NEW ALGORITHMS FOR THE PARALLEL CD++ SIMULATION ENVIRONMENT

By

Shafagh Jafer, B. Eng.

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for 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

© Copyright 2007, Shafagh Jafer

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

Acceptance of the thesis

New Algorithms for the Parallel CD++ Simulation Environment

Submitted by Shafagh Jafer

In partial fulfillment of the requirements for the degree of

Master of Applied Science

_____

Thesis Supervisor

Dr. Gabriel A. Wainer

_____

Chair, Department of Systems and Computer Engineering

Dr. Victor C. Aitken

Carleton University

2007

ABSTRACT

The **DEVS** (Discrete Event System Specification) formalism provides a discrete-event modeling and simulation (M&S) approach which allows construction of hierarchical models in a modular manner. Cell-DEVS extends the DEVS formalism, allowing the implementation of cellular models with timing delays. This work presents a new simulation technique of DEVS and Cell-DEVS models in parallel and distributed environments. The parallel simulators presented in here are based on Time Warp, an optimistic synchronization protocol, which are developed as new simulation engines for CD++, a M&S toolkit that implements DEVS and Cell-DEVS formalism. Two distinct parallel simulators, namely Purely Optimistic PCD++ and Conservative PCD++ are introduced which use hierarchical and flattened architecture respectively. Different Cell-DEVS models are built in CD++ in order to judge the performance of these two simulators. Moreover, two new algorithms, Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GRFM) are implemented to control optimism of the optimistic PCD++. The LRFM and GRFM techniques are modifications to the WARPED kernel which are applied to the optimistic PCD++. A set of detailed tests are collected to investigate the effect of these approaches on the simulator.

This work is dedicated to Mohammad
for all moments of love and inspiration.

ACKNOWLEDGEMENTS

I would like to gratefully thank my mother, father, Gufran, Yasser, Wajd, and Zahra who have motivated, encouraged, and supported me all the way.

Also, I want to thank my wonderful supervisor, Dr. Gabriel Wainer for his endless kindness, patience, and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

M&S	Modeling and Simulation
DEDS	Discrete Event Dynamic Systems
DEVS	Discrete Event System Specification
P-DEVS	Parallel Discrete Event System Specification
PDES	Parallel Discrete Event Simulation
MPI	Message Passing Interface
GVT	Global Virtual Time
LGVT	Local GVT estimate
LVT	Local Virtual Time
LP	Logical Process
LTSF	Least-Time-Stamp-First scheduling
NC	Node Coordinator
FC	Flat Coordinator
UCSS	User Controlled State Saving
NPSI	Near Perfect State Information
LRFM	Local Rollback Frequency Model
GRFM	Global Rollback Frequency Model

CHAPTER 1 INTRODUCTION

Modeling and simulation (M&S) methodologies have become crucial for implementing, designing, and analyzing a broad variety of systems. The simulation process begins with a problem to solve. First, the real system is observed, its entities are identified, and a **model** is constructed. Then, the model is executed using a **simulator** consisting of a computer system which executes the model's instructions and generates relevant output. These outputs are compared with the real system to verify the correctness of the model. In [Zei00] a general framework has been implemented which describes the basic entities in M&S and their relationships.

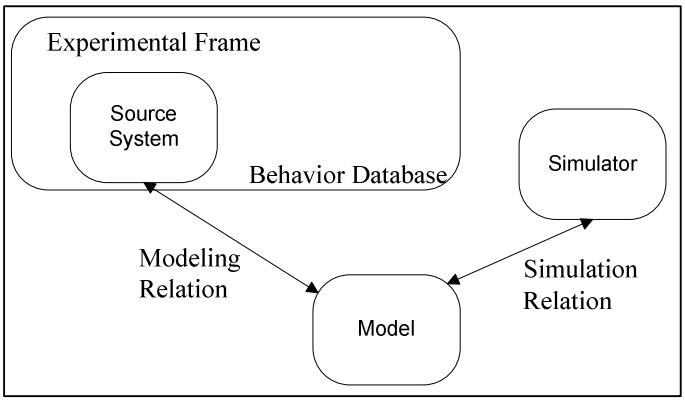


Figure 1. The basic entities and their relationships [Zei00]

This M&S framework consists of three basic entities which are linked by two relations:

- *Source system entity*: this entity is the real or virtual environment under analysis. This entity which is viewed as the data source, together with the *behavior database* form the **Experimental Frame.**
- *Model entity*: a model entity represents an abstraction of the source system represented by a set of instructions, rules, mathematical equations, or a set of constraints to approximate the behavior of the real system.
- *Simulator entity*: the simulator is a computer-based entity which is in charge of executing the model's instructions.
- *Modeling relation*: this relation links the **model** and the **source system** to validate the results generated by the model. In general, the model is considered valid if the

data it generates agree with the data generated by the source system in the experimental frame in use.

- *Simulation relation*: this relation lies between the **simulator** and the **model** to indicate how reliable is the simulator in terms of being capable to execute the model's instructions.

The separation between model and simulator significantly simplifies the model validation and simulator verification [Zei00]. Furthermore, this advantage gives the opportunity to use different simulation algorithms within the simulator or even using different simulators. Also, the separation of concerns involved in this architecture allows model reusability as well as later extension of the model.

Among the existing modeling and simulation techniques, **DEVS** (Discrete Event System Specification) formalism [Zei76] provides a discrete-event approach which allows construction of hierarchical models in a modular manner. DEVS is a sound formal framework based on generic dynamic systems concepts that allows model reuse, and reduction in development and testing time due to its hierarchical approach in constructing models. In this work, our main focus is on discrete-event M&S approach and DEVS formalism [Zei76, Zei00] which has been proven to be a universal formalism to represent **DEDS** (Discrete Event Dynamic Systems) [Cas93].

DEVS formalism has been extended to handle simultaneous event execution. **Parallel DEVS** or **P-DEVS** [Cho94], allows more efficient execution of models in parallel and distributed environments by keeping the major properties of the original DEVS formalism and just extending it to overcome the serialization constraints.

The **Timed Cell-DEVS** formalism [Wai98] is an extension to the traditional Cellular Automata [Wol86] which makes use of DEVS by defining every cell to represent an atomic DEVS model and coupling them together to form a complete cell space representing a coupled DEVS model. This formalism allows defining complex cell behavior with simple instructions. It also allows construction of n-dimensional cell spaces to represent more complicated discrete event models. Using this formalism, complex timing behavior can be represented by defining different timing delays among the cells of the cell space. The main advantage that Timed Cell-DEVS has over Cellular Automata is that using the *state* of each cell (since each cell is a basic DEVS model), only *activated*

cells are evaluated as opposed to the original Cellular Automata where all cells get evaluated at each time step which results in a noticeable waste of time.

