presented in Chapter 2 is the absence of two processors (*coordinator* and its descendant *cell coordinator*) in the new hierarchy. This is a result of the new approach implemented for flat simulation that eliminates the need for coordinators, which were present in the hierarchical case. Before describing with more detail the design of our

simulator and the tasks carried out by each DEVS processor, we analyze some factors of the hierarchical CD++ simulator (implemented in the previous version of CD++) and the basic ideas behind the new flat simulation mechanism presented in this work.

4.1 HIERARCHICAL AND FLAT SIMULATION IN CD++

The hierarchical approach implemented in CD++ was introduced in [Rod99, Wai00]. It creates a one-to-one correspondence between the model components and DEVS processors. CD++ produces a processor structure that resembles the structure of the model: a *simulator* object is created for every *atomic* component, and a *coordinator* object is created for every *coupled* component. Analogously, when executing Cell-DEVS models, a *simulator* is created for every *cell*, and a *cell coordinator* is created for every Cell-DEVS model.

Figure 17 shows a sample DEVS model. *Top* is composed by two coupled models (*Coupled #1* and *#2*) and two atomic models (*Atomic #4* and *#5*). *Coupled #1* and *Coupled #2* have three and two atomic models respectively. The arrows represent interconnections between components (e.g., between *Atomic #4* and *#5*), input ports (*in_1* and *in_2*) and output ports (*out*).

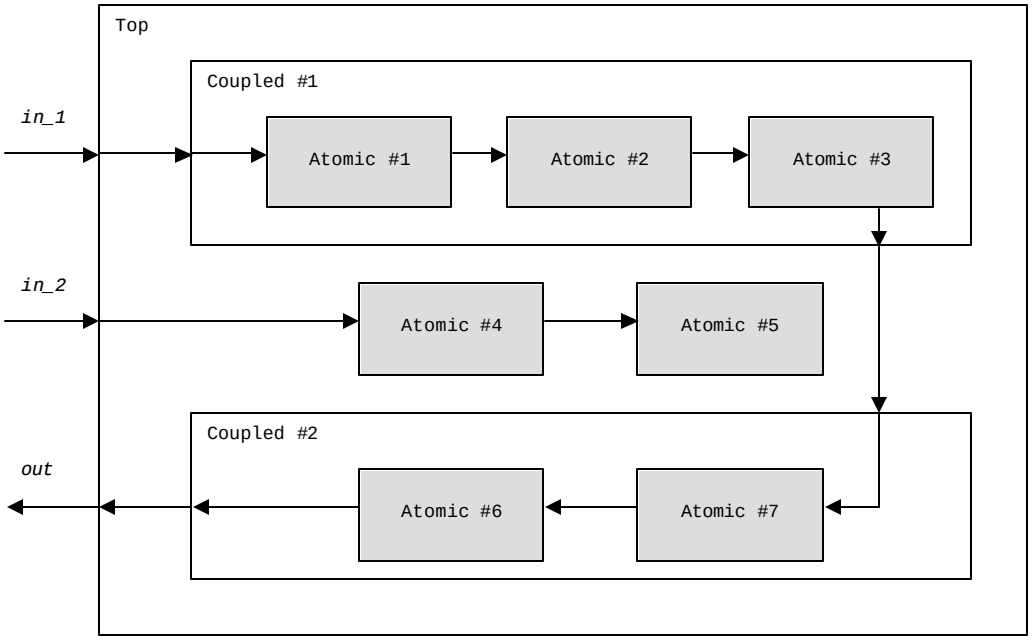


Figure 17: Layout of a sample DEVS model

Figure 18 illustrates the one-to-one correspondence between the model and simulator components created by CD++ when the hierarchical approach is used.

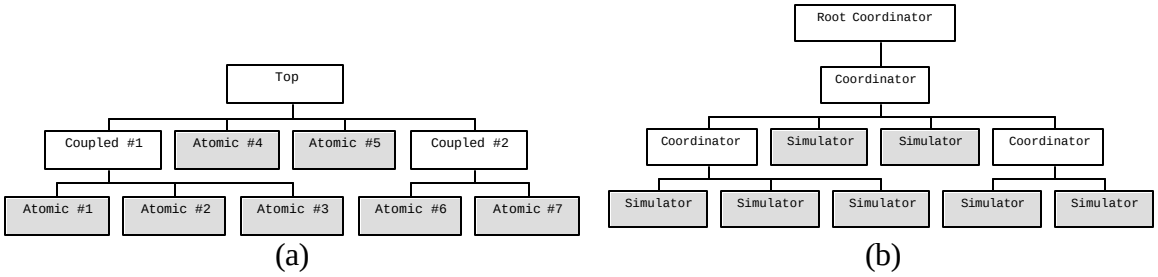


Figure 18: Sample DEVS model in hierarchical CD++
(a) models, and (b) processors

In Figure 18, we see that the processor hierarchy is replicated based on the model hierarchy, using *coordinators* instead of *coupled* components, and *simulators* instead of *atomic* components. A *root coordinator*, in charge of synchronization, time management and I/O operations, is added on top of the processor hierarchy.

The communication costs associated with the hierarchical simulator become visible when analyzing message passing among components. For example, let us examine what happens when an external event is received through port *in_1*. Firstly, the *root coordinator* has to send a message to the *coordinator* in charge of the *Top* model. Secondly, that *coordinator* forwards this message to the *coordinator* in charge of *Coupled #1*, in the lower level of the hierarchy. Thirdly, that message is forwarded again to the actual *simulator* in charge of *Atomic #1*. Then, the simulator executes the model's external transition function, δ_{ext}. A similar phenomenon is observed if *Atomic #3* sends an output through its port connected to *Atomic #7*. In this case, the message has to travel through three intermediate coordinators before reaching the final destination. The number of intermediate coordinators can be arbitrarily high depending on the studied model, and the corresponding overhead can be significantly large.

Based on different studies that show how flat simulation approaches can be more efficient for DEVS and Cell-DEVS simulation [Kim00a, Gli02a, Gli02d, Kim04], we eliminate the need for coordinators using the new set of processors shown in Figure 16. Our flat simulation strategy is based on ideas presented in [Gli02a, Gli02d] for stand-alone simulation, which showed good results in terms of performance [Gli02b, Gli02c]. However, a more sophisticated technique is presented, since it is necessary to deal with distributed execution of models. Figure 19 presents the processor hierarchy for this sample model when the flat simulation is used. At this point, we suppose that only one machine is used for its execution. The more general case with two or more processors will be shown later.

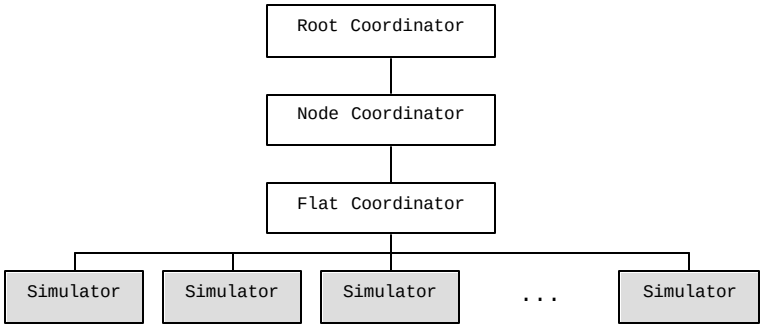


Figure 19: Processor hierarchy using a flat approach

Figure 19 shows that a *root coordinator* is maintained at the top of the hierarchy, handling I/O operations between model and environment, and starting the simulation. However, *root coordinator* is no longer responsible of synchronization and time management tasks. Two new processors, *node coordinator* and *flat coordinator*, are introduced in the hierarchy. *Node coordinator* is now in charge of synchronization and time management for this model, and its tasks will be described with more detail when discussing how parallel and distributed execution is implemented. The addition of a *flat coordinator* is key for allowing the execution of the model without intermediate *coordinators*, which was identified as a major source of overhead. The *flat coordinator* is responsible of receiving, translating, and sending messages between its children. In this case, for example, the flat coordinator needs the information about external input coupling, external output coupling, and internal coupling for *Top*, *Coupled #1*, and *Coupled #2*. Thus, the *flat coordinator* builds a flat structure of simulators, and handles all the information about the port mappings for every component in the model.

We can observe the difference between the communication cost of the new flat structure and the hierarchical case. For example, when *Atomic #3* sends an output through the port connected to *Atomic #7*, only two messages are required with the flat approach (instead of the four messages required for the hierarchical approach). Notice that since the *flat coordinator* has all the information about ports and ports mappings, it is not necessary to use any intermediate coordinator. This reduction in the number of exchanged messages improves simulation performance.

So far, we have discussed the basic idea of flat simulation in a single node. We will now study the case of flat distributed simulation using multiple processors.

First, in order to run the model over distributed processors, it is necessary to indicate the nodes that can participate in the simulation. Second, one has to indicate which components will be executed on each processor. Figure 20 shows the layout of our sample model partitioned into three blocks (*0*, *1*, and *2*), and Figure 21 shows how to specify this partition with CD++.

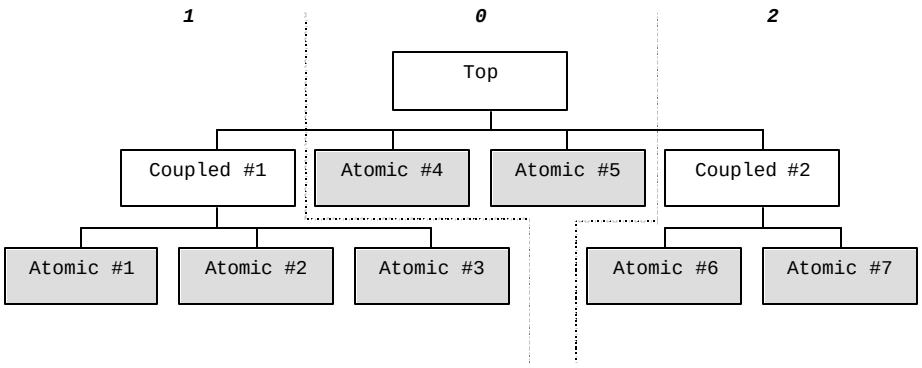


Figure 20: Model partitioned in three blocks

```
0 : atomic_4 atomic_5
1 : atomic_1 atomic_2 atomic_3
2 : atomic_6 atomic_7
```

Figure 21: Model partition file for CD++

As shown in Figure 21, users only have to specify the location for atomic components. Similarly, in Cell-DEVS models it is required to indicate the location of every cell. This can be done by specifying the individual location for every cell or by using ranges, following the notation used in [Tro01a].

During the instantiation and registration of each *simulator* object, *simulators* are associated to the corresponding logical process. The partition, once specified at the beginning of the simulation, is static; it is not possible to migrate *simulators* from one LP to a different one at runtime.

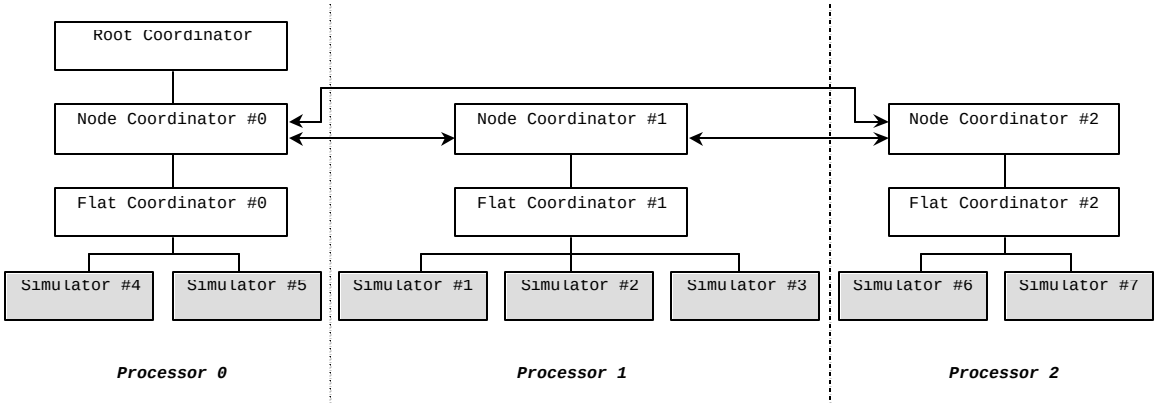


Figure 22: Distributed processor structure for partitioned model

Figure 22 shows the processor structure of our flat distributed simulator using the partitioning specified in Figure 21. The main node, processor 0, executes a logical process composed by *root coordinator*, *node coordinator #0*, *flat coordinator #0*, and

simulators #4 and *#5* (in charge of *Atomic #4* and *#5*). The processor 1 is in charge of *node coordinator #1*, *flat coordinator #1*, and *simulators #1*, *#2* and *#3*. Finally, processor 2 executes a LP with *node coordinator #2*, *flat coordinator #2*, and *simulators #6* and *#7*. *Node coordinators* can communicate with each other using inter-LP messaging (shown with arrows in Figure 22).

Notice that the general structure of processors running on each LP is almost identical. An important difference, however, between the main node (processor 0) and the other nodes is that the execution of the *root coordinator* always takes place in the first one. The *root coordinator* is in charge of starting the simulation and interacting with the environment. Messages received from the environment are handled by the *root coordinator* and then sent to the corresponding *node coordinator*. On the other hand, when a *node coordinator* processes an output that must be sent back to the environment, the output is sent to the *root coordinator*. Then, the *root coordinator* sends the output to the environment.

We can analyze some aspects of the new message passing mechanism between processors that results from an output sent from an atomic component, a_1, to another atomic component, a_2. Basically, two different cases can be observed: both *simulators* for a_1 and a_2 execute on the same logical process, or *simulators* for a_1 and a_2 execute on two different logical processes.

We start by analyzing the first case, which is more simple. When source and destination *simulators* are running on the same logical process, the *flat coordinator* running in that LP takes care of the entire situation. Firstly, the source *simulator* sends

the message to its parent, a *flat coordinator*. Secondly, since the *flat coordinator* has all the necessary information for the port mappings between those components, the *flat coordinator* sends that output to the corresponding *simulator*.

The second case requires involvement of more processors: if a *simulator* running on LP_i has to send an output to a simulator running on LP_j, it is necessary to forward this message to the corresponding node. In such a case, the *simulator* begins by sending its output to the *flat coordinator*. Since the *flat coordinator* identifies that the destination *simulator* is not one of its children, it forwards the message to its parent *node coordinator*. Then, the *node coordinator* running on LP_i forwards the message to the *node coordinator* running on LP_j. This is possible because node coordinators know where each *simulator* is running. Finally, the *node coordinator* running on LP_j forwards the message to its child, *flat coordinator*, which in turn sends it to the destination *simulator*. This situation is shown in Figure 23. The figure shows that node coordinators perform inter-LP communication. Notice that inter-LP communication can lead to violations to the local causality constraint, depending on the time at local and destination LPs. More specifically, if the timestamp of the message is smaller than the local time at the destination LP, a rollback is triggered. A more detailed description of this situation and a sample scenario are given in the next subsections.

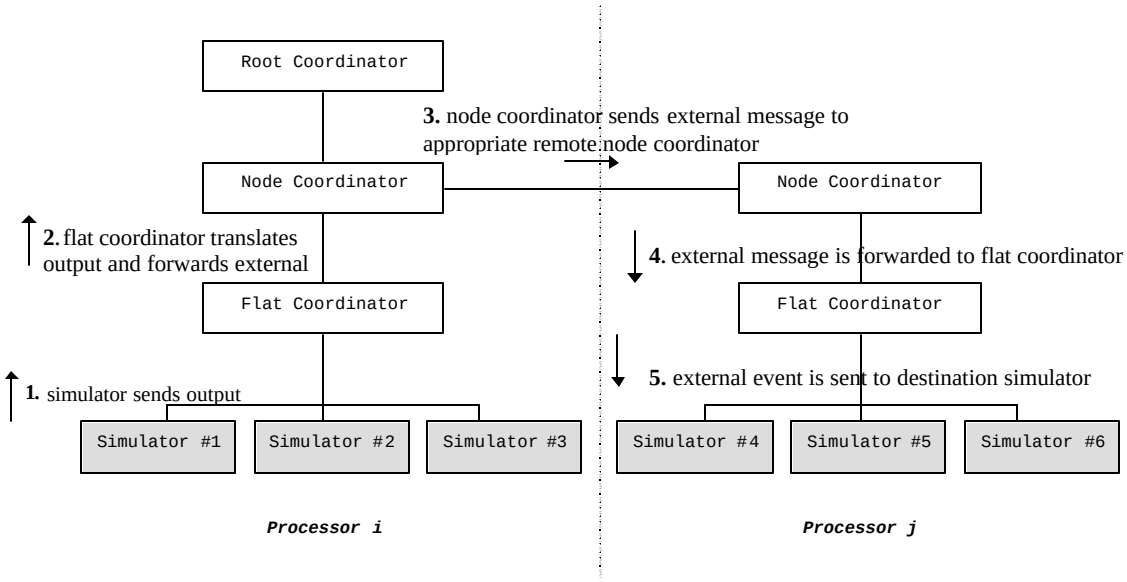


Figure 23: Sending an output to a remote simulator

4.2 ALGORITHMS FOR PARALLEL AND DISTRIBUTED SIMULATION USING A FLAT APPROACH

We describe the simulation mechanism more rigorously by presenting the behavior of each DEVS processor, namely *simulator*, *flat coordinator*, *node coordinator*, and *root coordinator*. These are the processors that carry out parallel and distributed simulation using a flat mechanism for DEVS and Cell-DEVS models. After presenting the algorithms for each processor, we discuss different scenarios showing the flow of messages.

Messages that can be exchanged among processors are: *init* (initialization message), *q* (external message), *y* (output message), @ (collect message), * (internal message), and *done*.

4.2.1 SIMULATOR

A *simulator* is created for each atomic component or cell in the system. It is responsible of generating the outputs and executing the transition functions of the associated model. The algorithms for the *simulator* are based on those presented by [Cho94b] with minor changes:

```
1 when a (init, 0) message is received
2   initialize model's variables
3   t_L = 0
4   t = ta (s)
5   send (done, t) to the parent flat coordinator
6 end when
```

When the initialization message is received, variables are initialized (lines 2 and 3) and the *simulator* informs its parent the time of the next scheduled internal transition (line 5).

```
 1 when a (@, t) message is received
 2   if t = t_N then
 3      y = λ(s)
 4      send (y, t) to the parent flat coordinator
 5      send (done, t) to the parent flat coordinator
 6   else
 7      raise error
 8   end if
 9 end when
10
11 when a (q, t) message is received
12   add event q to the bag
13 end when
```

When a *simulator* receives a (@, t), it generates an output (executing the atomic's output function, λ) which is sent to the parent flat coordinator (lines 3 to 5). When an external message (q, t) is received, it is simply stored in the bag of external events (line 12). These messages will be used later, when the external transition function is triggered.

```
 1 when a (*, t) message is received
 2   case tL ≤ t < tN
 3      e = t - tL
 4      s = δext (s, e, bag)
 5      empty bag
 6   end case
 7   case t = tN  and bag is empty
 8      s = δint (s)
 9   end case
10   case t = tN  and bag is not empty
11      s = δcon (s, bag)
12      empty bag
13   end case
14   case  t > tN or t < tL
15      raise error
16   end case
17   tL = t
18   tN = tL + ta (s)
19   send (done, tN) to parent flat coordinator
20 end when
```

An internal message (*, t) triggers the execution of a transition function. The *simulator* executes one of the three transition functions based on t (the elapsed time since the last scheduled transition), t_N (the time of the next scheduled transition), and the bag of external events.

If $t < t_N$ (lines 2 to 6), the internal transition function shall not be executed yet, and the bag of external events must have at least one element: the external transition function, δ_{ext}, is executed in this case. If $t = t_N$ (lines 7 to 9), it is time to execute the internal transition, δ_{int}. However, a conflict arises if the bag is not empty and $t = t_N$ (lines 10 to 13), the confluent transition δ_{con} has to be executed.

In every case, after executing the corresponding transition, a done message is sent to the parent *flat coordinator* indicating the next scheduled transition time (lines 17 to 19).

4.2.2 FLAT COORDINATOR

A *flat coordinator* has one or more children, which are the *simulators* running the atomic components, and one parent, the *node coordinator*. The *flat coordinator* relies on coupling information for the components running on this LP; it has to translate output events into input events. Additionally, it synchronizes models that are imminent in this logical process using a structure called *synchronize set*.

```
1 when a (init, 0) message is received from parent node coordinator
2    t_L = 0
3    for each child simulator s_i
4          send (init, 0) to child s_i
5    end for each
6    wait until all done messages have been received
7    t_N = minimum t_N of all components
8    send (done, t_N) to parent node coordinator
9 end when
```

When the initialization message is received, the *flat coordinator* forwards a (init, t) to all its children to complete the initialization phase (lines 3 to 5). Using the done messages received from them, the minimum time of next change is computed and communicated to the parent *node coordinator* via a done message (lines 6 to 8).

```
 1 when a (@, t) message is received from parent node coordinator
 2   if t = t_N then
 3         t_L = t
 4         for each imminent child s_i with minimum t_N
 5               send (@, t) to child s_i
 6               cache i in the synchronize set
 7         end for each
 8         wait until all done messages have been received
 9         send (done, t) to parent node coordinator
10   else
11         raise error
12   end if
13 end when
```

When a collect message (@, t) is received, the *flat coordinator* forwards this message to all its dependant *simulators* with minimum t (lines 3 to 7). Once all the responses (i.e., done messages from *simulators* which received a collect message in this simulation cycle) have been received (line 8), a done message is sent to the parent node coordinator. *Simulators* that have been scheduled for a transition are cached in the synchronize set.

```
 1 when a (y, t) message is received from child i
 2   if destination of y is the environment
 3      send (y, t) to parent node coordinator
 4   else
 5      for each influencee j of child i
```

```
 6          q = z_{i,j} (y)
 7          if (j is a local processor) then
 8             send (q, t) to child j
 9             cache j in the synchronize set
10          else
11             send (q, t) to parent node coordinator
12          end if
13       end for each
14    end if
15 end when
```

If the destination of the output (y, t) message received in the *flat coordinator* is the environment, the message has to be sent to the parent *node coordinator*, which will deal with this situation (lines 2 and 3). If not, all the influencees of the message are computed using the function Z_{ij}, and one or more (q, t) messages are sent accordingly (lines 5 to 13). For destination processors located on the same LP, messages are sent directly to the *simulator* (lines 8 and 9). Messages whose destinations are remote simulators are sent to the parent *node coordinator* (line 11), which will forward them to the corresponding LPs. Again, local components with scheduled transitions are cached in the synchronize set. We discuss a sample scenario that describes this situation after presenting the algorithms.

```
1 when a (q, t) message is received from parent node coordinator
2    if destination of q message is local then
3       add event q to the bag
4    else
5       raise error
6    end if
7 end when
```

When an external message (q, t) is received in a flat coordinator, it is stored in a bag of events.

```
 1 when a (*, t) message is received from parent node coordinator
 2   if t_L ≤ t ≤ t_N then
 3      for each q ∈ bag
 4         for each local receiver s_j of q
 5            send (q, t) to s_j
 6            cache j in the synchronize set
 7         end for each
 8      end for each
 9      empty bag
10      for each i ∈ synchronize set
11         send (*, t) to i
12      end for each
13      wait until all done messages are received
14      t_L = t
15      t_N = minimum t_N of all components
16      clear the synchronize set
17      send (done, t_N) to parent node coordinator
18   else
19      raise an error
20   end if
21 end when
```

Upon receiving an internal message (*, t), the flat coordinator sends the external messages that are stored in the bag to the corresponding components (lines 3 to 8). All the receivers of these external messages are added to the synchronize set. Then, an internal message is sent to all components in the synchronize set. After all done messages are received back from these components, the time of the next event is calculated and a done message is sent to the *node coordinator* (lines 13 to 17).

4.2.3 NODE COORDINATOR

One *node coordinator* is located on each logical process and it has one child, a *flat coordinator*. *Node coordinators* have important tasks associated to inter-LP communication, which happens when an atomic model running in the local LP has to send an output to another atomic model running in a remote LP. Additionally, a *node coordinator* is in charge of advancing the simulation time in the local LP based on the information received from the *root coordinator* and from its dependant *flat coordinator*. The algorithms describing its behavior are described next.

```
 1 when a (init, 0) message is received from root coordinator
 2   send (init, 0) to child flat coordinator
 3   wait for done message to be received from flat coordinator
 4   sort queue of events by arrival time
 5   t = min (t_N of flat coordinator, time of first event in queue)
 6   if t = t_N of queue then
 7      for each q in queue with time t
 8         send (q, t) to flat coordinator
 9      end for each
10   end if
11   send (@, t) to child flat coordinator
12   next-message-type = *
13 end when
```

The initialization message, sent by the *root coordinator*, triggers the simulation in each logical process. An initialization message (init, 0) is sent to the *flat coordinator* (line 2), which in turn will forward that message to every *simulator*. The first simulation cycle starts immediately after a (done, t) message is received. The time for the first collect message is determined by the minimum between the first element in queue of external

events and the time of next change reported by the flat coordinator, which represents the minimum time of next change reported by *simulators* (lines 3 to 5). The variable next-message-type is used on each simulation cycle (after the reception of done messages) to determine which type of message has to be sent (i.e., collect or internal).

```
 1 when a (done, t) message is received from child flat coordinator
 2   if next-message-type = * then
 3      send (*, t) to child flat coordinator
 4      next-message-type = @
 5   else
 6      t = min (t_N of flat coordinator, time of first event in queue)
 7      if t > stop simulation time then
 8         stop simulation in this LP
 9      else
10         if t = t_N of first event in queue then
11            for each q in queue with time t
12               send (q, t) to flat coordinator
13            end for each
14         end if
15      end if
16      send (@, t) to child flat coordinator
17      next-message-type = *
18   end if
19 end when
```

If the message to be sent is a collect (lines 6 to 17), the process is analogous to the initialization phase. The minimum time t is computed, events with time t are sent (if there are any), the collect message is sent (line 16) and the next message type is set to internal (line 17). When an internal (*, t) message has to be sent to finish the current simulation cycle, the type of the next message to be sent is set to collect (line 4).

```
 1 when a (q, t) message is received
 2   if destination q is local
 3      send (q, t) to child flat coordinator
 4   else
 5      dest_nc = node coordinator running atomic model that must receive q
 6      send (q, t) to node coordinator dest_nc
 7   end if
 8 end when
 9
10 when a (y, t) message is received from child flat coordinator
11   if send-outputs-from-NC
12      send output (y, t) to environment
13   else
14      send output (y, t) to parent root coordinator
15   end if
16 end when
```

An external message (q, t) can be received in a *node coordinator* either from another (remote) *node coordinator* or from its dependant *flat coordinator*.

In the first case, this event must be sent to the dependant *flat coordinator* (line 3). This happens when a remote atomic component sends an output through a port connected to an atomic component executing in the local LP. As we have shown earlier, this message is forwarded by the *flat coordinator*'s algorithm to the corresponding *simulator*. Notice that the timestamp t of a message received from a remote *node coordinator* might be lower than the current time in this LP, which would violate the local causality constraint. In such a case, the LP has received an event in the past (a straggler message) and therefore it has to recover from this incorrect state by performing a rollback. The rollback has to bring that object back to a correct state: a state whose time is equal or

smaller than the time of the straggler message. In addition, the messages that were (incorrectly) transmitted from this *node coordinator* have to be canceled, which means that anti-messages have to be sent to the destination objects. We will address this situation later with further details giving a sample scenario.

In the second case, the message must be sent to the remote LP where the destination atomic component is running. Thus, it is necessary to determine which *node coordinator* is in charge of that LP, and then the message can be sent using inter-process communication (lines 5 and 6). Notice that this operation can cause a rollback in the destination LP, if the time at that remote LP is greater than the local time.

When a *node coordinator* receives an output message from its child (lines 10 to 16), a message has to be sent to the environment. There are two ways of dealing with outputs that have to be sent back to the environment. Our simulator uses the parameter *send-outputs-from-NC* to determine whether outputs must be processed directly by the *node coordinator* (line 12), or via the *root coordinator* (line 14). The first alternative reduces the number of messages required to process an output (messages do not have to travel through the *root coordinator*) but requires some post-processing if the outputs of multiple *node coordinators* have to be merged together. The second alternative centralizes the actual processing of outputs in the *root coordinator*; it does not require any post-processing but the overhead is larger. Notice that the number of inter-LP messages sent from *node coordinators* (running on machines 1 to *n*) to *root coordinator* can be large depending on the model's output behavior.

```
1 when a (q, t) message is received from parent root coordinator
2   add q to the sorted queue of events
3 end when
```

When an external event is received from the *root coordinator*, the event is stored in timestamp order in the queue of events. The destination *simulator* for that event will eventually receive it when that time is reached by this LP. We have shown earlier that the time is advanced in the *node coordinator* upon the reception of a (done, t) message.

4.2.4 ROOT COORDINATOR

The *root coordinator* is a special processor located in only one LP. It is responsible for starting the simulation, dealing with external events, and sending outputs back to the environment.

```
1 for each child node coordinator nc_i
2  send (init, 0) to nc_i
3 end for each
```

The *root coordinator* starts the simulation by sending initialization messages to every *node coordinator*. These coordinators are located on the different logical processes that form the simulation.

```
1 when a (q, t) is received from environment
2   t_L = t
3   for each child node coordinator nc_i which shares LP with
4      a destination atomic model of q message
5      send (q, t) to nc_i
6   end for each
7 end when
```

External events are received from the environment in the *root coordinator*. The *root coordinator* sends an external event to *node coordinators* that have one or more atomic model that should receive that message (lines 3 to 6).

```
1 when a (y, t) is received from child node coordinator
2   t_L = t
3   send (y, t) to environment
4 end when
```

Output messages received by the *root coordinator* are sent back to the environment. This code is never executed if the parameter *send-outputs-from-NC* is set, as shown in *node coordinator*'s algorithm for processing output messages. If *send-outputs-from-NC* is not set, the *root coordinator* consolidates the processing of output messages.

Figure 24 summarizes the flow of messages in distributed simulation of Parallel DEVS and Cell-DEVS models using the previous algorithms. Arrows indicate the direction of the message. The interaction between the environment and the *root coordinator*, shown in Figure 24 with a dashed line, is performed in CD++ by using an input file (for external events) and an output file (for outputs generated by the model). As we mentioned earlier, the *root coordinator* is also in charge of starting the simulation process by sending initialization messages. Although it is not shown in the diagram, the parameter *send-outputs-from-NC* allows sending outputs from the *node coordinator* to the environment without relying on the *root coordinator* as an intermediary. Some implementation issues associated with this are discussed later.

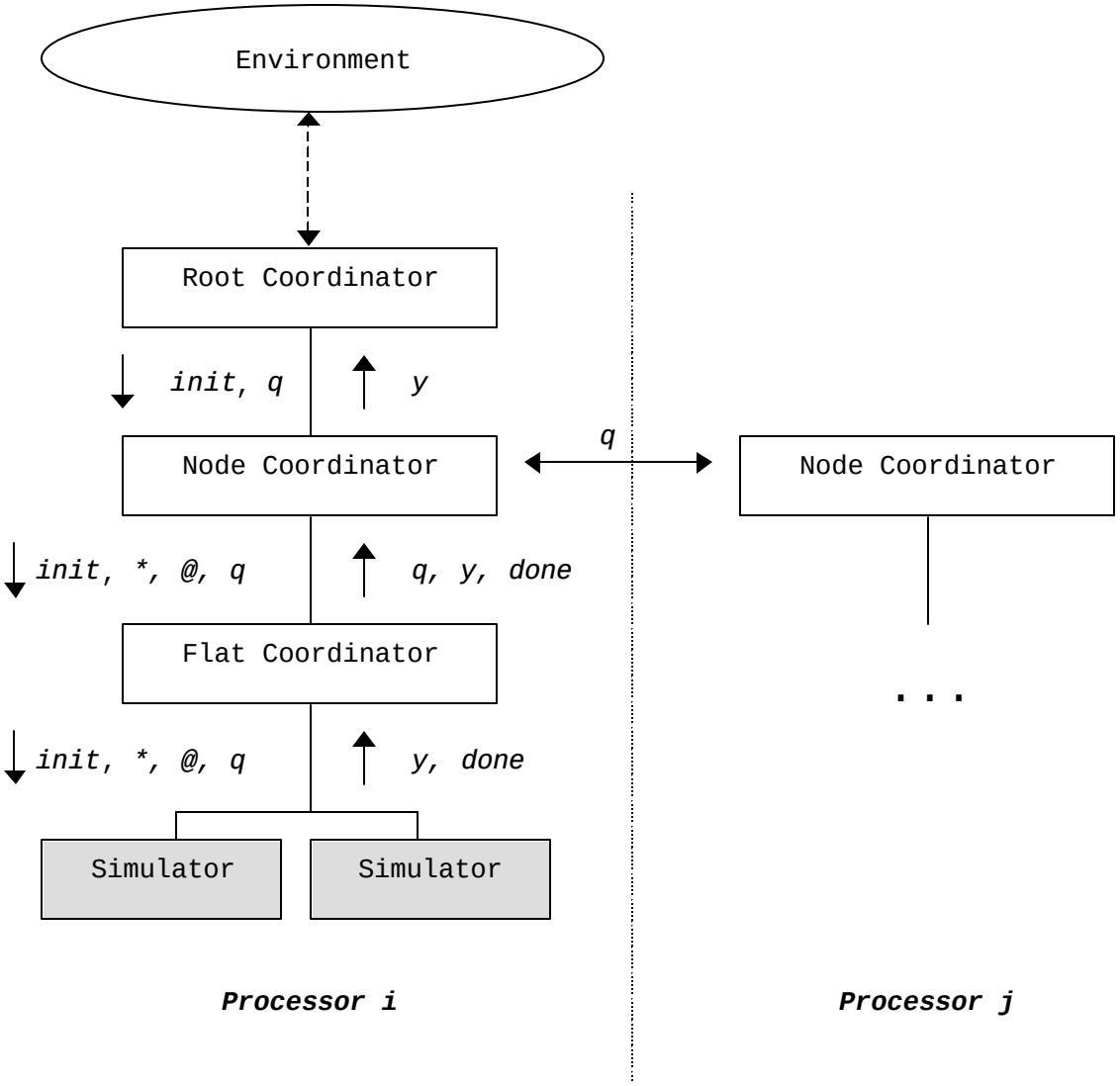


Figure 24: Message flow in a distributed simulation of DEVS and Cell-DEVS

Instead of using files to interact with the real world, a different approach is to use communication ports (e.g., the serial communication interface) to receive events from and send outputs to the environment. This approach is particularly interesting when developing models that interact with hardware components. A detailed description of this

alternative and some examples are given in [Gli04, Wai04] using a real-time CD++ simulator [Gli02a].

4.3 SAMPLE SCENARIOS

To better describe the behavior of the flat distributed simulation, we introduce different scenarios for the simulation of a sample bidimensional Cell-DEVS model. The execution of this 10x10 model is divided in two processors, each of which executes a rectangular area of 10x5 (i.e., 50 cells per machine).

Figure 25 shows the initialization phase for this sample model. The first simulation cycle is started by the *root coordinator*, which sends an initialization message to the *node coordinators* in LP 0 and 1 (messages 1 and 2). When the *(init,0)* message is received in a *node coordinator*, it is forwarded to the *flat coordinator* (messages 1.1 and 2.1). Then, *flat coordinators* forward these messages to their *simulators* (messages 1.2 to 1.51 for processor 0 and 2.2 to 2.51 for processor 1), and *simulators* execute the initialization function for each cell. After computing the time for the next change for that cell (using its time advance function), every simulator sends a *done* message to its parent *flat coordinator* reporting its time of next change. For example, S_1 indicates that there is an internal transition function to be executed at time 100 (message 1.52), whereas S_2 reports that there is no scheduled internal transition (message 1.53, which contains *inf* or infinity, and represents that the model is in a passive state). After receiving all done messages, the *flat coordinator* sends a done message to its parent *node coordinator* (messages 1.103 and 2.103) with the minimum time of its components, which in this case

is 100 for both LPs. Having received this information, the *node coordinator* checks for external messages to be sent at this point, and then it is ready to send the first collect message.

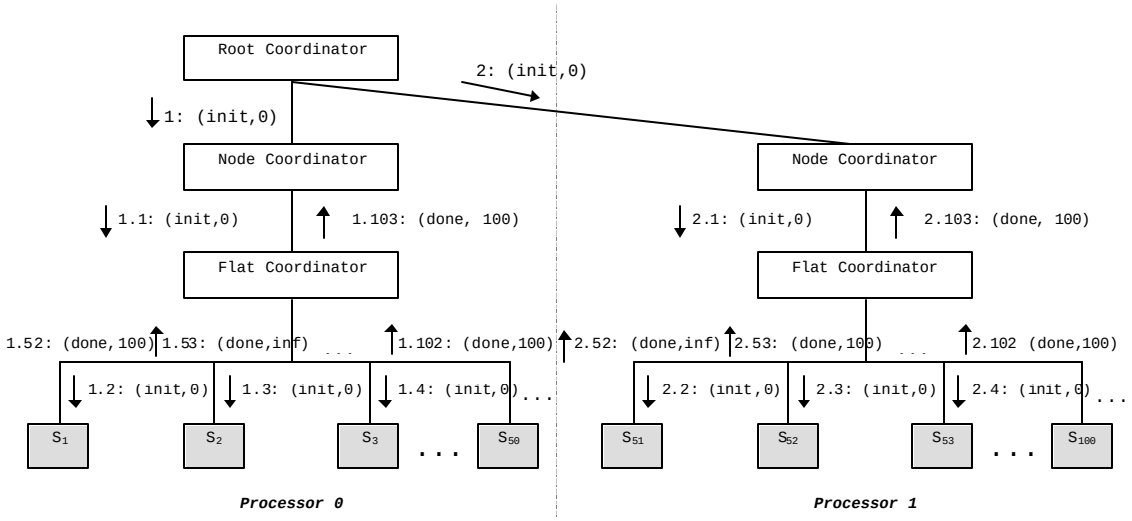


Figure 25: Initialization phase in sample Cell-DEVS model

The goal of the next phase is to collect the outputs of the imminent components. Figure 26 shows how *node coordinators* start the collect phase by sending the first message to their *flat coordinators* (messages 1 and 2). The *flat coordinators* forward a collect message only to imminent children, i.e., to *simulators* whose time of next change is the minimum (100). For example, in LP 0 it is sent to S_1 (message 1.2) but not to S_2, since the latter has reported a time of next change of *inf* with its last done message (see message 1.53 in Figure 25). When receiving a collect message, *simulators* execute their output functions and send the result to its parent (e.g., messages 1.20 and 2.18). Although it is not shown in the figure, the *flat coordinator* translates the received output messages and sends the external messages to the corresponding local influencees of that

component. In case of a remote destination and as discussed earlier, the message is sent to the local *node coordinator*, which will then forward it to the corresponding remote *node coordinator*. Additionally, *simulators* send done messages to the *flat coordinator* after sending the outputs. Then, the *flat coordinator* sends a done message to the *node coordinator* completing the collect phase.

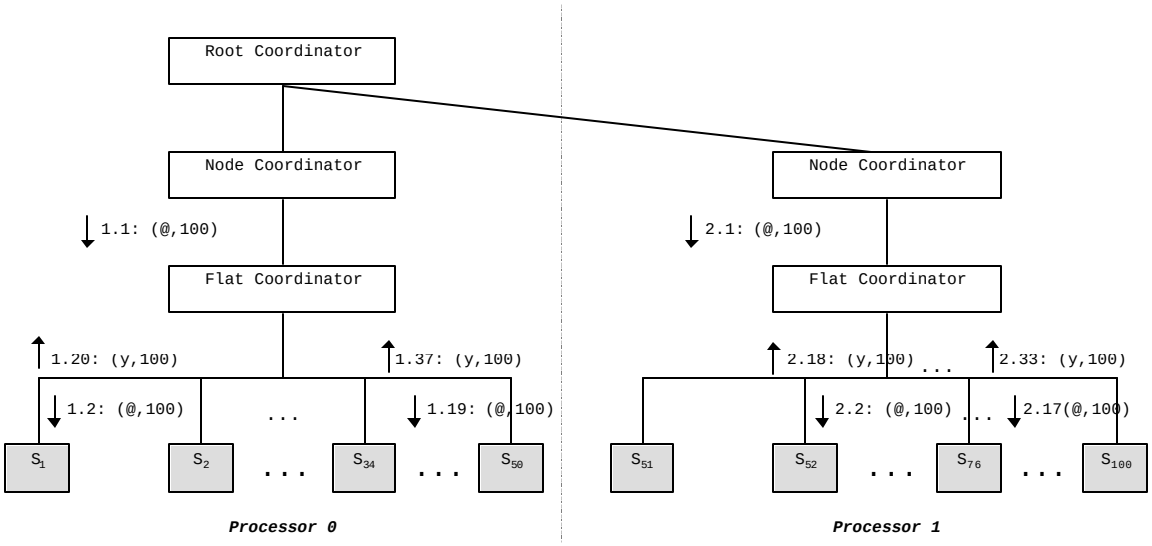


Figure 26: Collect phase in sample Cell-DEVS model

At this point, the *node coordinator* is ready to send an internal message (*) to start the next phase. This new cycle is similar to the collect phase in terms of message exchange: the *flat coordinator* forwards the internal message only to *simulators* that have a scheduled transition for the current simulation time (100). *Simulators* execute the internal, external, or confluent transition function according to the current time, the time of next change and the state of the bag of events (empty or not empty), as specified in the

algorithms shown earlier. Done messages are sent to inform the time for the next transition, and the *node coordinator* is ready to start a collect cycle again.