DEVS and Cell-DEVS formalisms have been successfully used to develop complex models in different fields of science including: physics, biology, chemistry, ecology, as well as computer networks, traffic modeling, and many other systems. Example of such models would be: fire spread in forests [Ame01], land battlefield of two armies [Mad05], computer networks [Ahm05], and ATLAS [Dia01].

Parallel and distributed simulation (PADS) techniques were proposed to resolve the issues of complex models simulation. As the models become larger and more complex, the problems of limited resources within a single-processor arise. Not only the shortage of resources, but also the long execution times brought up the idea of **Parallel discrete event simulation** (PDES) studies. Fujimoto [Fuj01] classifies three major research categories in the area of parallel and distributed simulation. The first research group is the high performance computing community which started in late 1970's and 1980's aiming at reducing execution time by using multiple processors. This community developed the world wide known fundamental ideas by proposing two synchronization algorithms: **Chandy-Misra-Bryant** [Bry77, Cha79] and **Time Warp** [Jef85]. The second group is the Defense community, which mainly focuses on facilitating interoperability and software reuse. Finally, the third group is the gaming and Internet community which is interested in developing realistic scenarios in distributed environments.

Parallel Cell-DEVS [Wai00] formalism extends the standard formalisms of Cell-DEVS to allow a higher degree of parallelism in parallel and distributed environments. This formalism overcomes the restrictions of serialized simulation by revising and extending Cell-DEVS to allow a higher degree of parallelism and allowing zero-delay transitions as well as multiple simultaneous events per external ports.

CD++ [Wai01a, Wai02] is a modeling toolkit that implements the DEVS and Cell-DEVS theories by applying the original formalisms. The toolkit includes facilities to build DEVS and Cell-DEVS models. This tool has been revised and extended several times, and currently supports standalone [Rod99], real-time [Gli02a], parallel

conservative [Tro03], parallel purely optimistic [Liu06], and web service-based [Mad06] simulation.

Synchronization as the key to parallel and distributed simulation requires a robust mechanism to handle communication among concurrent processes. In general, a parallel or distributed simulation runs on multiple parallel or distributed processors interconnected by a communication network. There exist two major classes of synchronization: *conservative* (or *pessimistic*) approaches and *optimistic* approaches. *Optimistic* approaches have a higher degree of parallelism unlike the conservative approaches where they are overly pessimistic and force the simulation to behave sequentially when it is not necessary. Conservative approaches rely very much on application-specific information when making run-time decisions on whether it is safe to process the event or not. On the other hand, the optimistic mechanisms are less reliant on the application for correct execution, therefore allowing a simplified software development and more transparent synchronization.

The focus of this work is on improving the capability of CD++ in supporting P-DEVS and Parallel Cell-DEVS modeling and simulation. This work is based on previous research: PCD++ which is an optimistic DEVS and Cell-DEVS parallel simulator [Liu06], and the conservative PCD++ simulator for DEVS and Cell-DEVS [Tro01]. Our work aims at: 1) modifying the existing optimistic simulator to enhance the performance of large scale models executions, 2) analyzing the performance of these two simulators using precise testing scenarios.

1.1. CONTRIBUTION

We present new implementations in the Time Warp protocol to improve the CD++-based parallel and distributed simulations by controlling the optimism of our optimistic PCD++ simulator. We have implemented two new protocols, namely *Local Rollback Frequency Model* (LRFM) and *Global Rollback Frequency Model* (GRFM) to limit the optimism [Szu00]. This was done by modifying the WARPED [Mar99] kernel to implement a Near Perfect State Information (NPSI) mechanism based on the number of rollbacks. The idea is to reduce the number of rollbacks by suspending the simulation object within logical

process that has large number of rollbacks, hence blocking it from flooding the net with anti-messages. However, this new design allows the logical process to stay receiving input events and inserting them into the corresponding message queues. After a predefined duration, the suspended simulation object is released and will resume its simulation duties. The LRFM protocol is only based on local information of the logical processes. Thus, the simulation object will be suspended or allowed to continue simulation only based on its number of rollbacks. In contrast, in the GRFM protocol, each simulation object uses global information in such a way that among all the simulation objects residing on all logical processes, the one with greatest number of rollbacks must be suspended for a predefined duration.

We also present a set of complex models implemented in Cell-DEVS using our CD++, including: Game of Life, Synapsin-Vesicle Reaction at Nerve Terminal, Fire Spread, and Ship Evacuation model. These models were selected based on their distinguishable functionality, complexity, and size to better highlight the capability of our simulator.

Finally, we have run a set of detailed test cases using a variety of models to observe the performance of both the Conservative PCD++ Simulator [Tro01] and the Purely Optimistic PCD++ simulator [Liu06]. Precise testing strategies were used to analyze the performance of our existing PCD++ simulators; the optimistic and the conservative as well as our LRFM- and GRFM-based protocols. The main goal of this research work is to create a workbench consisting of four different simulators; Conservative, Pure Optimistic, LRFM-based Optimistic, and GRFM-based Optimistic simulators. This workbench serves as simulation environment that can be used as the base in studying parallel simulations of DEVS and Cell-DEVS. On the other hand, the precise and detailed testing scenarios that we are presenting can be used along with this workbench to analyze the capability, performance, and robustness of PCD++ simulators. This work was the first attempt to use optimism controlling simulators for simulating parallel DEVS and Cell-DEVS model.

1.2. THESIS ORGANIZATION

This thesis is organized as follows:

Chapter 2 gives an overview about the state-of-the-art in the field of discrete event modeling and simulation by presenting DEVS and Cell-DEVS formalisms and their extensions. Then, the two major synchronization mechanisms namely optimistic approaches and conservative approaches for parallel and distributed simulation will be discussed. Finally, a survey of the existing DEVS-based simulation toolkits is provided.

Chapter 3 covers the software architecture of the purely optimistic parallel CD++ simulator (PCD++). The layered architecture of the software will be presented followed by a more detailed discussion of each layer.

Chapter 4 introduces the two parallel CD++ simulators by presenting the design and implementation of each of them. Also, the two simulators are compared in terms of their structure as well as functionalities in parallel and distributed simulations.

Chapter 5 illustrates different models implemented in Cell-DEVS on our CD++ toolkit.

Chapter 6 presents two new algorithms that we have implemented in WARPED kernel. First the rollback mechanism of the optimistic PCD++ simulator is discussed. Then, the *Near-perfect State Information* protocol is presented. Finally, our new algorithms; Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GFRM) are illustrated.

Chapter 7 covers the experimental results for measuring the performance of four different PCD++ simulators.

Chapter 8 presents the main conclusion of the thesis as well as future research work that can extend the outcome of this work.