For the sake of simplicity, we assumed that *node coordinators'* external event queues are empty or have events whose timestamps are greater than the local time. The *node coordinator*'s algorithm, presented earlier, describes how to deal with external messages that are pending to be sent.

So far, we have only considered cases where communication is performed within the same logical process. Therefore, these scenarios adhere to the local causality constraint at all times, since events and messages are processed in nondecreasing timestamp order as scheduled by *node coordinators*. Nevertheless, as we mentioned earlier in this chapter, *node coordinators* can communicate with each other. When a component running in LP_i has to send an output to a component running on LP_j, a *node coordinator* is in charge of sending this message using inter-LP communication. In this case, it is possible to receive a straggler message. If the message has a timestamp greater than the local time in the destination LP, the simulation can continue normally, which is the more simple case. However, if the message has a timestamp earlier than the local time, this is a straggler message: a violation to the local causality constraint has occurred, and a rollback has to be performed. This scenario is illustrated in Figure 27.

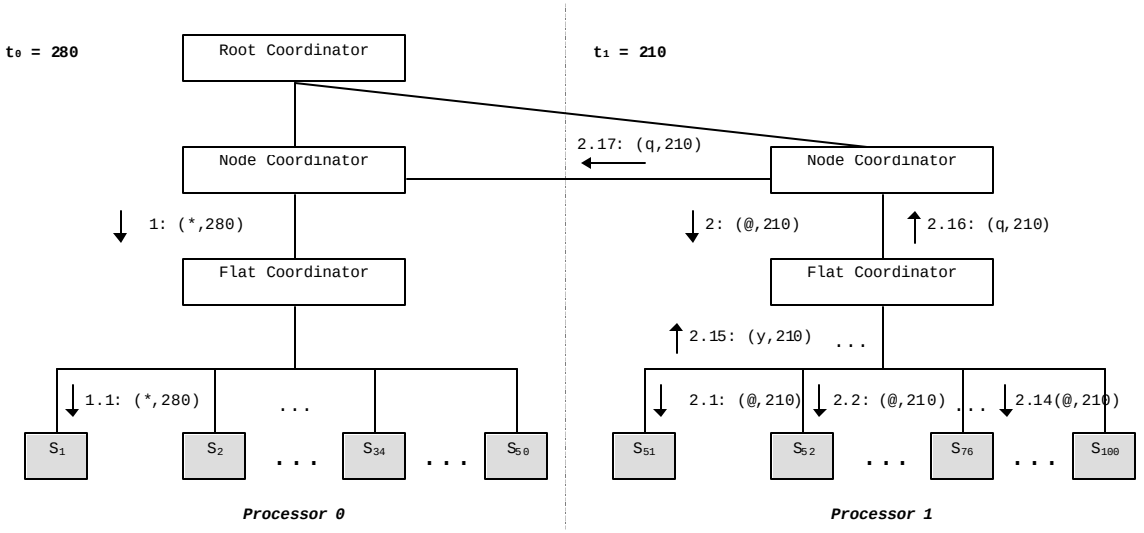


Figure 27: Straggler message received during the simulation of a Cell-DEVS model

Figure 27 shows the events that lead to the reception of a straggler message in processor 0. The local time at processor 0, t_0, is 280 and t_1 is 210. In processor 0, the *node coordinator* has sent an internal message, which is being forwarded by the *flat coordinator* to S_1 (messages 1 and 1.1). At the same time, in LP 1 the *node coordinator* has sent a collect message (message 2), which after being forwarded (message 2.1 to 2.14) results in an output from a S_{52} (message 2.15) that has to be sent to S_1. This message is forwarded as an external message, *q*, from the *flat coordinator* (2.16) to the *node coordinator*. Then, the *node coordinator* in processor 1 forwards it to the *node coordinator* in processor 0 (message 2.17) because that is where S_1 is being executed. At the moment of receiving the external message in the destination *node coordinator*, one can see that the timestamp of the message, 210, is smaller than the time at the local processor, $t_0 = 280$. This straggler message (2.17) triggers a rollback in processor 0.

In case of a rollback, different tasks have to be performed. These tasks are mainly carried out by the Time Warp algorithm implemented by Warped (as discussed in Chapter 2), and allow the simulation to recover from a local causality violation. Figure 28 shows the state of the *node coordinator*'s input, state, and output queues at the moment of receiving a straggler message with timestamp 210.

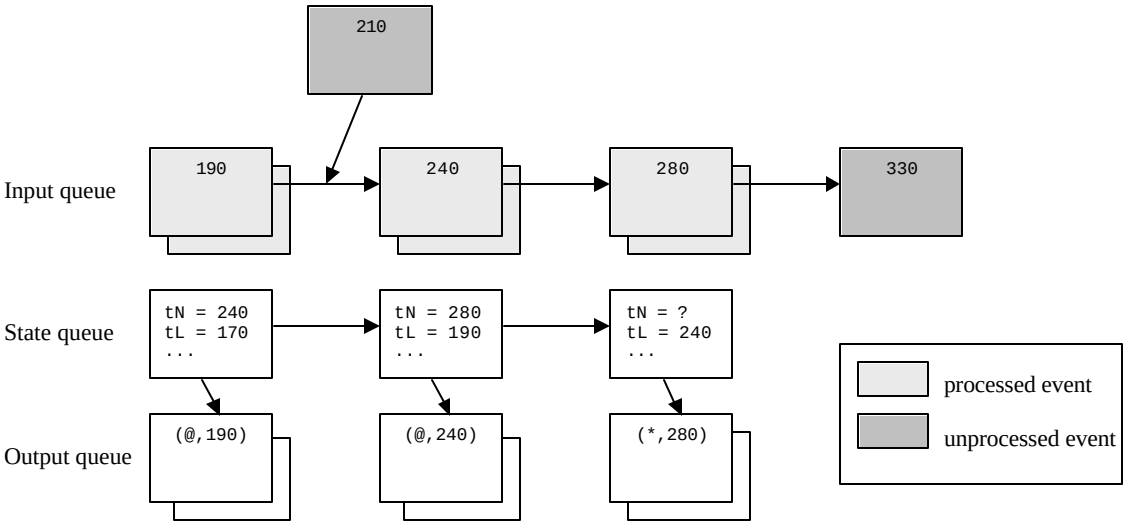


Figure 28: Reception of a straggler message in a *node coordinator*

Firstly, the state of the *node coordinator* has to be restored to a previous state where the time is equal to or smaller than 210, which is the straggler's timestamp. This is possible because Time Warp stores the previous states of simulation objects. Figure 28 shows that, in this case, the object has to restore its state back to time 190. Secondly, the *node coordinator* has to send anti-messages to other objects that had received messages from it in states that are now being rolled back (i.e., messages sent a times 240 and 280). A negative message is a duplicate of the original one with a flag indicating it is actually an anti-message. This mechanism propagates the rollback to the corresponding

simulation objects. Figure 29 depicts the *node coordinator*'s queues after the rollback was completed.

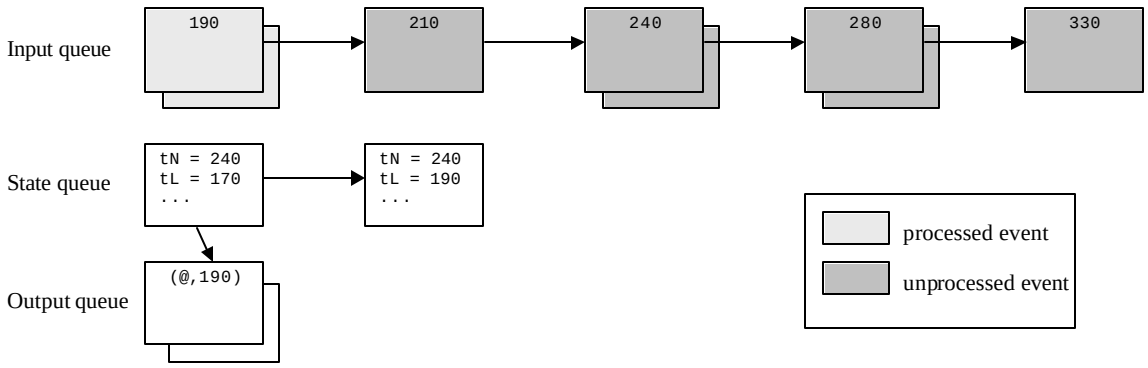


Figure 29: State of the *node coordinator* after the rollback

After all the states were rolled back and the negative messages were sent, the *node coordinator* can return to process the events, starting by the one that caused the rollback.

Chapter 5: IMPLEMENTING THE ABSTRACT SIMULATORS

This chapter presents some implementation issues of the new distributed CD++ simulator, which is built on top of the Warped middleware (version 1.02) [Mar97]. Warped and MPICH [Gro96], its underlying communication layer implementing the MPI protocol [MPI95, Don96], are written in C++ and were compiled with open source GNU C++ compiler, g++, version 2.9x.

To understand the implementation of our simulator over Warped, it is necessary to examine Warped's application program interface with more detail, as we extended many of these classes. The interface is based on Jefferson's work on Time Warp [Jef85] and provides several basic definitions of classes that deal with simulation objects, states, events, and logical processes.

Four of the fundamental classes that form Warped's API are shown in Figure 30 along with some of their methods and variables. The figure shows the basic class definitions for simulation objects, states for simulation objects, and events that can be exchanged by simulation objects. It also shows part of the *LogicalProcess* class.

```
class TimeWarp {
    TimeWarp();
    virtual ~TimeWarp();
    virtual void initialize();
    virtual void finalize();
    virtual void executeProcess() = 0;
    void saveState();
    virtual void rollback(VTime);
    void rollbackFileQueues(VTime);
    VTime calculateMin();
    void inputGcollect(VTime);
    void stateGcollect(VTime);
    void outputGcollect(VTime);
    void sendEvent (BasicEvent * );
    BasicEvent* getEvent();
}

class BasicState {
    BasicState();
    virtual ~BasicState();
    BasicEvent* inputPos;
    Container<BasicEvent>* outputPos;
    virtual BasicState& operator=(BasicState&);
    BasicState* copyState( BasicState*);
}

class BasicEvent {
    int size;
    int sender;
    int dest;
    Vtime sendTime;
    Vtime recvTime;
}

class LogicalProcess {
    LogicalProcess(int, int, int);
    registerObject(TimeWarp);
    int getNumObjects();
    int getTotalNumberOfObjects();
    int getLPid();
    void allRegistered();
    void simulate(VTime);
    void calculateLGVT();
    void calculateGVT();
}
```

Figure 30: Some classes of the Warped API [Mar97]

TimeWarp is the basic class provided by Warped that defines data and methods needed for every simulation object to participate in a simulation. The three main methods that determine the behavior of simulation objects are *initialize*, *finalize* and *executeProcess*. The method *initialize* is called once at the beginning of the simulation

for every object. The method *executeProcess* contains code to be performed every time a simulation object is scheduled for execution, i.e., when it has an event ready to be processed. From the moment when *executeProcess* is called until its execution is completed, no other object can be under execution on the same logical process. The method *finalize* is executed for each simulation object at the end of the simulation, and it is usually used to release allocated memory, collect statistics, etc. The method *saveState* is called automatically by the Warped kernel to save the current state of an object. This method is triggered by the logical process at the end of each simulation cycle to store information that might be needed later in case of rollbacks. In case of receiving a straggler message with a timestamp *t*, *rollback(t)* is called to rollback this object to a previous time (which is equal or prior to the specified time *t*). This method restores the state of the object and sends the necessary anti-messages. *rollbackFileQueues* performs a rollback on the files associated with this simulation object. The method *calculateMin* reports the minimum time of the unprocessed events, and is used to compute the global virtual time. Garbage collection in the input, queue, and output queues is performed by the methods *inputGcollect*, *stateGcollect*, and *outputGcollect*, respectively. Using the time specified as a parameter, these methods invalidate the states and events and release the memory associated with them. *getEvent* and *sendEvent* are used for receiving and sending messages and will be discussed with more detail later.

The state of a simulation object is defined by an instance of the basic Warped class *BasicState* (or by a user-defined class that inherits from it). The state of an object contains the information that can change in each simulation cycle, including pointers to

input and output queues (*inputPos* and *outputPos*). Methods to determine whether two events are equivalent (using operator =) and a to create a copy of this state (*copyState*) are also provided.

Simulator objects communicate by exchanging messages, which belong to the class *BasicEvent* or to one of its subclasses. A valid message must contain, at least, information about its size, source, destination, local time at source, and timestamp (i.e., the time at which it should be processed). The timestamp of the message must be greater or equal to the local time.

LogicalProcess is the class that groups the simulation objects that execute in the same machine. To create a new logical process, it is necessary to specify the total number of objects in the simulation, the number of simulation objects to be handled on this LP, and the number of LPs participating in the simulation. The method *registerObject(TimeWarp)* is used to define which objects are running on this LP, and the method *allRegistered* indicates that every component has been registered. *allRegistered* is used to determine if every simulation object has an associated LP. The method *simulate(VTime)* starts the execution of this logical process. If a parameter is specified, the simulation stops when the GVT is greater than the specified time; otherwise, the simulation runs until completion. *getNumObjects*, *getTotalNumberOfObjects* and *getLPid* are methods that report basic information about the LP, namely number of local simulation objects, total number of simulation objects, and id associated with this LP. *calculateLGVT* is used to compute the local global virtual time at the end of each simulation cycle. It is calculated by a *GVTManager* as the minimum time reported by

simulation objects. *calculateGVT* is used to compute the global virtual time, and is also handled by the class *GVTManager*.

Figure 31 shows the new class diagram of the DEVS processors along with some of their main methods that implement the algorithms described in Chapter 4.

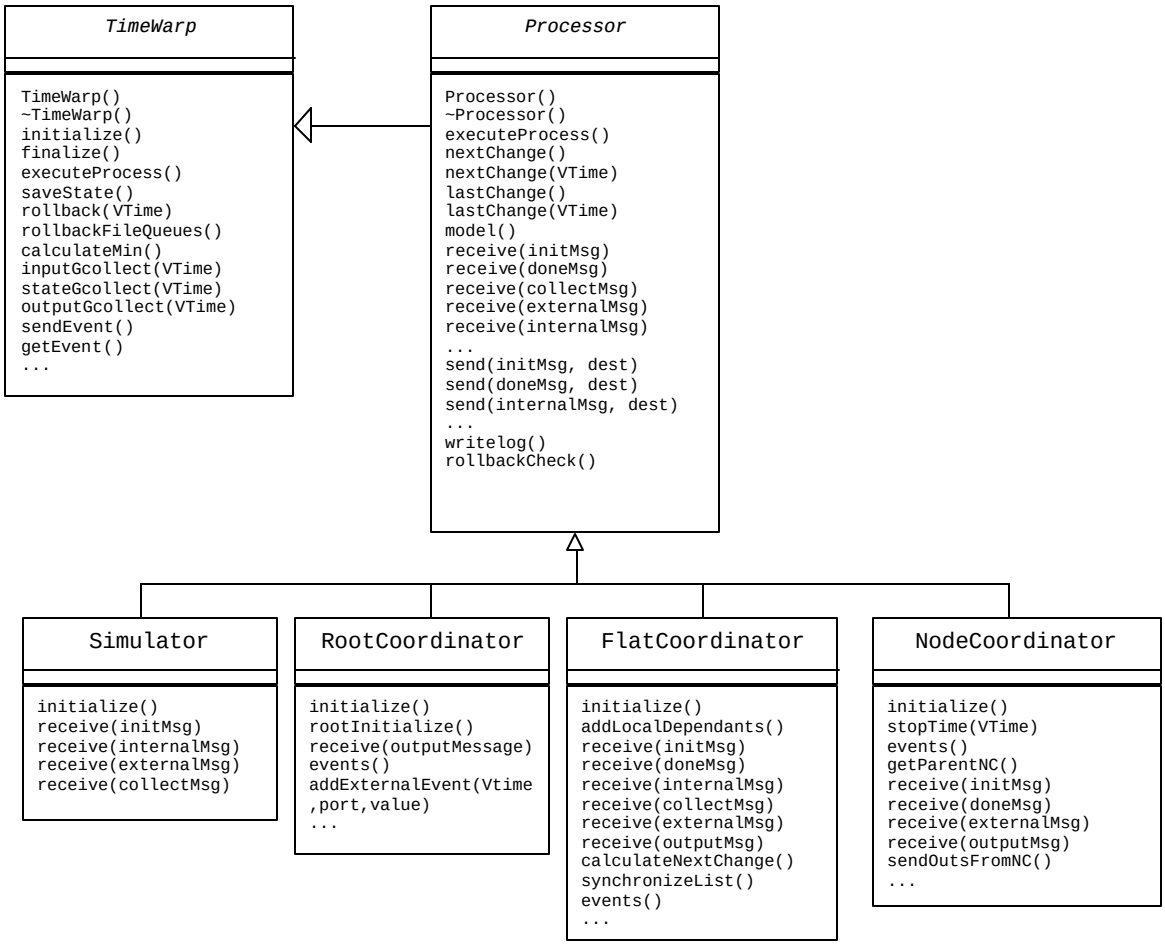


Figure 31: UML class diagram for the new DEVS processors

We defined *processor* as an abstract class that inherits from Warped *TimeWarp* class. *processor* provides basic functionality and data that are common to all DEVS processors in the application. It defines the methods *initialize, executeProcess* and

finalize as well as other methods and variables. It also defines some methods (e.g., *receive(initMsg)*, *receive(doneMsg)*) that have to be redefined by its subclasses, as we will show next. In general, *processor* includes the definition of:

a) send methods (e.g., *send(initMsg,dest)*, *send(doneMsg,dest)*), for sending each type of message. Send methods defined by CD++ use, in turn, the method *sendEvent* defined by Warped in the *TimeWarp* class.

b) time management methods (e.g., *timeNext()*, *timeLast()*, *timeNext(VTime)*, *timeLast(VTime)*), which are used to report and update the time of next scheduled change, time of last change, etc., associated with this processor.

c) *initialize*, *finalize*, and some debugging methods (e.g., *writeLog()*), which perform tasks that are common to all processors. Some of these tasks include opening and closing log files for the associated simulation object, writing in those log files, and printing the processor's name and identification.

d) *executeProcess()*, which is the method that defines the behavior of any DEVS processor, as explained later.

e) *rollbackCheck()*, which is called in the receive method, and checks for straggler messages (i.e., whether the timestamp of the received message is smaller than the time at this processor), and

f) some basic variables, such as model associated to this processor, processor's parent, id and descriptors.

The method *executeProcess* of the *processor* class is common to every DEVS processor, and therefore it is not redefined by any of its subclasses.

processor.executeProcess() is in charge of getting the first event in the queue of events (using the method *getEvent*, which is defined by the Warped kernel), logging the necessary information, and calling the corresponding *receive* method based on the message type. The *receive* method casts the event to its correct type using an enumerated field, *messageType*, which will be described later when we discuss the definition of messages.

The *receive* methods for each DEVS processor are the actual implementation of the algorithms presented in Chapter 4. These methods describe what to do in case of the reception of a message. For example, the *receive (initMessage)* method of a flat coordinator follows the algorithm presented earlier. First, it sends initialization messages to all of its children (using the method *send(initMessage,dest)*). Second, it has to wait until all done messages are received from its dependant *simulators*. *nodeCoordinator* keeps track of the number of done messages it has received using the method *doneCount()*. Finally, it determines and updates the time of next change (using *nextChange(VTime)*, implemented in *processor*) and sends this value to its parent *node coordinator* (using *send(doneMsg,dest)*, also implemented in *processor*). The *receive (initMessage)* method defined for *simulator*, in contrast, initializes the model variables, computes the next time for the next transition (using the time advance function, *ta*) and sends a done message to its parent, which is a *flat coordinator*. Since a *simulator* has access to the definition of its associated atomic model, it is possible for it to execute its functions (e.g., internal transition function, time advance function).

In addition to the different DEVS processors that we discussed, we defined states and events associated with those processors, which extend Warped basic classes.

We defined a class for the basic state of a DEVS processor, *ProcessorState*, which inherits fom the Warped class *BasicSate* shown in Figure 30. *ProcessorState* defines basic time-related data, such as the time of last change and time of next change. It stores data that represents the object's state and can change at each simulation cycle. The *simulator* also associates to its own state the value of associated atomic component, defined as *AtomicState.*

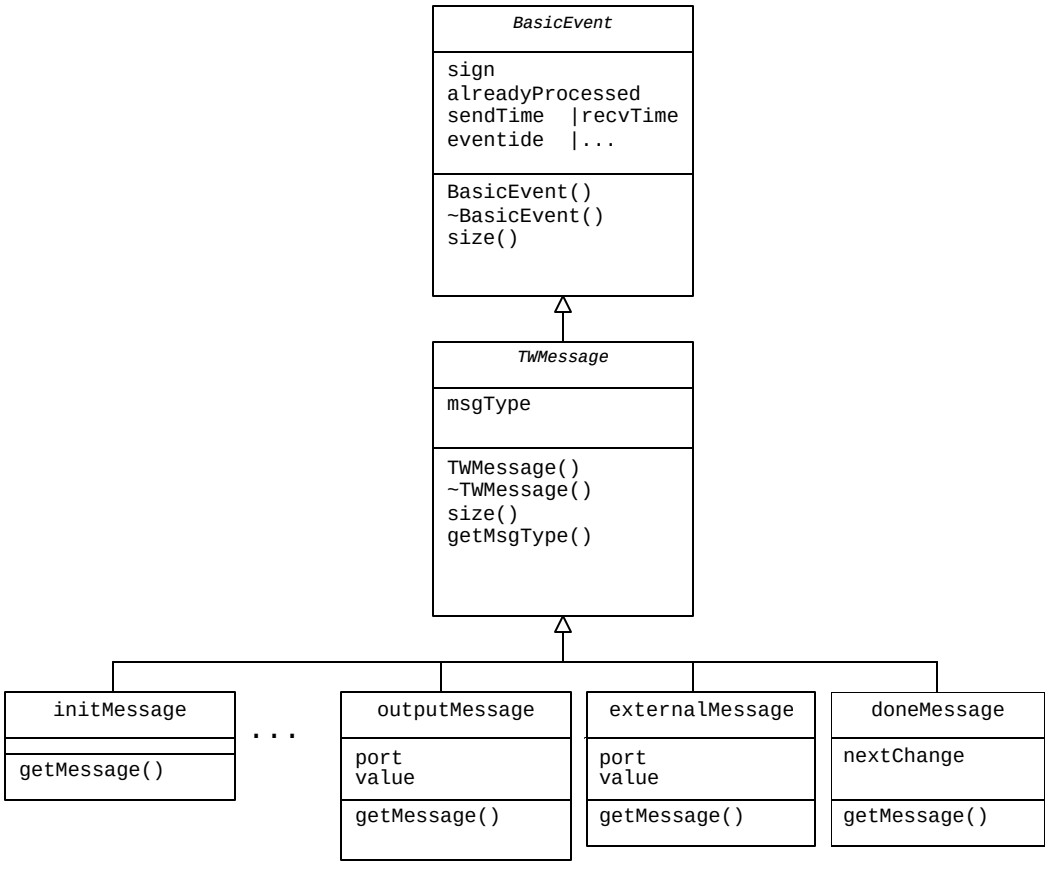


Figure 32: Class diagram for messages in CD++

Figure 32 shows the corresponding class diagram for message-related classes. We defined classes for messages that are exchanged by the processors: *initMessage*, *internalMessage*, *externalMessage*, *outputMessage*, *collectMessage*, and *doneMessage*. All of them inherit from our class *TWMessage*, which in turn inherits from Warped class *BasicEvent*.

Warped guarantees that every TimeWarp simulation object (which in our application means every instantiation of any DEVS processor) includes an input queue, an output queue, and a state queue. The input queue holds the events that the object has to process (possibly, some that have already been processed are also kept). The output queue holds events generated and sent by this simulation object. In case of a rollback, the object's output queue is used to issue negative messages as specified by Jefferson's Time Warp algorithm, as described in Chapter 2. As demonstrated in [Jef85], anti-messages can only be sent for messages whose timestamps are later than the global virtual time. Therefore, messages with timestamps that are earlier than or equal to GVT can be deleted. The state queue holds previous states for this TimeWarp object. Similarly, the state queue only has to keep states whose timestamps are later than the GVT, so that in case of a rollback that state can be recovered.

We mentioned that *LogicalProcess* is one of the fundamental classes defined by Warped. Figure 33 shows a class diagram for *ParallelMainSimulator*, which extends the basic *LogicalProcess* outlined in Figure 30. *ParallelMainSimulator* is the class that implements logical processes in our application.

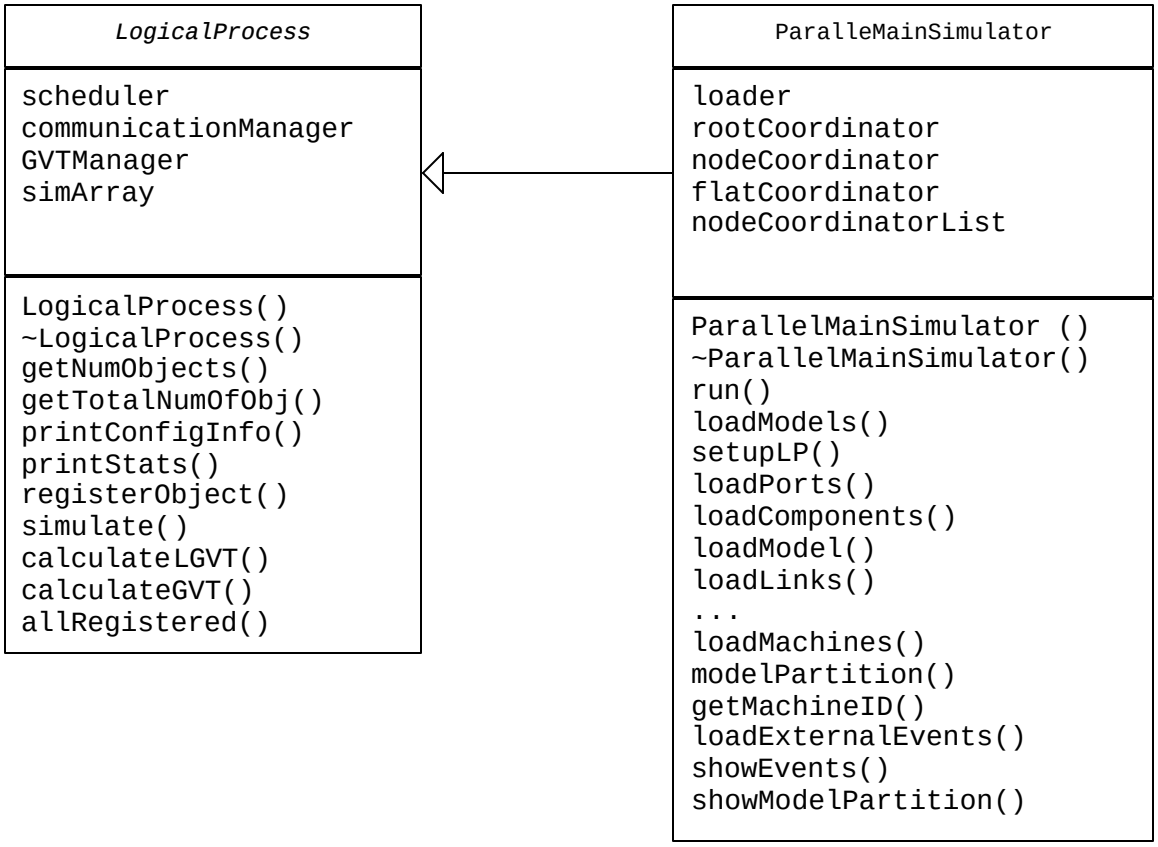


Figure 33: Classes *LogicalProcess* and *ParallelMainSimulator*

At the beginning of a distributed CD++ simulation, one instance of *ParallelMainSimulator* is set up on each machine. Each *ParallelMainSimulator* groups the simulation objects running on this logical process: a *node coordinator*, a *flat coordinator*, one or more *simulators* and, in the case of the main machine (i.e., processor 0), a *root coordinator*. The *ParallelMainSimulator* is in charge of creating the structure of DEVS processors as shown in the previous chapter. To do so, *ParallelMainSimulator* calls the method *LogicalProcess.registerObject(p)* for each DEVS processor *p* that runs on this node. After every object is registered, *LogicalProcess.allRegistered()* is executed.

Simulation objects sharing a *ParallelMainSimulator* also share its *GVTManager* (in charge of calculating the global virtual time), *CommManager* (dealing with inter-LP communication), and *Scheduler* (in charge of scheduling the events received in the queue). More information on these classes can be found in [War04].

Each *ParallelMainSimulator* can access information about the *root coordinator* (which may be running locally –if this is the main machine- or remotely), the *node coordinator* and the *flat coordinator* running on this processor. In addition, information about all *node coordinators* can be accessed through a *nodeCoordinatorList*. Local simulation objects can be accessed directly via *LogicalProcess.simArray*.

ParallelMainSimulator has several methods to load all the information about the models (e.g., *loadPorts(), loadComponents(), loadModel(), loadLinks()*) and about the machines and model partition (*loadMachines(), modelPartition()*).

Warped provides different functions for manipulating elements in their queues (e.g., garbage collection, finding and inserting an element). A garbage collection mechanism is triggered by the kernel to release memory allocated by states and events that are no longer valid. For more information on these Warped features, see [War04].

Output files generated by an application based on Warped need to use a special Warped class, *FileQueue*, to perform output operations. At the beginning of the execution, the application has to inform the number of files that will be created. When using *FileQueue*, information is physically written to the file only when it is safe to do so (i.e., when it is impossible to have a rollback for that data). Uncommited data can be

rolled back by Warped if necessary. Warped kernel is in charge of flushing and closing physical files and deallocating memory.

There are two types of files written by CD++ as the simulation advances, which use the *FileQueue* mechanism provided by Warped: output files and log files.

Outputs generated by the model are written by CD++ in the output file. When we described the algorithms of *root coordinator* and *node coordinator* in the previous chapter, we discussed two different alternatives for generating models' outputs (see parameter *send-outputs-from-NC* in the *node coordinator*'s algorithms and *sendOutsFromNC()* in Figure 31)

One approach is to make *root coordinator* handle all outputs to be sent to the environment. Thus, *node coordinators* forward their outputs to *root coordinator*, which acts as an intermediator. A different possibility is to have *node coordinators* sending outputs directly to the environment. These two alternatives are implemented in CD++ with a *FileQueue* object in the *root coordinator* or with a *FileQueue* object for each *node coordinator* to handle outputs.

5.1 EXECUTION OF DEVS AND CELL-DEVS MODELS

Users must define different files to run DEVS and Cell-DEVS models in CD++. The minimum information that has to be specified is the model and its partition. For DEVS models, atomic and coupled models have to be defined. Atomic models are written in C++, whereas coupled models are specified using a built-in language. Cell-DEVS models are written using a built-in language, which allows specifying the size and

structure of the cell space, connection with other existing DEVS models, type of delay, neighborhood, border and rules for each cell (or region of cells). A partition file is used to specify how models will be distributed across the machines that participate in the simulation.

As we discussed earlier, our new simulator does not modify any class of the CD++ model hierarchy from the one that was introduced by [Tro01a]. Therefore, DEVS and Cell-DEVS models written for the previous version of parallel CD++ can be executed with our simulator without modifications. DEVS models written for other versions of the tool require minimum modifications in order to be executed by the new simulator.

Users can specify other optional information. When running a Cell-DEVS model, users can indicate the initial values for the cells, and log files to store the debugging of the model's rules. For DEVS and Cell-DEVS models, it is possible to specify external events that will be received by the model in an event file. External events are received via the model's input ports and times are written in the *hours:minutes:seconds:milliseconds* format. A sample event file is shown in Figure 34. For example, the first line shows an external event arriving at 00:00:04:000 via port *in_1* with a value of 1.

```
00:00:04:000    in_1     1
00:00:12:000    in_2     1
00:00:27:000    in_1     21
00:00:53:000    in_2     10
...
```

Figure 34: Sample CD++ event file

CD++ simulation messages can be logged either for debugging purposes or for

studying the internal behavior of the simulator. Users can specify only a subset of messages to be logged (e.g., it is possible to record only output, external, and done messages, while omitting initialization, internal, and collect messages).

```
3 / L / I / 00:00:00:000 / NodeCoordinator(450) / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,0)(26) / 00:00:00:000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,1)(27) / 00:00:00:000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,2)(28) / 00:00:00:000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,3)(29) / 00:00:00:000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,4)(30) / 00:00:00:000 / FlatCoordinator(454)
...
3 / L / D / 00:00:00:000 / life(4,4)(45) /00:00:00:000/ FlatCoordinator(454)
3 / L / @ / 00:00:00:000 / ParallelNodeCoordinator(450) / FlatCoordinator(454)
3 / L / Y / 00:00:00:000 / life(0,0)(26) / out / 0.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,0)(26) / 00:00:00:000 / FlatCoordinator(454)
3 / L / Y / 00:00:00:000 / life(0,1)(27) / out / 0.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,1)(27) / 00:00:00:000 / FlatCoordinator(454)
3 / L / Y / 00:00:00:000 / life(0,2)(28) / out / 0.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,2)(28) / 00:00:00:000 / FlatCoordinator(454)
3 / L / Y / 00:00:00:000 / life(0,3)(29) / out / 0.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,3)(29) / 00:00:00:000 / FlatCoordinator(454)
3 / L / Y / 00:00:00:000 / life(0,4)(30) / out / 0.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,4)(30) / 00:00:00:000 / FlatCoordinator(454)
3 / L / Y / 00:00:00:000 / life(1,0)(31) / out / 0.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(1,0)(31) / 00:00:00:000 / FlatCoordinator(454)
...
3 / L / * / 00:00:00:000 / ParallelNodeCoordinator(450) / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,0)(26) / ... / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,1)(27) / ... / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,2)(28) / 00:00:00:100 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,3)(29) / ... / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(0,4)(30) / ... / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(1,0)(31) / 00:00:00:100 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(1,1)(32) / ... / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(1,2)(33) / ... / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(1,3)(34) / 00:00:00:100 / FlatCoordinator(454)
3 / L / D / 00:00:00:000 / life(1,4)(35) / 00:00:00:100 / FlatCoordinator(454)
...
3 / L / D / 00:00:00:000 / life(4,4)(45) / 00:00:00:100 / FlatCoordinator(454)
3 / L / @ / 00:00:00:100 / NodeCoordinator(450) / FlatCoordinator(454)
3 / L / Y / 00:00:00:100 / life(0,2)(28) / out / 1.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:100 / life(0,2)(28) / 00:00:00:000 / FlatCoordinator(454)
3 / L / Y / 00:00:00:100 / life(1,0)(31) / out / 1.000 / FlatCoordinator(454)
3 / L / D / 00:00:00:100 / life(1,0)(31) / 00:00:00:000 / FlatCoordinator(454)
...
3 / L / * / 00:00:00:100 / NodeCoordinator(450) / FlatCoordinator(454)
3 / L / D / 00:00:00:100 / life(0,2)(28) / ... / FlatCoordinator(454)
...
```

Figure 35: *flat coordinator* log file for a sample Cell-DEVS model (partial)

Figure 35 shows an example of a log file generated by a *flat coordinator* during the execution of a bidimensional Cell-DEVS model called "life". The execution of this model is distributed across 4 machines, each of which runs a 5x5 area of the model. Log files keep track of the messages received by each processor participating in the simulation. In Figure 35, we see some of the messages received by a *flat coordinator*. Every entry in the log file includes: the machine in which the DEVS processor is running (in this case, 3), the type of message (*L* for local, which indicates that the source is running in the same LP, or *R* for remote, which indicates an inter-LP message), timestamp, and some information about the message (e.g., time of next transition for a done message, or port and value for an output message). The file also gives information about the processor ids of source and destination of the message.

The first line in Figure 35 shows a (local) initialization message received by *flat coordinator* (with id 454) at time 00:00:00:000 from its parent *node coordinator* (with id 450). As we described in the previous chapter, a *flat coordinator* responds to an initialization message by forwarding it to all its children (this is not shown in Figure 35 but in each *simulator*'s log file, where the messages are received). Once the initialization function is executed for every cell, these *simulators* send done (D) messages informing the time of next change, as shown in the following lines (done messages at time 00:00:00:000 received from *life(0,0)(26)* to *life(4,4)(45)*).

After all done messages are received, the *flat coordinator* reports its time of next change (00:00:00:000) to its parent (which is not shown in Figure 35 but in *node coordinator*'s log file), completing the initialization phase for the *flat coordinator*.

After the initialization phase is completed, Figure 35 shows the beginning of a collect (@) phase by a message received at time 00:00:00:000 from *node coordinator*. This message is forwarded to *simulators*. Then, *simulators* respond with output (Y) messages (which contain a port, *out*, and a value associated to that port, in this case 1.000 or 0.000) and done messages.

Following the collect phase, an internal (*) message at time 00:00:00:000 is received at *flat coordinator* from *node coordinator*. As described by *flat coordinator*'s algorithm, this message is forwarded to the corresponding *simulators*. Their responses report the time for their next internal transition. For example, *life(0,0)* and *life(0,1)* inform that they do not have a scheduled internal transition (this is informed in the log file by a "..." in the field for time of next transition, which represents *inf* or infinity as discussed in Chapter 4). On the other hand, *life(0,2)* reports that it has a internal transition scheduled in 100 ms. These messages complete the simulation cycle for time 00:00:00:000. Subsequently, a new collect message is received at time 00:00:00:100 and the process described earlier is repeated.

The log file for the *simulator* running *life(0,2)* is shown in Figure 36. It shows the messages for one of the *simulators* that interact with the *flat coordinator* shown in Figure 35.

```
3 / L / I / 00:00:00:000 / FlatCoordinator(454) / life(0,2)(28)
3 / L / @ / 00:00:00:000 / FlatCoordinator(454) / life(0,2)(28)
3 / L / X / 00:00:00:000 / FlatCoordinator(454) / in_value / 0.000
...
3 / L / * / 00:00:00:000 / FlatCoordinator(454) / life(0,2)(28)
3 / L / @ / 00:00:00:100 / FlatCoordinator(454) / life(0,2)(28)
3 / L / X / 00:00:00:100 / FlatCoordinator(454) / in_value / 0.000
...
3 / L / * / 00:00:00:100 / FlatCoordinator(454) / life(0,2)(28)
...
3 / L / @ / 00:00:00:200 / FlatCoordinator(454) / life(0,2)(28)
3 / R / X / 00:00:00:200 / FlatCoordinator(454) / in_value / 0.000
...
3 / L / * / 00:00:00:200 / FlatCoordinator(454) / life(0,2)(28)
...
```

Figure 36: *simulator* log file for cell model *life(0,2)* (partial)

The first line in Figure 36 shows the initialization message received at time 00:00:00:000 from *flat coordinator*, which triggers the initialization function for this cell. The *simulator* responds by sending a done message with the time of next change, which can be seen in the fourth line in Figure 35. The second line in Figure 36 shows the reception of a collect message at time 00:00:00:000, which triggers the execution of the output function for this cell. The output and done messages sent by *life(0,2)* are received by the *flat coordinator*, and can be seen in Figure 35. Then, an internal message is received by the *simulator*, which triggers the internal transition function for this model. Then, a new simulation cycle starts with the reception of a collect message at time 00:00:00:100, repeating the previous process.

We have discussed the log files generated by Cell-DEVS models. Log files generated by DEVS model are very similar. For example, Figure 37 shows a log file generated by a *simulator* in charge of an atomic model called *rtc4*.

```
2 / L / I / 00:00:00:000 / FlatCoordinator(22) / rtc4(01)
2 / L / @ / 00:00:00:000 / FlatCoordinator(22) / rtc4(01)
2 / L / * / 00:00:00:000 / FlatCoordinator(22) / rtc4(01)
2 / L / @ / 00:00:08:000 / FlatCoordinator(22) / rtc4(01)
2 / L / * / 00:00:08:000 / FlatCoordinator(22) / rtc4(01)
2 / L / @ / 00:00:15:000 / FlatCoordinator(22) / rtc4(01)
2 / L / * / 00:00:15:000 / FlatCoordinator(22) / rtc4(01)
...
```

Figure 37: *simulator* log file for a sample atomic model (partial)

Messages received by this *simulator* (whose id is 01) are shown in Figure 37, starting with a initialization message from *flat coordinator*. This message triggers the initialization function for the associated atomic model, *rtc4*, and a done message is sent to its parent, *FlatCoordinator(22)*, informing the time of next change. This done message, which is registered in the *flat coordinator*'s log file, indicates that the model has an internal transition function scheduled for time 00:00:00:000. Then, collect and internal messages are received, completing the simulation cycle for this time. The next simulation cycle starts again with a collect message at 00:00:08:000, followed by an internal message.