CHAPTER 2 REVIEW OF THE STATE OF THE ART

This chapter gives an overview about the state-of-the-art in the field of discrete event modeling and simulation. Section 2.1 and 2.2 will present background information about DEVS and Cell-DEVS formalisms and their extensions. Then, the two major synchronization mechanisms namely optimistic approaches and conservative approaches for parallel and distributed simulation will be discussed in Section 2.3. Finally, Section 2.4 will cover a survey of the existing DEVS-based simulation toolkits.

2.1. DEVS AND PARALLEL DEVS FORMALISMS

DEVS [Zei76, Zei00] is a formalism for modeling and simulation of DEDS (Discrete Events Dynamic Systems) which provides a framework for the definition of hierarchical models in a modular way by decomposing the real system into behavioral (atomic) and structural (coupled) components. DEVS theory provides a rigorous methodology for representing models, and it does present an abstract way of thinking about the world with independence of the simulation mechanisms and the underlying hardware and middleware. A DEVS atomic model is formally defined by:

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

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;

S is the set of sequential states;

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

$\delta_{ext}: Q \times X \rightarrow S$ is the external state transition function, where $Q = \{(s,e) \mid s \in S, 0 < \in < ta(s)\}$ is the total state set, e is the time elapsed since the last state transition;

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

$ta: S \rightarrow R^{+}_{0,\infty}$ is the time advance function.

The semantics for this definition is given as follows. At any time, a DEVS coupled model is in a state $s \in S$. In the absence of external events, the model will stay in this state for the duration specified by ta(*s*). When the elapsed time *e*, is equal to ta(*s*), the state duration expires and the atomic model will send the output $\lambda(s)$ and performs an internal transition to a new state specified by $\delta_{int}(s)$. Transitions that occur due to the expiration of ta(*s*) are called internal transitions. However, state transition can also happen due to arrival of an external event which will place the model into a new state specified by $\delta_{ext}(s,e,x)$; where *s* is the current state, *e* is the elapsed time, and *x* is the input value. The time advance function ta(s) can take any real value from 0 to ∞. A state with ta(s) value of zero is called *transient* state, and on the other hand, if ta(s) is equal to ∞ the state is said to be *passive*, in which the system will remain in this state until receiving an external event. Figure 2 shows the description of states and variables in DEVS models.

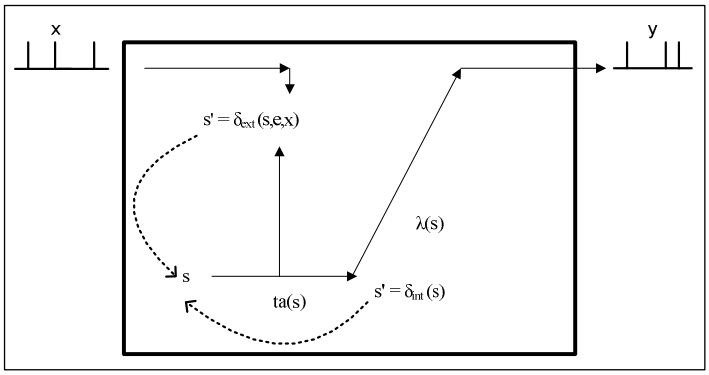


Figure 2. DEVS semantics

A DEVS *coupled model* is composed of several atomic or coupled submodels, which 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 requirements are imposed on the components, which must also be DEVS models:

For each $d \in D$

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

$X_d = \{(p,v) \mid p \in IPorts_d, v \in X_p\}$, and

$Y_d = \{(p,v) \mid p \in OPorts_d, v \in Y_p\}$.

The component couplings are subject to the following requirements:

External input coupling (EIC) connects external inputs to component inputs,

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

External output coupling (EOC) connects component outputs to external outputs,

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

Internal coupling (IC) connects component outputs to component inputs,

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

Select: $2^D - \{\} \rightarrow D$ is the tie-breaking function for imminent components.

Direct feedback loops are not allowed, i.e., an output port of a component may not be connected to an input port of the same component. Formally:

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

Also, the values sent from a source port must follow the range inclusion constraint of a destination port, formally expressed as:

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

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

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

From the coupled DEVS formalism it can be observed that due to the *closure* property, a coupled model is regarded as a new DEVS model [Zei00]. This property ensures that the overall behavior of a coupled model is equivalent to a basic atomic model, and therefore allows hierarchical model construction. The X and Y sets describe the input and output events of the coupled model. Upon reception of an input event, it has to be redirected to the corresponding atomic component. Similarly, when an output is generated by a component, it must be mapped as an input to another component or sent out as an output of the coupled model. The mapping mechanism is defined by the *Z* function.

In coupled DEVS models, when multiple imminent components are scheduled for an internal transition at the same time, this can lead to ambiguity. For example, let's consider a case where we have two imminent components: A, and B. When component A executes its internal transition, it produces an output that maps to an external event for component B. However, at this moment, component B is already scheduled for an internal transition. This will cause an ambiguity for component B, not knowing which transition to execute first. The coupled DEVS formalism suggests two alternatives for this scenario: 1) execute the external transition first with *e* being equal to *ta*(*s*) and then the internal transition, or 2) execute the internal transition first and then the external transition with *e* being equal to zero. DEVS resolves this ambiguity by introducing the *select* tie-breaking function. This function gives order to the imminent components of a coupled model so that only one component has $e = 0$. Then the rest of imminent components are divided into two groups: 1) a set of components that receive an external output from this model, 2) the rest of components. The first group will then execute their external transition functions with $e = ta(s)$, and the second group will be imminent during the next simulation cycle which may further require the use of *select* function to decide which component is going to be the first. The use of tie-breaking mechanism adds overhead to the simulation and, in addition, decreases the level of parallelism and forces the simulation to have a serialized manner. Since the *select* mechanism associates priorities with imminent components, it will cause a potential bottleneck in the simulation system when many interconnected atomic models are imminent at the same time.

Parallel DEVS or **P-DEVS** [Cho94a] is an extension to DEVS that eliminates all the serialization constraints and provides an environment for executing simultaneous DEVS models in parallel. P-DEVS implements *confluent* function to deal with collision scenarios at which events get scheduled simultaneously [Zei00]. Collision handling: the modeler has the responsibility of controlling collision's behavior.

An atomic P-DEVS model is specified by:

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

where

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

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

S	is the set of sequential states;
$\delta_{ext}: Q \times X_M^b \rightarrow S$	is the external state transition 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.

From the following, differences between DEVS and P-DEVS can be noted:

- Instead of having a single input, a bag of inputs is implied to enable concurrent execution of events.
- To define the model's state at the time of collision (i.e. simultaneous internal and external transitions), the confluent function δ_{con} has been defined. The modeler takes care of this function and specifies the behavior of the model when collision occurs.

The elimination of the sequential *Select* function and its replacement with the *confluent transition function* gives all the imminent components equal priority and the permission to be activated and to send their output to other components at the same time. At the other end, the receiver component is only responsible for identifying the type of the received input event and taking the required actions.

P-DEVS coupled models are similar to DEVS, except for the omission of *Select* function. Formally, a coupled model is defined as:

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

Therefore, the set of input and output events (*X* and *Y*), components (*D* and M_d), and couplings (*EIC*, *EOC*, and *IC*) are identically the same as of DEVS. Since in P-DEVS there is no serialization among imminent components, in case of having multiple imminent components within a coupled P-DEVS model, firstly, all the outputs are collected and redirected to the corresponding influences, secondly, the transition function is executed [Zei00].

2.1.1. DEVS model example: A Bluetooth simulator

In order to show how to define DEVS models, we have built a Bluetooth DEVS model to show how two paired devices communicate with each other. Bluetooth is a wireless connection that enables devices such as mobile phones, computers and PDAs to exchange information. Figure 3 illustrates the components of the model. As shown in the figure, the model consists of four components: SenderMobile, ReceiverMobile, DataTransmission, and CennectionManager. The ConnectionManager is further decomposed into PairedDeviceFinder, and PermissionVerifier.

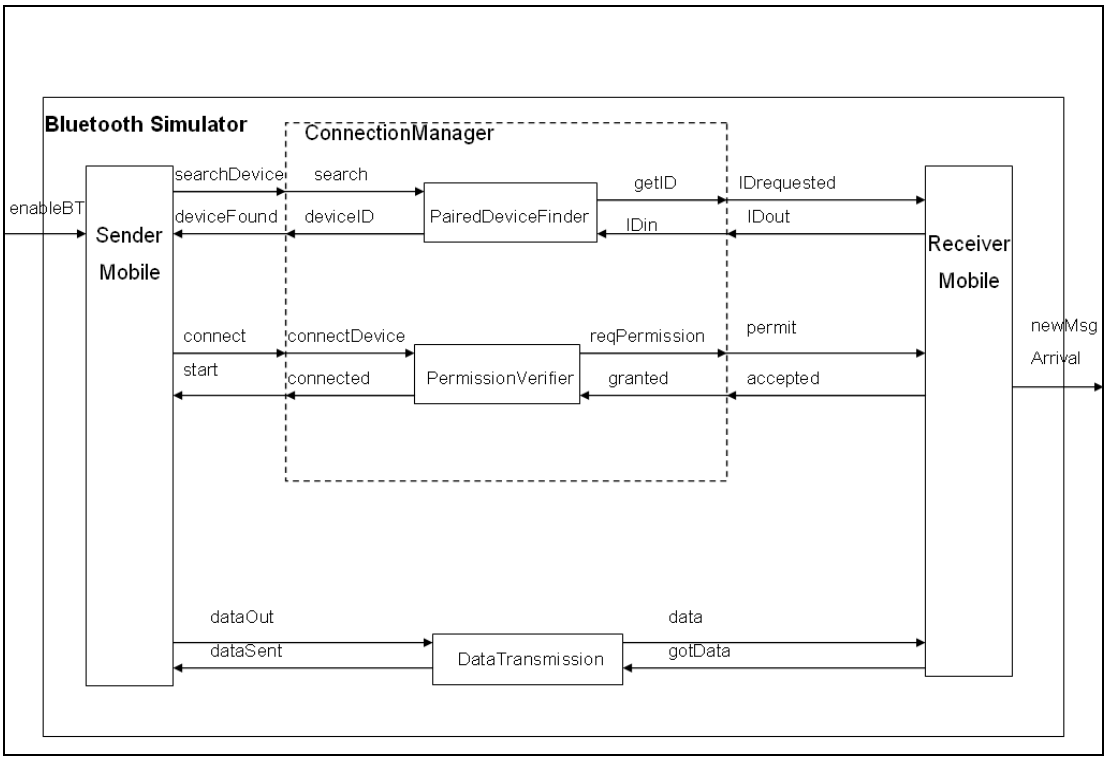


Figure 3. Structure of Bluetooth Simulator

The sender is activated when it receives “enableBluetooth” command which means the user of the mobile device wants to send data to another mobile using Bluetooth. Once the sender mobile gets its Bluetooth enabled, the connection manager is responsible to search for other mobile devices that are in range and have their Bluetooth feature on. As soon as a paired device is found, its ID is sent to the sender mobile (it will be displayed on the mobile screen, for simplicity in this model it is assumed that there is only one paired device available in that range). Once the sender mobile is informed about

the existence of the receiver mobile, it will issue a “connect” command to start the transmission of data. The connection manager is then activated again and will request permission from the receiver mobile to transfer the data. After that, the receiver mobile will grant the permission and the connection manager will notify the sender mobile to start the transmission. The sender mobile sends the data to the dataTransmission handler which takes care of ensuring enough capacity at the receiver as well as reliable transmission. Once the transmission is done successfully the sender is informed and it goes back to passive state by disabling its Bluetooth and waiting for another input command (i.e. “enableBT”). As shown in Figure 3, the Bluetooth Simulator has one input and one output. The *enableBT* input indicates that the user of *SenderMobile* would like to start a Bluetooth transfer of data to a *ReceiverMobile*. Whenever this command is issued the *SenderMobile* is activated and its state changes from *passive* to *active*. The *newMsgArrival* output indicates that there is new message received. At the *ReceiverMobile* there is a counter which counts the number of new messages arrived. The coupled component *ConnectionManager* and the atomic component *DataTransmission* handle connection establishment and data transfer between SenderMobile and ReceiverMobile atomic components.

Formal Specifications for Atomic Models:

The formal specifications <S, X, Y, δ_{int}, δ_{ext}, λ, ta> for the atomic models are defined as follows:

ReceiverMobile:

Assumption: the receiver will always accept the request of receiving data from other mobile device. It is assumed that the sender and receiver mobiles know each other. Also, the receiver’s Bluetooth is always enabled.

S = {phase, ID, totalMsgs, ID_requested, permission_given, gotMsg }

X = {IDrequested, permit, data}

Y = {IDout, accepted, gotData, newMsgArrival}

DataTransmission:

Once connection is established, the SenderMobile will send the data through the DataTransmission to the ReciverMobile. At the receiver when the data is received, the

DataTransmission will signal the sender that data was transmitted successfully. The Sender then goes back to passive state and waits for the next “enableBT” request.

S = {phase, data, dataReceived}

X = {dataOut, gotData}

Y = {dataSent, data}

PairedDeviceFinder:

This unit is responsible for searching paired devices (mobile sets that are Bluetooth enabled and are in range). SenderMobile requests to have the ID of the paired devices. Once a paired device is found its ID is sent to the SenderMobile. In this assignment it is assumed that there is only one receiver mobile available in that area.