Figure 38 shows a log file for a *node coordinator* in charge of LP 0 during a simulation distributed in 3 processors.

```
0 / L / I / 00:00:00:000 /RootCoordinator(00)/NodeCoordinator(21)
0 / L / D / 00:00:00:000 /FlatCoordinator(24)/00:00:00:000/NodeCoordinator(21)
0 / L / D / 00:00:00:000 /FlatCoordinator(24)/00:00:00:000/NodeCoordinator(21)
0 / L / D / 00:00:00:000 /FlatCoordinator(24)/00:00:00:100/NodeCoordinator(21)
0 / L / D / 00:00:00:100 /FlatCoordinator(24)/00:00:00:000/NodeCoordinator(21)
0 / L / D / 00:00:00:100 /FlatCoordinator(24)/00:00:00:400/NodeCoordinator(21)
1 / R / X / 00:00:00:150 /NodeCoordinator(22)/in_port_1/0.000
...
```

Figure 38: *node coordinator* log file (partial)

Figure 38 shows the messages received in a *node coordinator* (whose id is 21) from other DEVS processors. An initialization message from the *root coordinator* is received at time 00:00:00:000. After forwarding this message, the corresponding done message from *flat coordinator* (line 2). Then, the *node coordinator* receives done messages in relation to the collect and internal messages sent at time 00:00:00:000 and equivalent messages for time 00:00:00:100. At time 00:00:00:150, an external message is received from a sibling *node coordinator* (its id is 22) from machine 1. The second field of the message indicates that it is a remote message, and its port (*in_port_1*) and value (*0.000*) are included in the log.

Log files generated by Cell-DEVS models can be used to visualize the results of the simulation. CD++ supports 2D and 3D visualization using different shapes and colors to represent each cell. For more information on how to visualize Cell-DEVS models in CD++, see [Wai03].

We described the log files generated by Cell-DEVS and DEVS models, which show the messages exchanged by DEVS processors. In addition to log files, as we discussed earlier, CD++ models can generate a file that registers all the outputs sent to the environment, as shown next.

```
00:00:08:000    out     1
00:00:15:000    out     2
00:00:48:000    out     3
00:01:03:000    out     4
00:01:17:000    out     5
...
```

Figure 39: Sample output file for a DEVS model

Figure 39 shows an output file generated by a sample DEVS model. The output file indicates the time, port, and value of each output. For example, the first line represents an output generated at time 00:00:08:000 through the output port *out* with a value of 1.

Chapter 6: PERFORMANCE ANALYSIS

In this chapter, we introduce a new synthetic benchmark devoted to automate the evaluation of DEVS-based simulation approaches called DEVStone. DEVStone assists the task of analyzing the performance of a simulator by generating models with different size, complexity and behavior, resembling different kinds of real world applications. We use DEVStone to analyze the overhead of our new simulator, comparing it with other engines supported by CD++. Then, we analyze the performance of our simulator for Cell-DEVS models.

6.1 DEVSTONE

Analyzing the performance of a simulation engine can be a very complex task. Users can create a wide variety of models with different structures, levels of complexity and degrees of interaction. Most studies of simulation techniques are focused on specific tools. For instance in [Tro01b], the authors presented performance studies of Cell-DEVS models in a parallel simulation environment. In [Zei96], the authors focused on a watershed model to show performance improvements in parallel and distributed architectures. A comparison of performance issues for two particular simulators (DEVSCluster and D-DEVSim++) is given in [Kim04]. DEVS was shown to be more efficient than the continuous counterparts when simulating natural [Zei97a], and artificial systems, such as a photovoltaic system [Fil02b]. However, those studies do not provide a

thorough analysis for the execution of models with different characteristics, neither do they give a common metric to compare results among different DEVS simulators.

Instead of limiting our effort solely to testing individual models, we developed a synthetic benchmark to aid not only this but also future initiatives in the area, as ongoing developments intended to improve DEVS simulators also require a way to assess performance. We introduce the **DEVStone** benchmark, a synthetic model generator that automatically creates models according to our goals. Its accuracy relies on the execution of a large pool of models to provide a robust test set. DEVStone generates models with different size, complexity and behavior, resembling different kinds of real world applications. Hence, it is possible to analyze the efficiency of a simulation engine with relation to the characteristics of a category of models of interest. The tool can be used to assess the efficiency of DEVS simulation engines, and it provides a common metric to compare the results using different tools.

We focus in the aspects of the models that have impact on performance, namely size of the model and the workload carried out in the transition functions. A DEVStone generator produces models using the following parameters:

- type: different structure and interconnection schemes between the components.
- depth: the number of levels in the modeling hierarchy.
- width: the number of components in each intermediate coupled model.
- internal transition time: the execution time spent by internal transition functions.
- external transition time: the execution time spent by external transition functions.

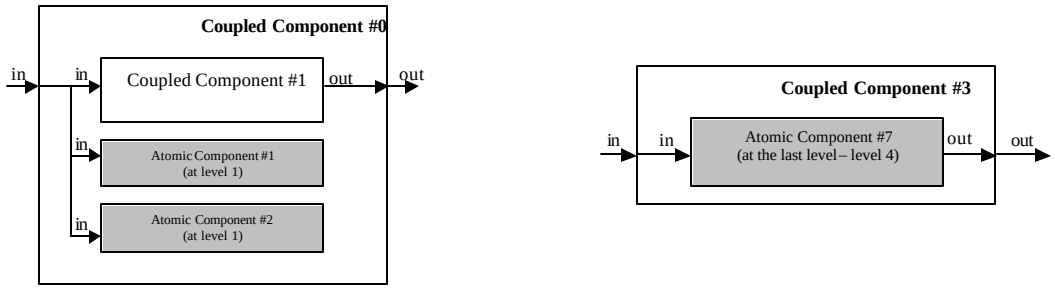
In general, being *d* the depth and *w* the width, we build a coupled model with *d* levels in the hierarchy, all of which consist of *w-1* atomic models, with the exception of the lowest level of the hierarchy, in which a coupled component is composed of a single atomic model. In addition, internal and external transition functions are programmed to execute a fixed amount of time specified by the user. In both transition functions we consume CPU clocks by running Dhrystones [Wei84]. The Dhrystone synthetic benchmark uses published statistics on the use of programming language features, and it is available for different programming languages (Ada, C++, Java, Pascal, etc.). Dhrystone code consists of a mix of instructions using integer arithmetic; therefore, it is a good choice for analyzing models like DEVS in which state variables have discrete values.

DEVStone uses three different types of models with variations in their internal and external structure:

- **LI** models, with a low level of interconnections for each coupled model
- **HI** models with a high level of input couplings, and
- **HO** models with high level of coupling and numerous outputs.

In **LI** models, every coupled component includes only one input and one output port. Figure 40 shows a sample LI model, in which we have four layers of coupled components, each containing three submodels. The arrows represent input and output ports, solid-white boxes represent coupled components and shaded-gray boxes represent atomic components. The *Coupled Component #0* in Figure 2 (a) consists of one coupled and two atomic components. The lower levels in the hierarchy (*Coupled Component #1,*

Coupled Component #2) use the same internal structure. *Coupled Component #3* is a "leaf" model, which contains one atomic child (#7).



(a) (b)

Figure 40: Example of a LI model: (a) top level; (b) level 4

```
[top]
components:  comp0 comp01@Atom comp02@Atom
out : out
in : in
link : in in@comp0
link : in in@comp01
link : in in@comp02
link : out@comp0 out

[comp0]
components : comp1 comp11@Atom comp12@Atom
out : out
in : in
link : in in@comp1
link : in in@comp11
link : in in@comp12
link : out@comp1 out

...

[comp01]
preparation : 00:00:00:000
intDelay : 0
extDelay : 0

[comp02]
preparation : 00:00:00:000
intDelay : 0
extDelay : 0

...
```

Figure 41: Model file generated by DEVStone for a LI model

Figure 41 shows the model file associated with such model, which is generated by DEVStone. As we discussed earlier, coupled models are entirely defined in this file (their components, internal coupling, etc.). Although atomic components require a separate C++ file, a section of the model file is used to define their preparation time, and two parameters (*intDelay* and *extDelay*) that determine the internal and external transition time.

As we know the model structure and the time spent by each component in executing transition functions, we can compute the execution time for the model analytically. First, we devise the number of atomic and coupled models in the structure, which can be derived from the composition of the model type. In LI models of depth *d* and width *w*, we have *d* coupled models with *w-1* atomic components each (except for the leaf, with only one atomic component). Consequently, the total number of atomic components is:

$$\#\ \text{Atomic Models} = (\text{width} - 1) * (\text{depth} - 1) + 1$$

Since we have a predefined interconnection pattern, we can anticipate the message routes triggered by an external event and the time spent in transition functions. LI models forward external events to each atomic component and to lower levels in the model hierarchy. Each external event triggers the atomic's external transition function and, subsequently, an internal transition is scheduled. Thus, the number of internal and external transition functions to be triggered is:

$$\begin{aligned} \#\ \textit{Internal Transitions} &= \#\ \textit{Atomic Models} \\ \#\ \textit{External Transitions} &= \#\ \textit{Atomic Models} \end{aligned} \qquad (1)$$

HI models have the same number of atomic components, but more interconnections. Each atomic component *k* connects its output to the input port of component *k+1* (with the exception of one last atomic component on each level, which does not have any output port). Therefore, there are more messages exchanged upon the reception of an external event, and the associated overhead grows accordingly. In a model with depth *d*, and width *w*, we have,

$$\begin{aligned} \#\ Atomic\ Models &= (w-1) * (d-1) + 1 \\ \#\ Internal\ Transitions &= \sum_{i=1\,..\,w-1} i * (d-1) + 1 \\ \#\ External\ Transitions &= \sum_{i=1\,..\,w-1} i * (d-1) + 1 \end{aligned} \qquad (2)$$

Each coupled model forwards the external events to its *w-1* atomic children and also to its coupled child. This process of forwarding messages is repeated in each coupled component except for the leaf component, which forwards the messages to its single atomic child.

HO models have the same number of atomic and coupled components, but coupled models have two input and two output ports in each level. The second input port in the coupled component is connected to its first atomic component. That atomic model connects its output to the second output of its parent. The increased number of interconnections results in the execution of more transition functions after the model issues its output, and consequently generates more overhead. For this model type we have,

$$
\begin{aligned}
\#\ Atomic\ Models &= (w-1) * (d-1) + 1 \\
\#\ Internal\ Transitions &= S_{(i=1\ ..\ w-1)}\ i * (d-1) + 1 \\
\#\ External\ Transitions &= S_{(i=1\ ..\ w-1)}\ i * (d-1) + 1
\end{aligned}
\tag{3}
$$

Coupled components forward each external event to their *w-1* atomic children and also to their coupled child. This process is repeated for each coupled model until the leaf component receives the event, which is forwarded to its single atomic component.

DEVStone can be used in any simulator with capabilities for defining and executing Dhrystone code. We can use single-layered models for comparison with tools with non-hierarchical structure. Likewise, if the chosen modeling technique does not support the execution of internal transitions, we can compare the simulators by building a DEVStone in which the execution time for internal transitions is zero.

6.2 PERFORMANCE ANALYSIS FOR DEVS MODELS

CD++ supports different simulation techniques, some of which were discussed earlier in this work. The original version of CD++ provides a stand-alone engine for execution on a single processor [Rod99, Wai02]. In [Tro01a, Tro03], a parallel version of the toolkit was presented. It uses a conservative synchronization protocol, as opposed to the simulation technique based on an optimistic synchronization protocol introduced in this work.

Our first goal is to determine the overhead of the new simulation engine. To analyze its overhead, we use our DEVStone synthetic benchmark. Moreover, we compare the overhead of the new engine with the overhead of the previous implementations.

The advantages of taking this approach are twofold. First, the execution of these experiments allows us to test the usefulness of the DEVStone benchmark. Second, we can test our new simulator thoroughly, and compare the results with engines that have shown good performance results for DEVS and Cell-DEVS execution [Gli02b, Tro01b].

The following tests compare the overhead of three simulators: (i) original, (ii) parallel simulator with conservative protocol, and (iii) our new simulator, which implements parallel simulation using an optimistic protocol. In addition, we compare the execution results with the theoretical execution time for each type of model, computed as in equation (4).

$$\text{Total theoretical time} = [\,(\#\ \textit{External Transitions} * \textit{TimeInExternalTransition}\,) + (\#\ \textit{Internal Transitions} * \textit{TimeInInternalTransition})\,] * \textit{NumberOfExtEvents} \quad (4)$$

In this set of experiments, we are focused on measuring the overhead of the new simulator and comparing the results with other (stand-alone and parallel) engines. Thus, these simulations are executed on a dedicated single-processor machine. All models were executed using 10 external events, each of them triggering a known number of external and internal transition functions defined by equations (1), (2), and (3). Table 1 shows the parameters we used for different tests, including model type, structure and time spent on

transition functions (*e.g.*, model E is of HI type, it is composed of 3 levels, and has 6 components per level).

Table 1: Simulation parameters

Simulation	**Model Type**	**Depth**	**Width**	d_{int}	d_{ext}
A	LI	3	10	50 ms	50 ms
B	LI	10	3	50 ms	50 ms
C	LI	5	5	50 ms	50 ms
D	LI	10	10	50 ms	50 ms
E	HI	3	6	50 ms	50 ms
F	HI	6	3	50 ms	50 ms
G	HI	5	5	50 ms	50 ms
H	HI	6	6	50 ms	50 ms
I	HO	3	6	100 ms	0 ms
J	HO	6	3	0 ms	100 ms
K	HO	5	5	50 ms	50 ms
L	HO	6	6	50 ms	50 ms

The experiments were executed in a **single processor**, allowing us to measure the pure overhead incurred by our simulator, and enabling comparisons not only with the other parallel (conservative) simulator but also with the original (stand-alone) simulation engine.

In order to better understand the influence of the tools in the total execution time, we also measured the percentage of overhead. The overhead is computed as the ratio between theoretical and actual execution time.

The following figures show the execution times and the associated overheads grouped by model type: Figure 42 and Figure 43 (LI models), Figure 44 and Figure 45 (HI models) and Figure 46 and Figure 47 (HO models).

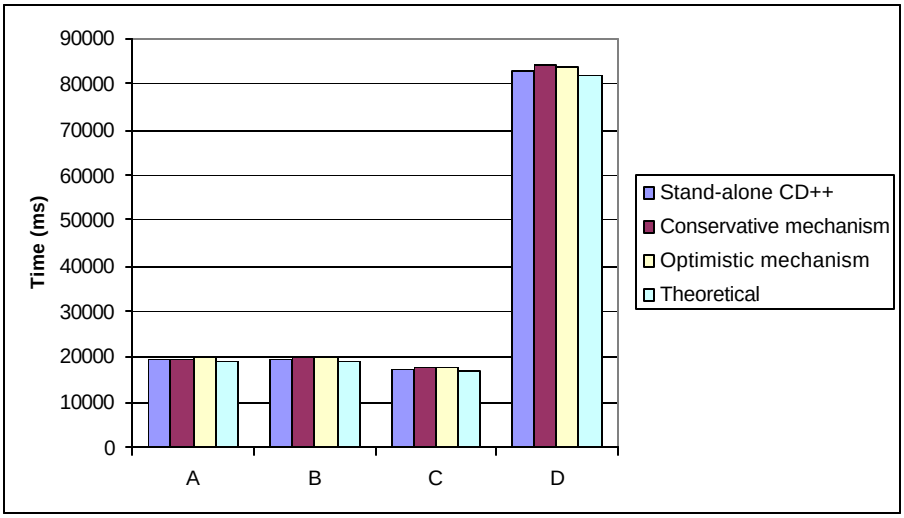


Figure 42: Execution times for LI models in a single CPU using the optimistic parallel simulator and other simulation engines

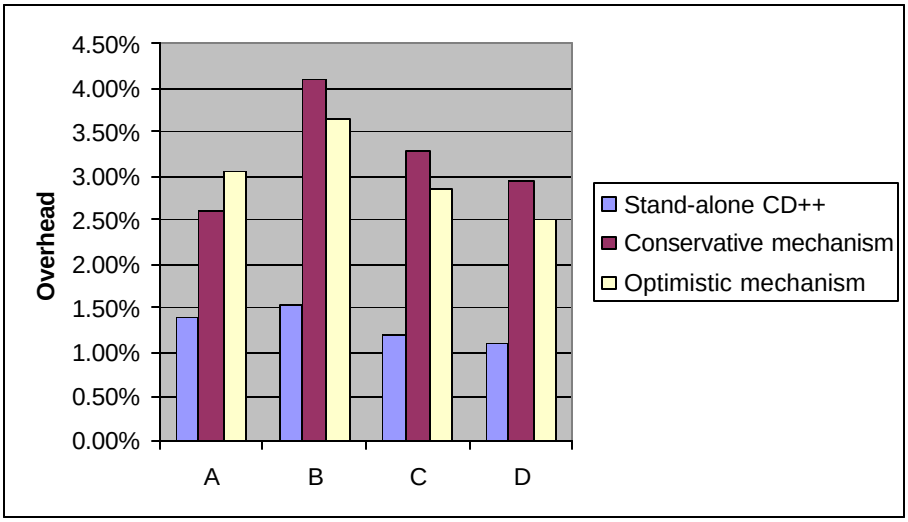


Figure 43: Overhead incurred by the optimistic parallel simulator and other simulation engines for LI models

Figure 42 shows the execution times for LI models, which belong to the most simple type of models generated by DEVStone. We can measure the difference between the theoretical execution time (which only comprises time required to execute the Dhrystone code in the internal and external transitions) and the execution time for each engine. As a result of the relatively simple structure of models A, B, C, and D, the differences are small; in all cases, the differences fall in the range of 270 to 2110 ms. As expected, the smallest difference between theoretical and execution time is observed when executing model A (with a structure of 3x10), which is the smallest and most simple model in the test set. The overhead for executing such a model is 1.40% (for the stand-alone version), 2.61% (for the conservative parallel version) and 3.06% (for the optimistic parallel version). On the other hand, the largest overheads are observed for model D (10x10), which contains more than 80 models in its structure. For model D, the overhead for each simulator is 1.10%, 2.95%, and 2.51%, respectively. In all cases the overhead is kept below 5%.

For all LI models, the stand-alone outperforms both parallel alternatives. For models B, C, and D, the optimistic engine outperforms the conservative one, whereas for model A the conservative engine outperforms the optimistic one. However, we believe that the results obtained for the optimistic simulator are very promising in terms of performance. As we mentioned earlier, these simulations are executed in a single processor, and therefore it was expected that the stand-alone engine would outperform the optimistic approach. Moreover, we discussed that the implementation of the optimistic simulator is more complex and has more tasks that take place at each

simulation cycle (e.g., associated with the synchronization mechanism and the mechanism for saving states, input, and output queues). Although the overhead associated with those tasks can be considerable, the optimistic simulator still outperformed the conservative simulator for models B, C, and D. This is a consequence of the reduction in communication overhead incurred by the flat approach (implemented in the optimistic simulator) over the hierarchical case (implemented in the conservative simulator). As we discussed in Chapter 4, the flat simulator transforms the hierarchical structure of a DEVS model into a more simple, flat structure of DEVS processors. We discussed the potential for reducing the number of messages exchanged in the flat mechanism by comparing both approaches. The execution of models B, C, and D show that the performance gains of using a flat simulator outweigh the increased overhead associated with the optimistic simulator. This is not the case for model A, where the hierarchical, conservative engine still performs better than the flat, optimistic engine. We believe this is a consequence of the structure (3x10) of model A. Its structure is wide (10) but not very deep (3 levels), and therefore the reduction in messages exchanged when using a flat approach is not that important. A more detailed discussion on these topics is given later.