S = {phase, receiverID, reqSearch}

X = {search, IDin}

Y = {getID, deviceID}

PermissionVerifier:

After getting the ID of the receiver, the sender will have to ask for permission in order to start the data transmission. PermissionVerifier unit will handle this by coordinating between the sender and receiver.

S = {phase, accessOK, reqConnection }

X = {connectDevice, granted}

Y = {connected, reqPermission}

SenderMobile:

Assumption: the receiver will always accept the request of receiving data from other mobile device. It is assumed that the sender and receiver mobiles know each other. Also, the receiver’s Bluetooth is always enabled.

S = {phase, gotID, receiverID,enable, sending }

X = {enableBT, deviceFound, start, dataSent }

Y = {searchDevice, connect, dataOut }

Formal Specifications for Coupled Models:

The formal specifications <X, Y, D, EIC, EOC, IC, SELECT > for the coupled model ConnectionManager and BluetoothSimulator are defined as follows:

ConnectionManager:

X = {search, IDin, connectDevice, granted};

Y = {deviceID, getID, connected, reqPermission };

D = {PermissionVerifier, PairedDeviceFinder};

EIC = {(ConnectionManager.search, PariedDeviveFinder.search),
(ConnectionManager.IDin, PariedDeviveFinder.IDin),
(ConnectionManager. connectDevice, PermissionVerifier. connectDevice),
(ConnectionManager.granted, PermissionVerifier. granted)}

EOC = {(ConnectionManager. deviceID, PariedDeviveFinder. deviceID),
(ConnectionManager. getID, PariedDeviveFinder. getID),
(ConnectionManager. connected, PermissionVerifier. connected),
(ConnectionManager. reqPermission, PermissionVerifier. reqPermission)}

IC = {ϕ}

SELECT: ({PermissionVerifier, PairedDeviceFinder }) = PairedDeviceFinder;

BluetoothSimulator Simulator: This is the TOP component encapsulating the whole model.

X = {enableBT };

Y = {newMsgArrival };

D = {SenderMobile, ReceiverMobile, ConnectionManager, DataTransmission };

EIC = {(BluetoothSimulator.enableBT, SenderMobile. enableBT),

EOC = {(BluetoothSimulator. newMsgArrival, ReceiverMobile. newMsgArrival),

IC = { (SenderMobile.searchDevice, ConnectionManager.search),
(SenderMobile.connect, ConnectionManager.connectDevice),
(ConnectionManager.deviceID, SenderMobile.deviceFound),
(ConnectionManager.connected, SenderMobile.start) ,(SenderMobile.dataOut,
DataTransmission.dataOut), ,(SenderMobile.dataSent,
DataTransmission.dataSent) ,(DataTransmission.data, ReceiverMobile.data) ,
(DataTransmission.gotData, ReceiverMobile.gotData),
(ReceiverMobile.IDrequested, ConnectionManager. IDrequested) ,
(ReceiverMobile.IDout, ConnectionManager. IDout)
(ReceiverMobile.permit, ConnectionManager. permit)
(ReceiverMobile.accepted, ConnectionManager. accepted)}

SELECT: ({DataTransmission, ConnectionManager }) = ConnectionManager
({SenderMobile, ConnectionManager }) = SenderMobile
({SenderMobile, DataTransmission }) = SenderMobile

Using CD++ toolkit, the described model can be simulated and the results are tested for verification. The *closure under coupling* property of the coupled DEVS formalism allows incremental testing of the model. This is performed by testing each individual atomic model as well as the coupled ones. The TOP component which encapsulates the whole model can be tested as well, and the results are compared with ones obtained from atomic component's test results.

For the top coupled model testing can be done by inputting only one event which is "enableBT". This is the command issued by the user of the SenderMobile requesting to enable its Bluetooth and transferring data to other paired devices. At the end of simulation the out put "newMsgArrival" is expected which shows how many massages were received. For example by setting "enableBT" to 3, the following results should be seen:

```
Input:                              Output:

00:00:00:00   enableBT 1      00:00:28:000 newmsgarrival 1
00:00:30:00   enableBT 1      00:00:58:000 newmsgarrival 2
00:00:60:00   enableBT 1      00:01:28:000 newmsgarrival 3
00:00:120:00  enableBT 1      00:02:28:000 newmsgarrival 4
00:00:180:00  enableBT 1      00:03:28:000 newmsgarrival 5
00:00:2400:00 enableBT 1      00:05:28:000 newmsgarrival 6
00:00:300:00  enableBT 1      00:40:28:000 newmsgarrival 7
```

From the above simulation results we can see that at the receiver end 7 new messages were received. This result and also the corresponding timings verify the correctness of our model.

2.2. TIMED CELL-DEVS AND PARALLEL CELL-DEVS FORMALISMS

The Cellular Automata formalism [Wol86] uses cell spaces to represent real systems. A cellular automaton is an infinite regular n-dimensional lattice, where each cell holds one finite value. The lattice consists of cells having state variables and a computing apparatus, which is in charge of updating cell's state according to a local rule. This is

performed by using the current cell's state and those of a finite set of nearby cells (called the neighborhood of the cell) [Wai00].

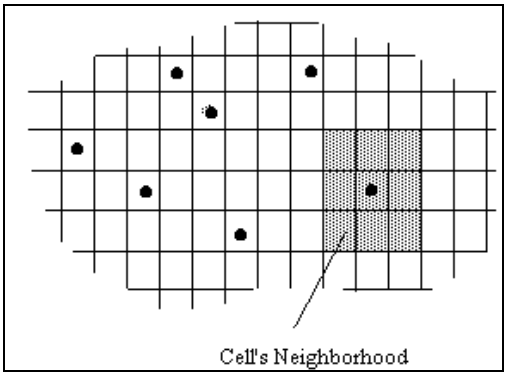


Figure 4. Sketch of a Cellular Automaton [Wai00]

The major limitation of this discrete time paradigm is that at each discrete time step when the values of all the cells get updated, usually there are several cells which do not require this update, therefore noticeable computational time is wasted [Wai01b]. To solve this problem, **Cell-DEVS** [Wai98] was proposed which integrates DEVS and cellular automata by presenting each cell as an atomic DEVS model.

Cell-DEVS extends DEVS formalism, allowing the implementation of cellular models with timing delays. Two types of timing delays can be used, namely *transport* and *inertial* [Gia76]. When transport delay is used, the future value is added to queue sorted by output time, allowing the previous values that were scheduled for output but have not yet been sent to be kept. On the other hand, inertial delays allow a preemptive policy at which any previous scheduled output value will be deleted and the new value will be scheduled. A Cell-DEVS atomic model is defined by [Wai01b]:

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

where

X	is a set of external input events;
Y	is a set of external output events;
I	represents the model's modular interface;
S	is the set of sequential states for the cell;
θ	is the cell state definition;
N	is the set of states for the input events;
d	is the delay for the cell;

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

The modular interface (I) represents the input/output ports of the cell and their connection to the neighbor cell. Communications among cells are performed through these ports. The values inserted through input ports are used to compute the future state of the cell by evaluating the local computation function τ. Once τ is computed, if the result is different than the current cell's state, this new state value must be sent out to all neighboring cells informing the state change. Otherwise, the cell remains in its current state and therefore no output will be propagated to other cells. This will happen when the time given by the delay function expires. Finally, the internal, external transition functions and output functions (λ) define this behavior. Cell-DEVS improves execution performance of cellular models by using a discrete-event approach. It also enhances the cell's timing definition by making it more expressive. Cell-DEVS coupled models represent the cell space as follows:

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

where

X_{list}	is the input coupling list;
Y_{list}	is the output coupling list;
I	represents the definition of the model's interface;
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.

A coupled model is composed of an array of atomic cells (C) with given size and dimensions where each cell is connected through standard DEVS input/output ports to the cells defined in the neighborhood (N). Since the cell space is finite, the borders of the cells are either connected to a different neighborhood than the rest of the space, or they are "wrapped" (i.e. B = {∅}) in which they are connected to those in the opposite one using the inverse neighborhood relationship. However, border cells have a different behavior due to their particular locations, which result in a non-uniform neighborhood. The Z function defines the internal and external coupling of cells in the model. It translates the outputs of the i^{th} output port in cell C_a into values for the i^{th} input port in cell C_b. *Select* function has similar functionality as in basic DEVS models, where it is the tie-breaking function for the imminent components.

As in coupled DEVS models, the use of *Select* function produces serialization, and therefore similar limitations when the Cell-DEVS models are considered to be executed in parallel. These limitations would lead to lack of parallelism exploitation and a probable inconsistency with the real system [Wai99]. Moreover, since the timed Cell-DEVS allows only one input from each input port, zero-delay transitions are not possible and also the external DEVS models are not allowed to send two simultaneous events to the same cell. The **Parallel Cell-DEVS** [Wai00] formalism overcomes these restrictions by revising and extending Cell-DEVS to allow a higher degree of parallelism and allowing zero-delay transitions as well as multiple simultaneous events per external model. Below is a summary of distinguishable characteristics of parallel Cell-DEVS which are presented in [Wai00]:

1. Parallel Cell-DEVS models are equivalent to parallel DEVS models.
2. Closure under coupling holds for parallel Cell-DEVS models as well: that is a coupled Cell-DEVS model is equivalent to an atomic Cell-DEVS model.

2.3. THE CD++ TOOLKIT

CD++ [Wai02] is a modeling tool that implements the DEVS and Cell-DEVS theories by applying the original formalisms. The toolkit includes facilities to build DEVS and Cell-DEVS models. DEVS atomic models can be programmed and incorporated into a class

hierarchy programmed in C++. Furthermore, coupled models can be defined using a built-in specification language. Therefore, coupled and Cell-DEVS models need not to be programmed, rather the tool provides a specification language that defines the model's coupling, the initial values, the external events, and the local transition rules for Cell-DEVS models.

CD++ also includes an interpreter for Cell-DEVS models [Wai98]. The language is based on the formal specifications of Cell-DEVS. The model specification includes the definition of the size and dimension of the cell space, the shape of the neighborhood and the border. The cell's local computing function is defined using a set of rules with the form *POSTCONDITION DELAY { PRECONDITION }*. These indicate that when the *PRECONDITION* is met, the state of the cell will change to the designated *POSTCONDITION* after the duration specified by *DELAY*. If the precondition is not met, then the next rule is evaluated until a rule is satisfied or there are no more rules.

2.4. PARALLEL AND DISTRIBUTED SIMULATION

As we mentioned earlier, P-DEVS and Parallel Cell-DEVS extend the standard formalisms of their type to allow a higher degree of parallelism in parallel and distributed environments. In such environments the entire task of simulation is divided among the processors or nodes. Therefore each one of these concurrent nodes handles a smaller portion of the simulation while the whole process of execution takes place in parallel and as a result in a significantly reduced time.

During a parallel, distributed simulation, a number of Logical Processes (**LP**) will be in charge of carrying on the execution of the model on different CPUs [Fuj00]. Each LP executes a part of the simulation by assigning it to one or more *simulation objects*. Logical processes will communicate to each other by sending time-stamped messages. Synchronization among these LPs is violated when an out of order event is received by one of the LPs. This violation is referred to as *causality error*. Such a scenario is represented by Figure 5 where two LPs each with one event in its input queue process their events simultaneously. LP2 as a result of executing *e1* (time stamp of *e1* = 1), generates and sends a new even *e2* (with time stamp = 2) LP1. But at that time, LP1 has

already processed *e6* and therefore the time of LP2 has been already advanced to 6. As a result, arrival of *e4* at LP2 violates the *local causality constraint* and a causality error arises.

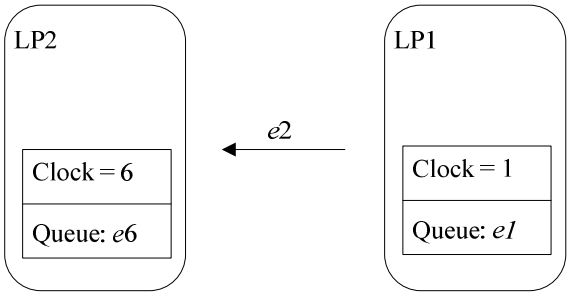


Figure 5. Causality error at LP2

Synchronization among nodes (LPs) is the most challenging problem of parallel and distributed simulation. There exist three different types of synchronization strategies for event driven simulations:

1. No synchronization at all: synchronization is ensured by the application (i.e. sequential simulations).
2. Pessimistic (conservative) synchronization [Bry77, Cha79]: causality violations are strictly avoided.
3. Optimistic synchronization [Jef85]: causality errors are fixed by the notion of *rollbacks*.

2.4.1. Conservative parallel discrete event simulation

Conservative synchronization approaches were the first synchronization algorithms proposed in the late 1970s by R. E. Bryant [Bry77], K. M. Chandy and J. Misra [Cha78]. This synchronization technique which is known the Chandy-Misra-Bryant (CMB) algorithm, disallows any occurrence of causality errors. In conservative schemes, if a LP has an unprocessed event with timestamp *t* and it is guaranteed that no event with earlier timestamp can be received, then the probability that causality error may happen is zero. When the LP has a list of unprocessed events from all other LPs it can safely process the event with lowest timestamp because the future events will for sure have larger timestamps. As long as there are unprocessed events from all other LPs, then this cycle can be repeated and synchronization is guaranteed. However, if this condition is not met, then there is a risk of deadlock. Technique to resolve this deadlock is to find the model's

lookahead, which provides the smallest time stamp of the new events that a process can schedule in the future. Null messages are responsible to carry out the lookahead information among LPs. This way each LP, based on the lookahead information that it receives from all other LPs can derive a lower bound on the time stamp (LBTS) of the events that it will receive in future. As a result, the LP would know which event is safe to process. An example of a safe lookahead value is the timestamp of the first unprocessed event in the input queue. The main drawback of the conservative synchronization approach is the time-wasting flow of null messages which degrade the simulation performance significantly. Optimistic approaches also offer two important advantages over conservative techniques [Fuj03].