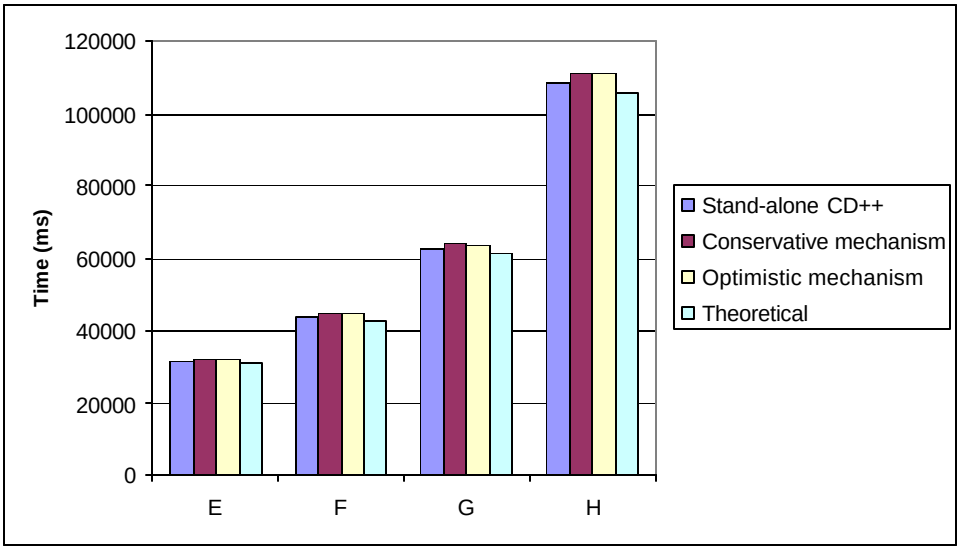


Figure 44: Execution times for HI models in a single CPU
using the optimistic parallel simulator and other simulation engines

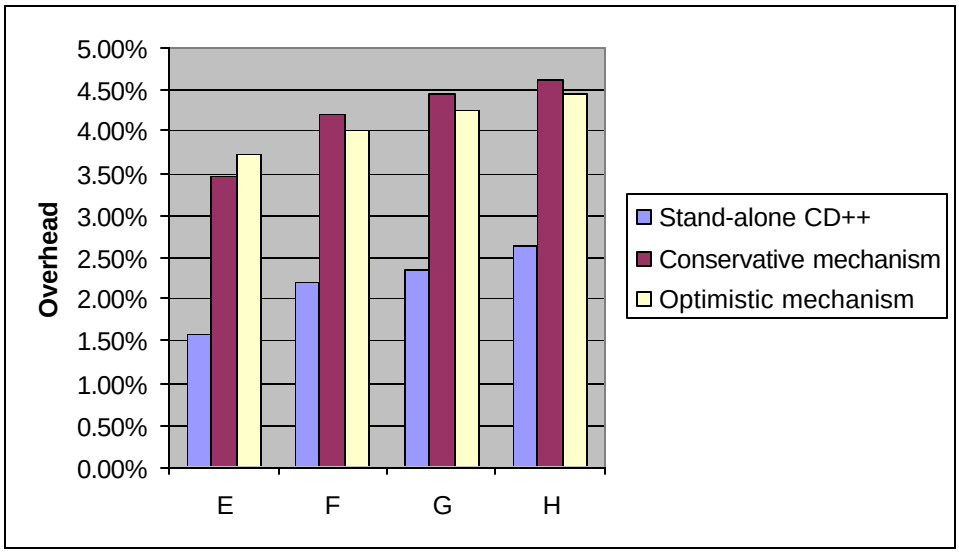


Figure 45: Overhead incurred by the optimistic parallel simulator
and other simulation engines for HI models

Figure 44 and Figure 45 illustrate the results for executing HI models, which are more complex than LI models. As we mentioned before, HI models have more interconnections between inner components, which results in more transitions functions

to be executed. In these cases, we observe results that are similar to those obtained for LI models. Differences between theoretical and actual execution times fall in the range of 500 to 5130 ms, and overheads are in the range of 1.59% to 4.62%. In general we see that the stand-alone simulator, whose overhead is 2.65% in the worst case (model H, 6x6), outperforms both parallel alternatives for all HI models. The optimistic engine outperforms the conservative simulator for models F, G, and H, whereas the opposite is observed for model E. As we discussed for LI models, the flat simulator implemented in the optimistic engine outweighs, in some cases, the increased overhead associated with its more complex implementation. However, this is not the case for model E, whose structure is wide, but not very deep.

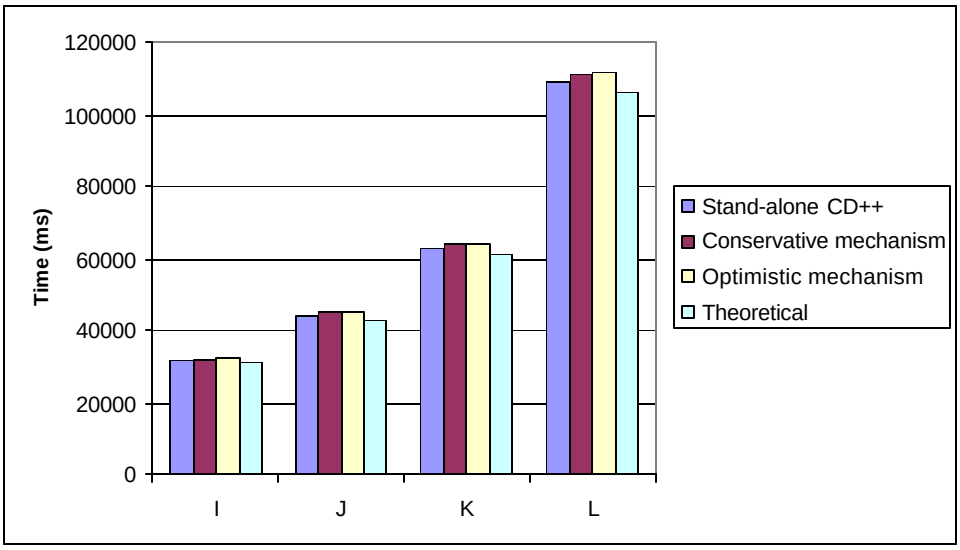


Figure 46: Execution times for HO models in a single CPU
using the optimistic parallel simulator and other simulation engines

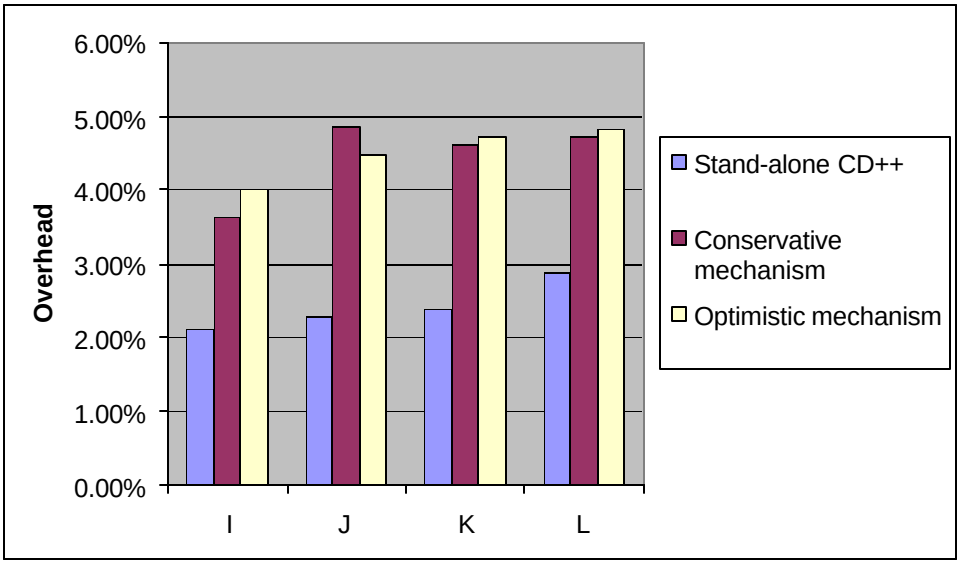


Figure 47: Overhead incurred by the optimistic parallel simulator and other simulation engines for HO models

Finally, Figure 46 and Figure 47 show the execution times and associated overhead of HO models, which have the most complex structure generated by DEVStone. The results illustrate the same trend shown earlier for LI and HI models. In this case, the models are more complex, execute more transition functions and generate more outputs. Consequently, the simulators have to perform more tasks and more messages are exchanged, which leads to larger differences. The differences between theoretical and actual execution times are in the range of 670 ms for the smallest model (I) and 5490 ms for the largest one (L). The stand-alone simulator also provides the best performance in all cases. The conservative simulator outperforms the optimistic one for models I, K, and L, whereas the reverse holds for model J. The structure for model J is fairly deep (6 levels) and not very wide (3 models per level). As we discussed for previous cases (e.g., models A and E), as a result of the more simple structure used by the flat mechanism to

simulate this model, the optimistic simulator outperforms the hierarchical, conservative one for executing model J.

The execution of LI, HI, and HO models gives us information about the execution performance for our new simulator, and a comparison with the previous alternatives available in CD++. In general, we see that the stand-alone simulator outperforms both parallel alternatives. This is a consequence of the more simple architecture and implementation of the stand-alone engine. As we have discussed earlier in this work, the parallel simulators are built on top of two layers of middleware (Warped and MPI). The use of these middleware associates more overhead at execution time, in particular for message passing. In addition, there is an overhead associated with the optimistic simulator. When the optimistic simulator is used, as we discussed in Chapter 4 and 5, the simulation objects save states, and input and output events to allow recovering from potential rollbacks in the future. The time spent on these tasks has an impact on the overall performance of the optimistic simulator.

We tried to reduce the communication overhead of the new simulator by implementing a flat approach. Our flat approach uses a flattened structure of DEVS processors to simulate the model, instead of a hierarchical structure which is usually more complex. In Chapter 4, we discussed the reduction in communication costs by comparing the number of exchanged messages incurred by our flat simulator with the number of messages exchanged by a hierarchical mechanism. Some of the results presented in this section substantiate our previous analysis. Although the optimistic simulator incurs in more overhead (associated with its more complex synchronization mechanism), in some

cases it still outperforms the conservative approach. More specifically, the new flat, optimistic simulator outperforms the conservative simulator for models B, C, D, F, G, H and J. On the other hand, for models A, E, I, K and L, the conservative engine outperforms the optimistic one despite the hierarchical approach, although the differences are relatively small. These experiments show that the gains obtained by using a flat approach compensate, and in some cases outweigh, the increased the overhead associated with the implementation of the optimistic simulator.

So far, we have studied the overhead incurred by the new flat, optimistic simulator using different DEVS models. Using DEVStone, we have compared its performance with other simulators whose performance has been analyzed and deemed appropriate [Gli02b, Tro01b]. Although in some cases it presented more overhead than other tools, the optimistic synchronization mechanism has the potential for enabling speedups in distributed environments.