2.4.2. Optimistic parallel discrete event simulation

In this technique, which was first proposed by Jefferson's Time Warp mechanism [Jef85], each LP has a *Local Virtual Time* (**LVT**) which advances every discrete step as events are executed on the process. Therefore, LPs execute their own portion of the simulation based on the LVT. Causality errors can occur when LPs send messages to each other. This way, an LP may receive a message with timestamp earlier than its current LVT. Such events are referred to as *straggler events*. If a straggler event is received the LP will launch a *rollback* operation, where the LP recovers from the causality error by undoing the effects of all the events that were processed and had timestamp greater than the timestamp of the straggler event. Messages that were falsely sent to other processes now must be canceled, which is performed by sending *anti-messages*.

The Time Warp protocol consists of two parts [Mar97]: the *local control mechanism* and the *global control mechanism*. The local control mechanism which is provided in each Time Warp process is in charge of rollback operations which include: sending anti-messages, restoring the state of the LP, readjusting Local Virtual Time (**LVT**), etc. On the other hand, the global control mechanism takes care of global issues such as memory management, I/O operations, and termination detection.

The rollback mechanism requires defining three structures in each process: an *input queue*, to keep all the received events ordered by their virtual receive time (earliest

time-stamped event is on the top of the queue), an *output queue*, to keep a negative copy (i.e. anti-message) of each message that the process has recently sent out ordered in virtual send time, and a *state queue*, to keep a copy of all recent states of the process (this is used during the rollback when the state of the process has to be restored to that of saved prior to rollback). When rollback occurs due to a straggler message, two major actions take place at the LP [Liu06]. First, the state of the LP is restored to the last saved one which is now the top element of the *state queue* (this state was saved at time earlier than the virtual receive time of the straggler). Second, the process has to recover from the causality error by sending anti-messages to cancel the effects of already sent messages. All the anti-messages in the *output queue* whose timestamp is later than the straggler's receive time must be sent out. On the other hand, arrival of anti-messages at other processes will cause further rollback if the timestamp of the anti-message is less than the LVT of the receiving process. Therefore, anti-messages (just as positive stragglers) would cause rollbacks and further propagation of anti-messages. These are referred to as *secondary* rollbacks which result in cascaded rollbacks flooding the simulation system with anti-messages.

The global control mechanism defines the *Global Virtual Time* (**GVT**) which is an instantaneous global snapshot of the system and the wall clock time defined as follows [Fuj00]:

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

Unlike LVTs, the GVT never decreases [Fre02]. Hence, at any time of the simulation, GVT shows the minimum virtual time. This ensures that any event that was processed before GVT is 100% safe and will never rollback. Therefore, all events in the input and output queue whose timestamps are less than GVT can be safely removed from the queues. Also, all the states in the state queue (except the last one saved) with saved time older than GVT can be safely removed. The operation of deleting old information (messages and LP's states) is referred to as *fossil collection*. This mechanism avoids wasting system resources.

In Time Warp systems the global control mechanism is responsible for calculating and advancing GVT. The main issue is that, high GVT calculation frequency saves memory and allows faster response time and better space utilization but at the same time generates a significant processing overhead. On the contrary, lowering the GVT calculation frequency will generate less processing overhead but requiring more memory as well as slowing down the response time.

Jefferson's original Time Warp has been revised and optimized several times to reduce the processing overhead and especially overcoming the issues of cascaded rollback. Advanced optimistic techniques in this filed have been explained in [Fuj00]. We will discuss in details the Time Warp protocol used for our simulator as well as the optimizations to this mechanism in Chapter 3.

2.5. DEVS-BASED SIMULATION TOOLKITS

Based on previous studies [Liu06], in here we will give a brief review of the existing tools that implement DEVS theory and its extensions.

- ADEVS [Nut06] is a discrete event system simulator that provides a C++ library for constructing discrete event simulations based on the Parallel DEVS and Dynamic DEVS (dynDEVS) formalisms.
- DEVS-C++ [Zei96] is a DEVS-based high performance simulation environment which supports modeling of large-scale, high resolution landscape models using special form of C++ classes called *containers.*
- DEVS-Scheme [Zei93] is a knowledge-based real-time environment which implements the DEVS formalism in Scheme (a Lisp dialect) and enables the modeler to specify models directly in its terms.
- DEVS/CORBA [Zei99a] is a runtime infrastructure built on top of CORBA middleware which supports parallel and distributed simulation of DEVS formalism. DEVS/CORBA can be used in a larger network-centric environment to provide a combination of graphical process modeling, discrete-event simulation, animation, activity-based costing, and optimization functions.

- € DEVS/HLA [Zei99b] is based on High Level Architecture (HLA) [HLA00] implemented in C++ which explains how an HLA-compliant DEVS environment improves the performance of large-scale distributed modeling and simulation.
- € DEVS/Grid [Seo04] is a grid-compliant modeling and simulation environment based on DEVS formalism. It is implemented using Java and Globus toolkit for Grid computing infrastructure and supports high performance distributed simulation.
- € DEVSCluster [Kim04] is a CORBA-based, multi-threaded distributed simulator implemented in Visual C++. It supports simulation in heterogeneous network environments.
- € DEVSJAVA [Sar98] is a DEVS-based simulator implemented in JAVA that supports high-level modeling.
- € GALATEA [Dav00] uses an object oriented architecture to implement a simulation platform that offers a family of languages for modeling multi-agent systems in DEVS. GALATEA is the product of two lines of research: simulation languages based on Zeigler's theory of simulation and logic-based agents
- € JDEVS [Fil02] serves as an experimental framework for natural systems modeling techniques. It allows discrete-event, general purpose, object-oriented, component based, GIS connected, collaborative, visual simulation model development and execution. This experimental environment can be used to solve any complex problems solvable by discrete-event simulation and is especially suited for natural system modeling and simulation.
- € JAMES [Uhr01b] is a Java-based agent modeling environment for simulation of the activities in the area of agent-oriented simulation. It is based on a parallel, distributed version of DEVS, emphasizing states and state transition.
- € PyDEVS is a simulator developed in ATOM3 [Del02], a tool for multi-paradigm modeling. ATOM3-DEVS is a tool for constructing DEVS models and generating Python code for the PyDEVS simulator.
- € PowerDEVS [Kof03] is an integrated tool for hybrid systems modeling and simulation based on the DEVS formalism. It is implemented in C++ and allows construction of Atomic DEVS models graphically.