6.3 PERFORMANCE ANALYSIS FOR CELL-DEVS MODELS

We studied the performance of our new simulator using a Cell-DEVS model based on the life game [Gar70]. This popular game consists of a bidimensional lattice of cells. Based on a simple set of rules, cells can live, die, or multiply. Figure 48 shows the definition of the model in CD++.

```
[top]
components : life

[life]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1)  life(0,0)  life(0,1)
neighbors : life(1,-1)  life(1,0)  life(1,1)
localtransition : conrad-rule
neighborports : value

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

Figure 48: Specification of Cell-DEVS model life in CD++

Figure 48 shows the definition of the model as a 20x20 wrapped Cell-DEVS model with transport delays and 3x3 neighborhood. As described in Chapter 2, the behavior of each cell is defined by the rules of the model (see the section *conrad-rule*), which have a fixed form of *VALUE DELAY {CONDITION}*. If the *CONDITION* is satisfied, the cell state becomes *VALUE* and then it is *DELAY*ed for the specified time. The survival of a cell depends on the number of active cells within its neighborhood. If the number of active cells, determined by *statecount(1,~value)*, is three or four, then the cell remains alive (specified by the first rule), otherwise it dies (specified by the second rule). The third rule specifies that an inactive cell becomes active if the number of active

cells in its neighborhood is three. In this model, the delay is 100 milliseconds for every rule.

We executed the life game using different cell spaces: 16x16 (256 cells), 20x20 (400 cells), 25x25 (625 cells) and 30x30 (900 cells). The initial configuration of cells for each model was randomly generated.

We have experienced some problems when straggler messages are received by a simulation object during the execution of DEVS and Cell-DEVS models. More specifically, the simulation aborts during the rollback mechanism triggered after a straggler message is detected in the destination. We have identified the code of the Warped middleware that triggers this incorrect finalization of the simulation. Although this is out of the scope of this work, we are working to find a solution to fix this problem in the middleware. In order to carry out the experiments that allow distributing the simulation in more than one processor, and to enable a performance analysis of our new simulator, the following models are designed avoiding inter-LP communication. Since message exchange between LPs does not happen, it is not possible to receive straggler messages. Consequently, rollbacks are not possible and the simulation can finish without errors.

First, the models were executed on one and four machines. We used simple rectangular partitions for the distributed case. Figure 49 depicts the partition used for the 20x20 life model, where each machine executes a region of 10x10 (100 cells). Analogous partitions were used for the other cell spaces.

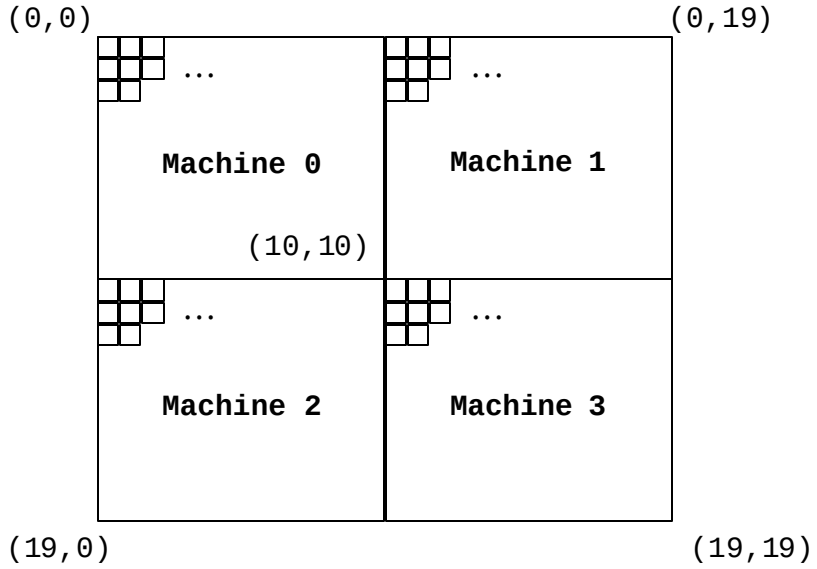


Figure 49: Partition of 20x20 life model in 4 machines

Figure 50 illustrates the execution times for the different configurations used for the life model.

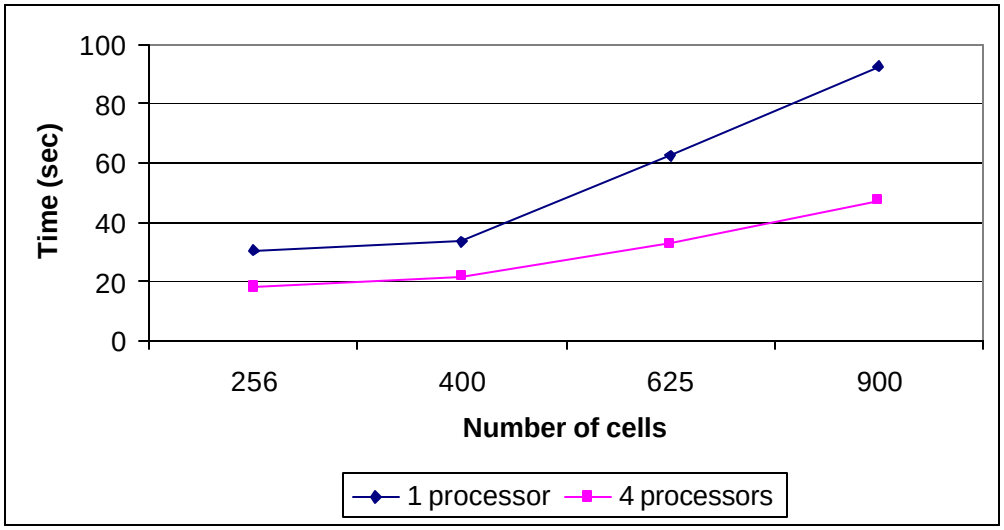


Figure 50: Execution times for life model (1 vs. 4 processors)

Figure 50 shows that, in all cases, the distributed execution of the model outperformed the execution in a single processor. The execution time for the model running on one processor varies from 30.7 to 90.8 seconds depending on the size of the model. On the other hand, when running the model in parallel on 4 machines, the execution time is smaller (between 18.1 and 47.5 seconds); in some cases, the optimistic simulator allows to reduce the execution time in ~50%. Moreover, Figure 50 shows that, as the size of the model increases, the slope for of the 4-processor simulations is less steep than the one for 1-processor simulations.

Recall from Chapter 4 and 5 that when using the distributed simulator presented in this work, scheduling of events and synchronization tasks are distributed among the *node coordinators* that participate in the simulation. When the simulation is executed on a single processor, one *node coordinator* and one *flat coordinator* handle the execution of all the cells in the model. Then it is that single *node coordinator* which centralizes all synchronization tasks and event scheduling for the entire model. Moreover, for the smallest model executed in this test, the *flat coordinator* may have to schedule messages (e.g., internal, collect) for up to 256 cells at every simulation cycle. Notice that the exact number of messages to be transmitted at each simulation cycle depends on the number of active cells in the model.

In contrast, when executing the model in *n* machines, a *node coordinator* and a *flat coordinator* are created on each logical process. Thus, each *node coordinator* handles synchronization tasks and scheduling of events for its own LP, and each *flat coordinator*

is in charge of the group of cells running locally. For the largest model shown in Figure 50, its *node coordinator* and *flat coordinator* handles 225 cells.

Figure 51 shows the execution speedup obtained by running the model in 4 processors. The execution speedup for *n* processors is measured as follows.

$$\text{Speedup} = \frac{\textit{execution time in 1 processor}}{\textit{execution time in n processors}} \qquad (5)$$

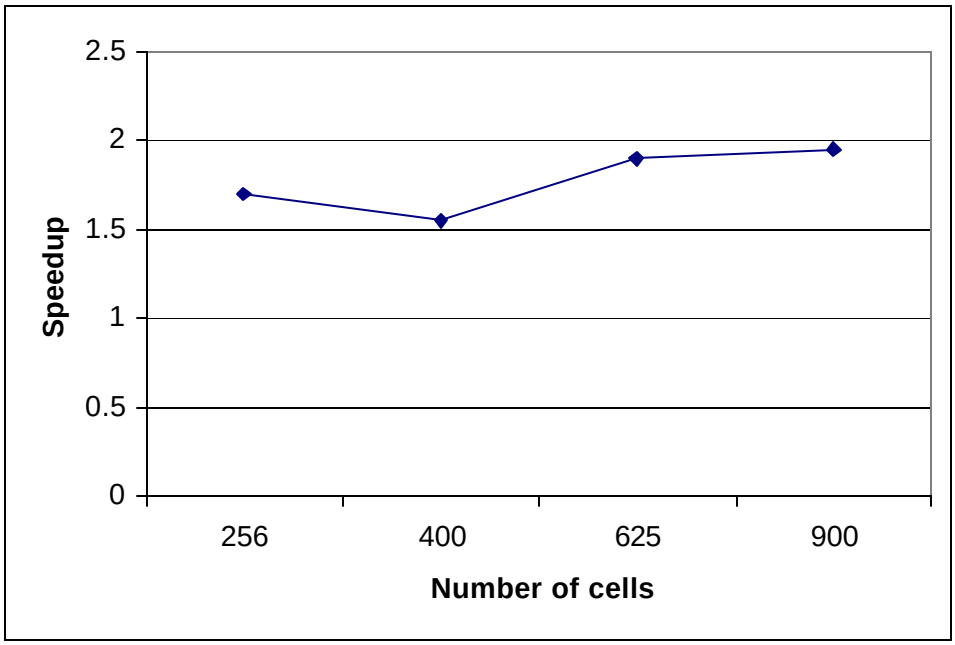


Figure 51: Execution speedups for life model running in 4 processors

Figure 51 shows that the factor of speedup falls between 1.55 and 1.95 when distributing the execution of the life model among 4 processors using this partitioning approach. In these cases, we observe that the increase in computing power obtained by using multiple machines is affected by the communication costs of synchronizing the

simulation. Moreover, the communication costs are more noticeable because the simulations are executed over a relatively slow network. The processors are connected via a 10 Mbit hub, which limits the simultaneous transfers rate to 10 Mbits per second. In addition, since these models are relatively small and do not have numerous active cells, the performance gains obtained by distributing these simulations are limited.

Figure 52 shows a comparison between our parallel simulator and the previous conservative simulator [Tro01a, Tro03] for different configurations of 30x30 (*life 1-4*) and 40x40 (*life 5-8*) models using 4 machines.

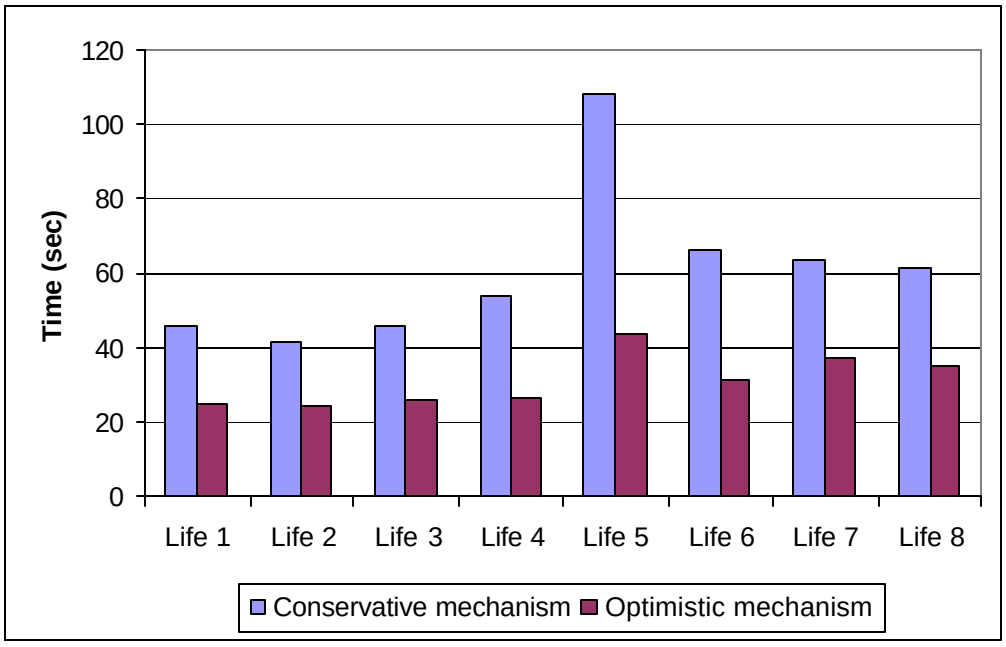


Figure 52: Execution times for life model
using optimistic and conservative simulators in 4 processors

Figure 52 shows that the optimistic simulator outperforms the conservative simulator for all configurations of 30x30 and 40x40 life models. In the configuration

labeled as *life 5* (a 30x30 model), most of the 900 cells are active in the first cycles of the simulation. In cases like this, we observe the the largest difference in execution times: 108 seconds for the conservative mechanism, 44 seconds for the optimistic one. In general, the difference is a result of the performance gains obtained not only by distributing the simulation in multiple processors but also by distributing the scheduling tasks in multiple *node coordinators*.

We are interested in analyzing the performance of our simulator for larger Cell-DEVS. The following figures show the execution times and speedups for different configurations of the life model with a cell space of 50x50 (i.e., 2500 cells). The differences among these configurations (labeled as *life A*, *B*, *C*, and *D* in Figure 53 and Figure 54) are the initial values used for the cells.

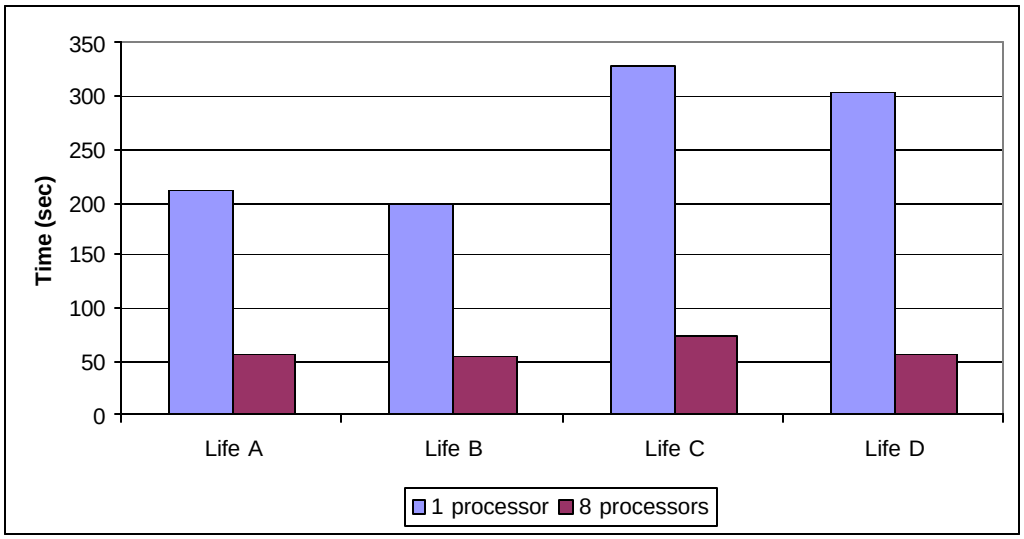


Figure 53: Execution times for 50x50 life model in 1 and 8 processors

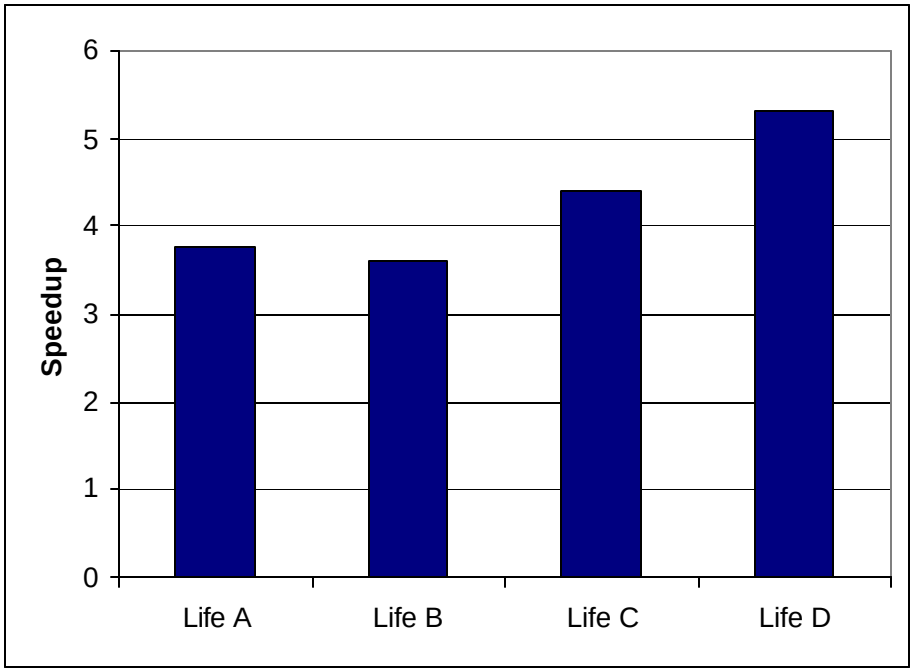


Figure 54: Execution speedups for 50x50 life model running in 8 processors

The execution times for these cases are significantly reduced when we distributed the simulation in 8 processors instead of using a single processor. In general, the distributed alternative achieves speedup factors in the range of 3.62 to 5.31, depending on the initial configuration.

When a 50x50 model is executed on a single processor, only one logical process is created. Hence, a single instance of a *flat coordinator* is in charge of the 2500 *simulators* participating in the simulation, and a single *node coordinator* is in charge of scheduling tasks for the entire model. Using a single machine for executing a Cell-DEVS model with this size results in a significant amount of memory needed to store data associated with the simulation (e.g., list of imminent components, port mapping for each model, pointers to each simulation object). But, more importantly, the execution of a

model with 2500 cells in a single processor has an impact on the time consumed for accessing this information. For example, consider the time required to update the list imminent components (i.e., models that are scheduled for a transition), which is maintained by a single *flat coordinator*. Similarly, consider the time consumed when retrieving the information associated with a simulator object (e.g., when a *flat simulator* has to find the destination of an output). In contrast, the distribution of this model in 8 machines allows a smaller structure associated with each logical process participating in the simulation. Each logical process has an associated *flat coordinator* and *node coordinator* that are in charge of 312 *simulators*. Figure 53 and Figure 54 show that distributing the simulation of a large model in 8 machines allows significant execution speedups.

A different partitioning approach is used for the following tests. The life model is executed in 1, 3, 4 and 5 processors using the same cell spaces (16x16, 20x20, 25x25, and 30x30). The idea of this partition approach is to divide the model in horizontal rectangles, as shown in Figure 55 for a 30x30 model partitioned among 3 machines.

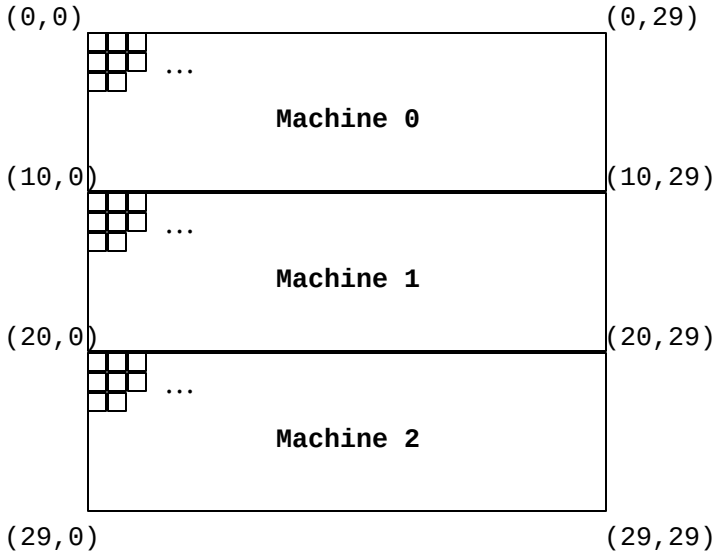


Figure 55: A different partition strategy for the life model

Figure 56 illustrates the execution times for the different configurations of the life model using 1, 3, 4, and 5 processors. Figure 57 illustrates the execution speedups.

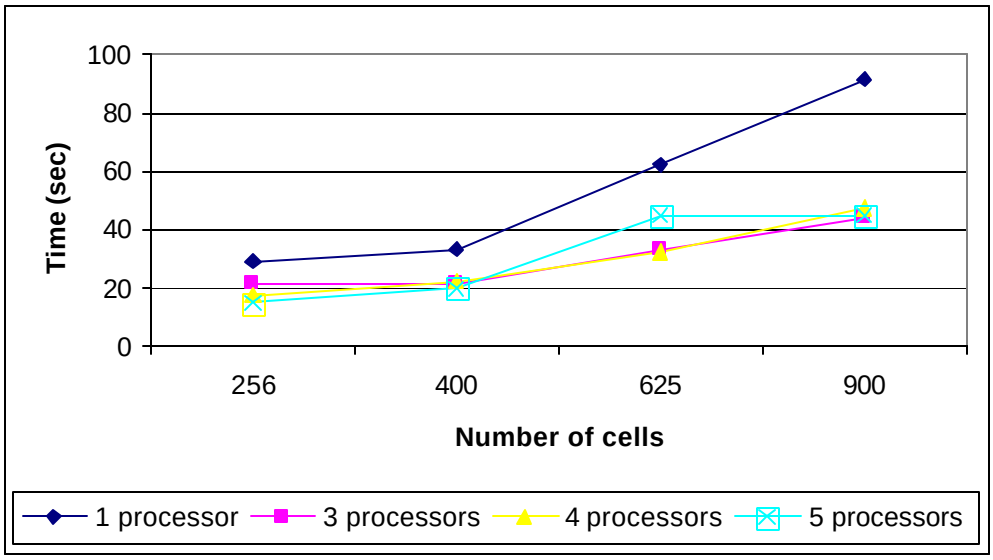


Figure 56: Execution times for life model using 1, 3, 4 and 5 processors

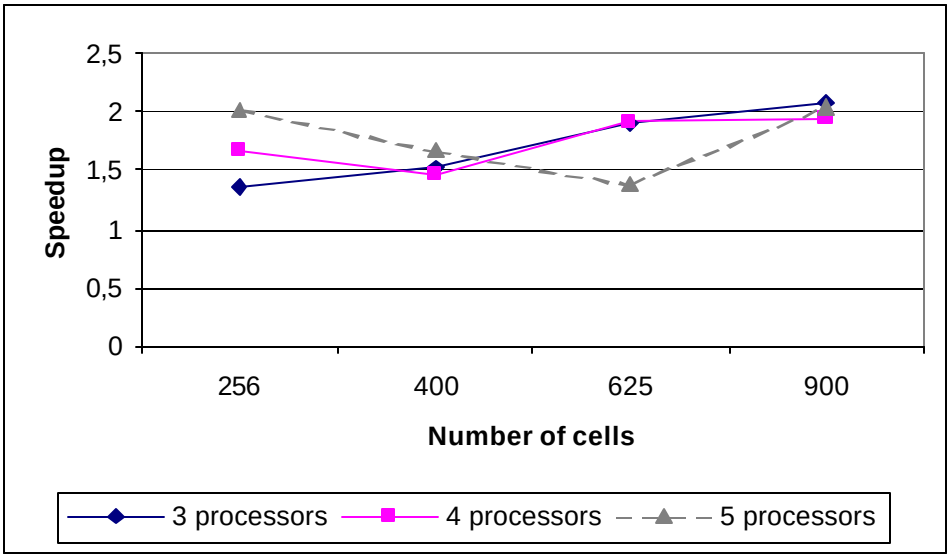


Figure 57: Speedups for life model distributed in 3, 4 and 5 processors

Figure 56 shows that the steepest slope is associated with one-processor executions. Distributing the simulation in 3, 4, or 5 processors reduces the execution time regardless of the size of the model. Figure 57 shows that the execution speedups for 3, 4 and 5 processors is approximately 1.4-2.1. As we discussed earlier, this speedup factor is a result of distributing a Cell-DEVS model that does not have numerous active cells. Therefore, the increase in computing power obtained by using more machines is, in some cases, outweighed by the communication costs of having to synchronize many logical processes. This factor becomes more noticeable when the simulations are executed in a relatively slow network like the one we used for these experiments. As we discussed before, the computers are connected using a 10Mbit hub which limits the simultaneous data transfer to 10 Mbits per second. For example, using 3 or 4 machines provides better performance than using 5 machines for the model with 625 cells. The best configuration

is hard to determine based on these runs. In some cases the minimum execution time was achieved by using 5 processors (256 and 400 cells), while in other cases it was achieved by 3 processors (900 cells) or 4 processors (625 cells).

The following set of tests uses a sample Cell-DEVS model to study the performance of models whose cells change frequently. The rules defining the behavior are simple: the current value of a cell changes from 0 to 1, and from 1 to 0, alternating at each simulation cycle. These rules produce changes for every cell at every simulation cycle. We execute models with 400 and 900 cells, using two different initial configurations for each case, labeled in Figure 58 as *models 1* to *4*.

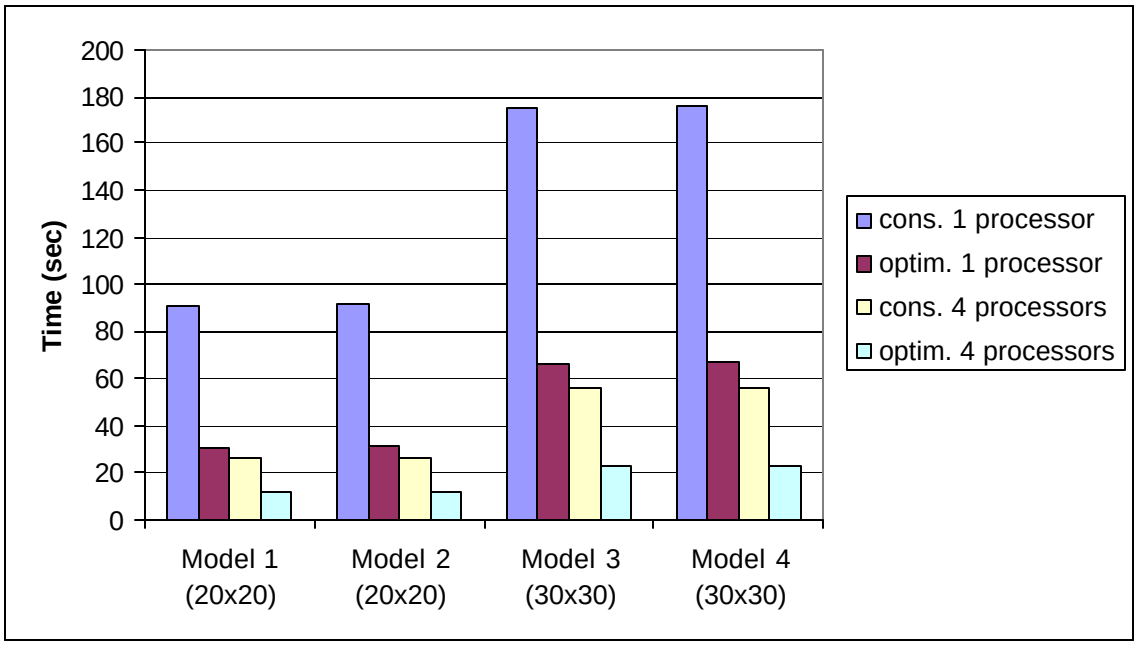


Figure 58: Execution times for Cell-DEVS model using conservative and optimistic simulators in 1 and 4 processors

Figure 58 shows that the simulation in 4 processors using the optimistic simulator achieves the best performance for all these cases. The conservative simulator distributed

in 4 machines outperforms its single-processor counterpart. The optimistic simulator running on a single machine achieves almost the same performance as the conservative simulator running on 4 processors, which shows the increased communication costs of the latter alternative and the good performance achieved by our simulator. Figure 59 shows the speedup of the optimistic simulator distributed in 1 and 4 processors in relation to the conservative simulator for the 20x20 and 30x30 models.

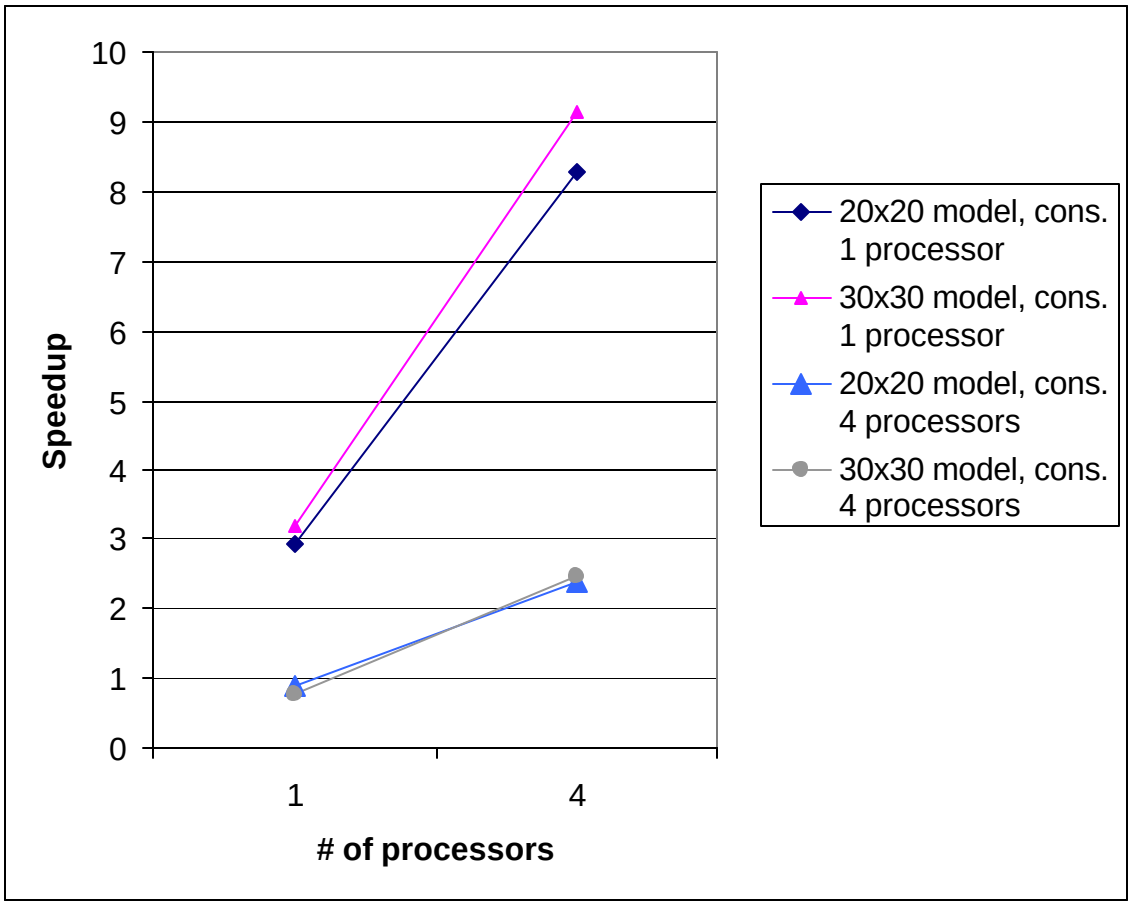


Figure 59: Speedup obtained by the optimistic simulator in 1 and 4 processors

Figure 59 illustrates the speedups obtained by our simulator using 1 and 4 processors in relation with the conservative simulator. The figure shows that the execution of the optimistic simulator in 1 processor allows significant speedups (2.91 for 20x20 models, 3.17 for 30x30 models) in comparison to the conservative simulator running on a single processor. The speedup factor obtained by executing the simulation in 4 machines using the optimistic approach instead of the equivalent partitioning for the conservative approach is approximately 2.45 for 20x20 and 30x30 models. The execution of the model using our approach in 4 processors enables speedup factors of up to 9.15 in comparison to the execution in a single-processor using the pessimistic technique. Although the execution of both 20x20 and 30x30 models using the pessimistic approach in 4 processors outperforms our simulator executing in 1 processor, it is only by a relatively small fraction (the speedup factor is .82-.86), showing the good performance of our simulator.

Other tests could be executed to better analyze the performance of the new simulator once some of the issues with Warped are solved. Some of these experiments include:

- Execution of life model using configurations that are more (or less) prone to having rollbacks in the participating nodes. This would allow determining the impact on performance of executing models where rollbacks are more (or less) frequent.
- Execution of Cell-DEVS models where the interaction between neighboring cells is frequent. For example, a model of bacteria reproduction (discussed in

[Ame03]) allows studying the propagation of a marine germ over a surface. The rules that describe the concentration of bacteria require frequent interaction with the neighborhood and would enable a characterization of performance for similar models.

- Execution of Cell-DEVS models where there is seldom interaction between neighboring cells. The behavior of people in a metro station, described in [Ame03], has these characteristics. As it was suggested earlier, this would allow a characterization of performance for similar models.
- Execution of a complex DEVS model, for example the automated manufacturing system (AMS) described in [Gli04], using a distributed environment. We could define different partitions and determine their impact on the performance of the simulations.

Chapter 7: CONCLUSIONS

This dissertation introduces a new flat simulation technique for Parallel DEVS and Cell-DEVS based on Time Warp, a well-known optimistic synchronization protocol. Our efforts address the need for efficient, fast execution of models using parallel and distributed simulation.

We propose an optimistic distributed mechanism that enables achieving higher degrees of parallelism than previous efforts, which only allowed exploiting parallelism that was inherent to the DEVS formalism. In the previous parallel version of the CD++ tool, synchronization tasks were in charge of a unique DEVS processor, *root coordinator*. Analogous approaches are implemented by other conservative simulators. In general, the centralization of all scheduling tasks creates a bottleneck in *root coordinator* and limits the degree of parallelism. Under our proposed approach, scheduling tasks are distributed on the logical processes; each *node coordinator* is in charge of the scheduling tasks for the local simulation objects. *Node coordinators* advance the simulation optimistically, assuming that there will be no straggler events. In case of detecting a violation to the local causality constraint, a rollback mechanism allows recovering from it.

The simulation approach we propose is carried out by four DEVS processors, namely *simulator*, *flat coordinator*, *node coordinator*, and *root coordinator*. Our design takes into consideration previous studies showing the impact of communication among processors on the overall performance of the simulator. We propose a flat simulation

mechanism, as opposed to the hierarchical mechanism implemented by other versions of CD++ and other tools. Our design eliminates the need for intermediate coordinators by transforming the hierarchical structure of the model into a more simple, non-hierarchical one. As a result, fewer messages have to be exchanged and, therefore, the communication overheads are reduced.

Evaluating the performance of simulation engines can be a very complex task. Instead of limiting our effort solely to testing individual models, we developed DEVStone, a synthetic benchmark to aid not only this but also future initiatives in the area, as ongoing developments intended to improve DEVS simulators also require a way to assess their performance. DEVStone is a synthetic model generator that automatically creates models which resemble real-world applications.

We used DEVStone to study the performance of our new CD++ simulator, and to compare its overhead with other engines supported by the tool. The use of DEVStone provided a common metric that made the comparisons straightforward. Moreover, less time had to be spent in developing models, and a larger batch of such models could be executed with less effort. Thus, it is easier to study the performance of the tool for many models with different characteristics.

DEVStone can be used in any simulator with capabilities for defining and executing Dhrystone code. We can use single-layered models for comparison with tools with non-hierarchical structures. Likewise, if the chosen modeling technique does not support the execution of internal transitions, we can compare the simulators by building a DEVStone in which the execution time for internal transitions is zero.

Using DEVStone, we compared the overhead of our new technique with the overhead of previous implementations. Although the overhead associated with synchronization tasks implemented by our simulator can be considerable, it still outperformed previous alternatives for some models in single-processor executions. This is a consequence of the flat mechanism implemented in our engine that outweighs, in some cases, the increased overhead associated with its more complex implementation. More importantly, we showed that when executing different types of DEVS models, the overhead is reasonable small (2.5%-5%).

We showed that the execution times for a particular Cell-DEVS model can be reduced using distributed simulation. Different model sizes where considered, ranging from 256 to 2500 cells. The execution of the model in a distributed environment allowed achieving better performance than stand-alone execution. Using distributed environments, our simulator outperforms other alternatives and achieves considerable speedups.

7.1 FUTURE WORK

There are several topics of interest for future research, which include:

- Working to solve the problems associated with the rollback mechanism associated with the Warped middleware. As we discussed in Chapter 6, we have experienced problems when receiving straggler messages in a simulator object during the execution of DEVS and Cell-DEVS models. We have

identified the code that forces the simulation to stop and, although this is out of the scope of our work, we are working to solve this problem.

- In terms of performance, the impact of using different parameters for the Time Warp protocol has yet to be determined. For example, the use of a dynamic cancellation strategy could lead to better performance, as suggested in [Rad96]. Dynamic cancellation techniques allow LPs to decide at runtime which cancellation strategy (i.e., lazy cancellation or aggressive cancellation) should be used, based on execution statistics.
- Extensions to the types of models created by DEVStone. Our first goal with DEVStone was to analyze and compare the performance of simulators running on a single processor. Our synthetic benchmark can be extended to generate models that could be easily partitioned across multiple processors. A simple approach is to generate a model with *n* coupled models in its top level. This would allow a straightforward partition into *n* machines. The internal structure of the coupled components could be based on the types of models already developed in this work.
- Analysis of different partition strategies. The choice of how to partition a model has an effect on its performance in parallel environments, as suggested in [Tro01b]. In many cases, determining the best partition for DEVS and Cell-DEVS models is not obvious. Having a characterization of several models and the best alternatives to partition each of them can help users decide how to distribute the execution of new models.

- Support for a dynamic partitioning mechanism. In relation to the previous point, dynamic partitioning mechanisms could be implemented in CD++. Dynamic partitioning allows modifying at runtime the partition used for the model. More specifically, this strategy could allow migrating a *simulator* running on a logical process where the load is heavy, to a different logical process where the load is lighter.
- Further analysis of simulator performance using a faster computer network. The performance analysis presented in this work was carried out on a relatively slow network, where the communication between machines is limited to a total of 10 Mbits per second. It would be interesting to execute the experiments in a faster network (e.g., using a 100-Mbit switch) for further analysis of the communication overheads, and comparison with current results.

REFERENCES

[Ame01] Ameghino, J.; Troccoli, A.; Wainer, G. “Models of complex physical systems using Cell-DEVS.” Proceedings of the 34th Annual Simulation Symposium. Seattle, WA. USA. 2001.

[Ame03] Ameghino, J.; Wainer, G.; Glinsky, E. “Applying Cell-DEVS in Models of Complex Systems.” Proceedings of the Summer Computer Simulation Conference. Montreal, QC. Canada. 2003.

[Bry77] Bryant, R.E. Simulation of Packet Communication Architecture Computer Systems. Massachusetts Institute of Technology, Cambridge, MA. USA. 1977.

[Cha79] Chandy, K.; Misra, J. “Distributed Simulation: A Case Study in Design and Verification of Distributed- Programs.” IEEE Transactions on Software Engineering, pp. 440-452. 1979.

[Cha81] Chandy, K.; Misra, J. Asynchronous distributed simulation via a sequence of distributed systems. *ACM Transactions on Computer Systems.* 3(1), pp. 63-75. 1981.

[Cha89] Chandy, K.; Sherman, R. “The Conditional Event Approach to Distributed Simulation” Proceedings of the Distributed Simulation Conference. Miami, FL. USA. 1989.

[Che04] Saehoon Cheon, Chungman Seo, Sunwoo Park, Bernard P. Zeigler, “Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System.” Advanced Simulation Technologies Conference – Design, Analysis, and Simulation of Distributed Systems Symposium. Arlington, USA. 2004.

[Cho94a] Chow, A.C.; Zeigler, B.P. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism.” Proceedings of the Winter Computer Simulation Conference. Orlando, FL. USA. 1994.

[Cho94b] Chow, A.C.; Kim, D.C.; Zeigler, B.P. “Abstract Simulator for the parallel DEVS formalism.” AI, Simulation, and Planning in High Autonomy Systems. Gainesville, FL. USA. 1994.

[Cho02] Chou, H-H.; Huang, W.; Reggia, J. “The Trend cellular automata programming environment for artificial life, parallel computing, and simulation research” *Transactions of the Society for Modeling and Simulation International.* vol. 78(2), pp. 59-75. 2002.

[Chr90] Christensen, E.R. *Hierarchical optimistic distributed simulation: combining DEVS and Time Warp.* PhD Thesis, University of Arizona. 1990.

[Dav00a] Davidson, A.; Wainer, G. “Specifying truck movement in traffic models using Cell-DEVS.” Proceedings of the 33rd IEEE/SCS Annual Simulation Symposium. Washington DC, USA. 2000.

[Dav00b] Dávila, J.; Uzcágegui, M. “GALATEA: A multi-agent, simulation platform.” Proceedings of the International Conference on Modeling, Simulation and Neural Networks. Mérida, Venezuela. 2000.

[Del02] de Lara, J.; Vangheluwe, H. “ATOM3: A Tool for Multi-Formalism Modeling and Meta-Modeling.” European Joint Conferences on Theory And Practice of Software. Grenoble, France 2002.

[Dia01] Díaz, A.; Vázquez, V.; Wainer, G. “Application of the ATLAS language in models of urban traffic.” Proceedings of the Annual Simulation Symposium. Seattle, WA. USA. 2001.

[Don96] Dongarra, J. et al. *MPI: The Complete Reference*. The MIT Press. 1996.

[Ell04] Elliott, J. M. G. “Cellsprings.” Available via: <http://jmge.net/java/csprings/>. [Accessed August, 2004.]

[Fil02a] Filippi, J-B.; Bernardi, F.; Delhom, M. “The JDEVS environmental modeling and simulation environment” Proceedings of the the IEMSS’02 Conference on Integrated Assessment and Decision Support. Lugano, Switzerland. 2002.

[Fil02b] Filippi, J-B.; Chiari, F.; Bisgambiglia, P. “Using JDEVS for the modeling and simulation of natural complex systems.” Proceedings of the AI, Simulation and Planning Conference. Lisbon, Portugal. 2002.

[Fre01] Freiwald, U.; Weimar, J.R. “JCASim - a Java system for simulating cellular automata” Theoretical and Practical Issues on Cellular Automata (ACRI 2000), S. Bandini and T.Worsch (eds), Springer Verlag, London. 2001.

[Fuj90] Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. Communications of the ACM, 33(10):31-53.

[Fuj99] Fujimoto, R.M. *Parallel and Distribution Simulation Systems.* Wiley. 1999.

[Fuj01] Fujimoto, R.M. “Parallel and Distributed Simulation Systems.” Proceedings of the Winter Computer Simulation Conference. Phoenix, AZ. USA. 2001.

[Gar70] Gardner M. The fantastic combinations of John Conway’s new solitaire game “Life.” *Scientific American.* vol. 23(4). pp. 120–123. 1970.

[Gli02a] Glinsky, E.; Wainer, G. “Definition of Real-Time simulation in the CD++ toolkit.” Proceedings of the Summer Computer Simulation Conference. San Diego, CA. USA. 2002.

[Gli02b] Glinsky, E.; Wainer, G. “Performance analysis of DEVS environments.” Proceedings of AI Simulation and Planning. Lisbon, Portugal. 2002.

[Gli02c] Glinsky, E.; Wainer, G. “Performance Analysis of Real-Time DEVS Models.” Proceedings of the Winter Computer Simulation Conference. San Diego, CA. USA. 2002.

[Gli02d] Glinsky, E.; Wainer, G. “Definition of Real Time Simulation in the CD++ toolkit.” Master’s thesis. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 2002.

[Gli04] Glinsky, E.; Wainer, G. “Real-Time CD++: an Environment for Modeling and Simulation of Hybrid Hardware/Software Systems.” Accepted for publication in Proceedings of the Winter Computer Simulation Conference. Washington DC, USA. December, 2004.

[Gro96] Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. “A high-performance, portable implementation of the MPI message-passing interface standard.” *Parallel Computing*. vol. 22, pp. 789-828. 1996.

[Him04] Himmelspach, J.; Uhrmacher, A.M “A Component-Based Simulation Layer for JAMES.” Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS). Kufstein, Austria. 2004.

[HLA00] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) — Framework and Rules. IEEE Std. 1516-2000. September, 2000.

[Jef85] Jefferson, D.R. “Virtual time.” *ACM Transactions on Programming Languages and Systems.* vol. 7(3), pp. 404-425. July, 1985.

[Kim94] Kim, T.G. “DEVSim++: C++ based Simulation with Hierarchical Modular DEVS Models.” User’s Manual CORE Lab, EE Dept, KAIST, Taejon, Korea. 1994.

[Kim96] Kim, K.H.; Seong, Y.R.; Kim, T.G.; Park, K.H. “Distributed Simulation of Hierarchical DEVS Models: Hierarchical Scheduling Locally and Time Warp Globally” *Transactions of the Society for Modeling and Simulation International.* vol. 13(3), pp. 135-154. 1996.

[Kim00a] Kim, K.; Kang W.; Sagong, B.; Seo, H. “Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One.” Proceedings of the 33rd Annual Simulation Symposium. Washington DC, USA. 2000.

[Kim00b] Kim, K.; Kang, W. “A CORBA-Based Distributed Simulation Methodology for Hierarchical DEVS.” IASTED International Conference on Applied Informatics. Innsbruck, Austria. 2000.

[Kim04] Kim, K.; Kang, W. “CORBA-Based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One.” International Conference on Computational Science and Its Applications (ICCSA). Assisi, Italy. 2004.

[Mar96] Martin, D.; McBrayer, T.; Wilsey, P. “WARPED: Time Warp Simulation Kernel for Analysis and Application Development.” Proceedings of the 29th Hawaii International Conference on System Sciences. 1996.

[Mar97] Martin, D.; McBrayer, T.; Radhakrishan, R.; Wilsey, P. “Time Warp Parallel Discrete Event Simulator.” *Technical report*. Computer Architecture Design Laboratory. University of Cincinnati. USA. 1997.

[Mor02] Moreno, N.; Ablan, M.; Tonella, G. “SPASIM: A Software to Simulate Cellular Automata.” Procedings of the Conference on Integrated Assessment and Decision Support (IEMSS). Lugano, Switzerland. 2002.

[MPI95] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report. Available via: <http://www.mpi-forum.org>. [Accessed August, 2004].

[Nic97] David M. Nicol and Jason Liu “The dark side of risk.” Proceedings of the Workshop on Parallel and Distributed Simulation (PADS). Lockenhaus, Austria. 1997.

[Nut04] Nutaro, J. ADEVS website. Available via <http://www.ece.arizona.edu/~nutaro/>. [Accessed May, 2004]

[OMG02] Object Management Group. The common object request broker: architecture and specification. Revision 3.0. OMG Technical report 2002-06-01, 492 Old Connecticut Path, Framingham, MA. USA.

[Pra99] Praehofer, H.; Sametinger, J.; Stritzinger, A. “Discrete Event Simulation using the JavaBeans Component Model.” Proceedings of International Conference On Web-Based Modeling & Simulation. San Francisco, CA. USA. 1999.

[Rad96] Radhakrishnan, R.; McBrayer, T.J.; Subramani, K.; Chetlur, M.; Balakrishnan, V.; Wilsey, P.A. “A Comparative Analysis of Various Time Warp Algorithms Implemented in the WARPED Simulation Kernel.” Proceedings of the Annual Simulation Symposium, pp. 107-116. New Orleans, LA. USA. 1996.

[Raj98] Rajasekaran, U. K. V. Improving the communication subsystem performance of warped. Master's thesis, University of Cincinnati, November 1998.

[Rao98] Rao, D.M.; Thondugulam, N.V.; Radhakrishnan, R.; Wilsey, P. “Unsynchronized parallel discrete event simulation.” Proceedings of the Winter Computer Simulation Conference. Washington DC, USA. 1998.

[Ron96] Rönngren, R.; Liljenstam, M.; Montagnat, J.; Ayani, R. “Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation.” Proceedings of the 10th ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation. Philadelphia, PA. USA. May, 1996.

[Rod99] Rodriguez, D.; Wainer, G. “New extensions to the CD++ tool.” Proceedings of the Summer Computer Simulation Conference. Chicago, IL. USA. 1999.

[Sar98] Sarjoughian, H.S.; Zeigler, B.P. "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment." Proceedings of the International Conference on Web-Based Modeling and Simulation. vol. 5, pp. 29-36. San Diego, CA. USA. 1998.

[Seo04] Seo, C.; Park, S.; Kim, B.; Cheon, S.; Zeigler, B. "Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment." Advanced Simulation Technologies conference (ASTC). Arlington, VA. USA. 2004.

[Tro01a] Troccoli, A.; Wainer, G. "CD++, a tool for simulating Parallel DEVS and Parallel Cell DEVS models." Technical report. Departamento de Computación, Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 2001.

[Tro01b] Troccoli, A.; Wainer, G. "Performance results of parallel Cell-DEVS execution." Proceedings of the Summer Computer Simulation Conference. Orlando, FL. USA. 2001.

[Tro03] Troccoli, A.; Wainer, G. "Implementing Parallel Cell-DEVS." Proceedings of the Annual Simulation Symposium. Washington DC, USA. 2003.

[Wai98] Wainer, G.; Giambiasi, N. "Specification, modeling and simulation of timed Cell-DEVS spaces." Technical Report n.: 98-007. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 1998.

[Wai00] Wainer, G.; "Improved cellular models with parallel Cell-DEVS." *Transactions of the SCS*. vol 17 (2). June 2000.

[Wai01] Wainer, G.; S. Daicz, S.; De Simoni, L.; Wasserman, D. "Using the ALFA-1 simulated processor for educational purposes." *ACM Journal on Educational Resources in Computing*. vol. 1(4), pp. 111-151. December 2001.

[Wai02] Wainer, G. "CD++: a toolkit to develop DEVS models." *Software - Practice and Experience*. vol. 32, pp. 1261-1306. 2002.

[Wai03] Wainer, G., Chen, W. "A Framework for Remote Execution and Visualization of Cell-DEVS Models." *Transactions of the Society for Modeling and Simulation International*. vol. 79 (11), pp. 626-647. November, 2003.

[Wai04] Wainer, G., Glinsky, E. "Model-Based Development of Embedded Systems with RT-CD++." IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Work-in-Progress session. Toronto, ON. Canada. 2004.

[War04] Warped: A Time Warp Simulation Kernel. *Warped Documentation for version 1.0*. Available via <www.ececs.uc.edu/~paw/warped/>. [Accessed September, 2004.]

[Wei84] Weicker, R. P. "Dhrystone: A synthetic systems programming benchmark." Communications of the ACM, volume 27, pp. 1013-1030, 1984.

[Wes96] Darrin West, Kiran Panesar, Kiran Panesar. “Automatic Incremental State Saving.” Proceedings of the 10th ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation. Philadelphia, PA. USA. May, 1996.

[Wol86] Wolfram, S. *Theory and applications of cellular automata.* Advances Series on Complex Systems. World Scientific. Singapore. 1986.

[Woj04] Wojtowicz, J. “1D and 2D Cellular Automata explorer.” Available via: <http://psoup.math.wisc.edu/mcell/>. [Accessed September, 2004.]

[Zei76] Zeigler, B. *Theory of Modeling and Simulation.* Wiley. 1976.

[Zei93] Zeigler, B.P.; Kim, J. “Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control.” *IEEE Transactions on Robotics and Automation.* vol. 9 (3), pp. 351-356. 1993.

[Zei96] Zeigler, B.; Moon, Y.; Kim, D. “DEVS-C++: A High Performance Modeling and Simulation Environment.” 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture. Hawaii, USA. 1996.

[Zei97a] Zeigler, B.; Moon, Y.; Kim, D.; Ball, G. “The DEVS Environment for High-Performance Modeling and Simulation” *IEEE Computational Science and Engineering.* vol. 4 (3), pp. 61 -71. 1997.

[Zei97b] Zeigler, B.P.; Kim, D.; Praehofer, H. “DEVS Formalism as a Framework for Advanced Distributed Simulation.” Proceedings of the International Workshop on Distributed Interactive Simulation and Real-Time Applications. Eilat, Israel. 1997.

[Zei99a] Zeigler, B.P.; H.S. Sarjoughian, “Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA.” Simulator Interoperability Workshop. 1999.

[Zei99b] Zeigler, B.P. “A Theory-based Conceptual Terminology for M&S VV&A.” Proceedings of the Simulation Interoperability Workshop. Orlando, FL. USA. 1999.

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