- SimBeans [Pra99] is a Component-based simulation framework based on DEVS and the JavaBean component model, developed as a modular, hierarchical composition of components.
- DEVS/P2P [Che04] is an interface for P-DEVS implementation over Peer-to-Peer message communication protocol. It supports hierarchical model partitioning, automatic coupling restructuring, automatic model deployment, and distributed/parallel/local simulation.
- DEVS/RMI [Zha06] is a natively distributed simulation system based on standard implementation of DEVS. It allows distributing simulation entities across network nodes seamlessly without any of the commonly used middleware. It is also built to support auto-adaptive and dynamic reconfiguration of simulations during run-time. DEVS/RMI approach is well suited for complex, computationally intensive simulation applications. It also provides an extremely flexible and efficient software development environment for simulation applications in a heterogeneous network environment.
- CD++ [Rod99, Wai02, Tro03] is an M&S toolkit developed in C++ that implements the original and Parallel DEVS and Cell-DEVS formalisms. It supports both standalone and parallel conservative simulations. This toolkit has been revised and tested in our research to realize distributed optimistic discrete-event simulations based on the Time Warp mechanism.
- SmallDEVS [Jan06] is a lightweight implementation of the original DEVS formalism which serves as an experimental software for research and education. It allows prototype-based object-oriented model construction, interactive modeling and simulation, and multi-simulation and reflective simulation.

CHAPTER 3 SOFTWARE ARCHITECTURE

In this chapter we will focus on the software architecture of the purely optimistic parallel CD++ simulator (PCD++) presented in [Liu06]. The layered architecture used by this simulator is the same layered design used in the parallel conservative simulator [Tro01]. We will present the layered architecture of the software in Section 3.1, followed by a more detailed discussion of each layer in Section 3.1 and Section 3.2.

3.1. LAYERED ARCHITECTURE

Figure 6 illustrates the layered architecture of the optimistic PCD++ simulator, where each layer only depends on the layers below it.

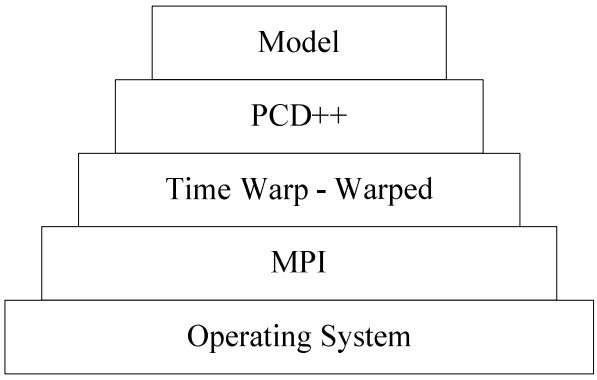


Figure 6. Layered architecture of the optimistic PCD++ simulator [Liu06]

The operating system resides on the bottom of the architecture. PCD++ uses Linux Operating System as the underlying platform for high-performance parallel and distributed computing. Above the Operating System lays the Message Passing Interface (MPI). MPI is a standard specification of message-passing library for high-performance communications on parallel machines and workstations clusters. The Operating System with the use of MPI provides the communication infrastructure for the PCD++ simulator.

We have used MPICH [Gro96] portable implementation of MPI which allows developing parallel and distributed applications. The WARPED [Rad98, War07] simulation kernel is our next layer which serves as a configuration middleware that implements the Time Warp mechanism and a verity of optimization algorithms. On top of the WARPED kernel we have the optimistic PCD++ simulator implementing the Parallel DEVS and Cell-DEVS formalisms. The PCD++ simulator provides the framework for creating and executing DEVS and Cell-DEVS models in distributed environments using the Time Warp protocol. Finally, the top most layer is the DEVS or Cell-DEVS model created in CD++.

3.2. THE TIME WARP LAYER - WARPED KERNEL

WARPED [Rad 98, War07] is a public domain simulation kernel originally developed at the University of Cincinnati to provide an implementation of Jefferson's original Time Warp algorithm [Jef85]. The WARPED kernel is an attempt to make a freely available Time Warp simulation kernel that is easily ported, simple to modify and extend, and readily attached to new applications. The services provided by WARPED were used to implement our CD++ simulators: the conservative PCD++ simulator [Tro01], and the optimistic PCD++ simulator [Liu06]. This kernel serves as a middleware to implement CD++ simulator by allowing the use of Time Warp optimizations.

WARPED is developed using C++ language and compiles with open source GNU C++ compiler, g++. WARPED kernel uses MPI [MPI95] message passing standard for communication among distributed and parallel computing nodes. As mentioned in the previous section, for our PCD++ simulators we have used MPICH [Gro96] the freely available implementation of MPI which is ported to different platforms including Linux. Figure 7 illustrates the layout of how LPs and their simulation objects communicate in WARPED using the MPICH message passing interface. As shown on the figure, there exist two types of communications [Mar96]: *direct communication* for message exchange among local simulation objects (the ones sitting of the same LP), and *MPI communication* for message exchange among remote simulation objects (the one hosting on different LPs).

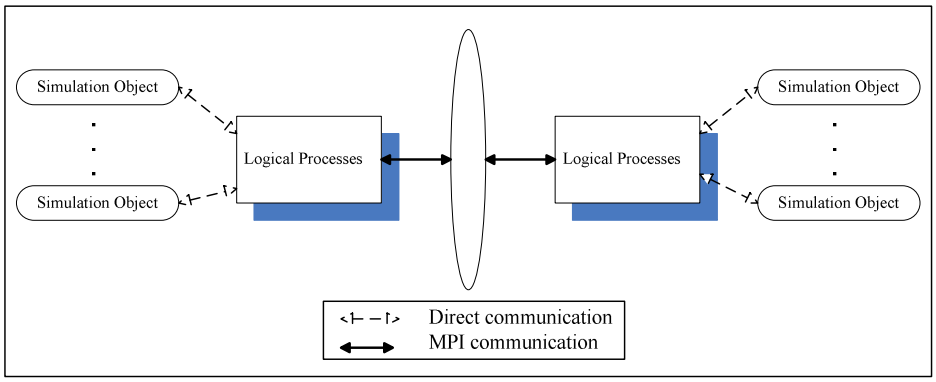


Figure 7. Structure of LPs and simulation objects in WARPED

WARPED provides an application program interface which includes base classes for simulation objects (Warped objects), object's state, and the events which get exchanged among simulation objects. This API allows users to create their own application by creating new classes derived from the ones offered. Also, the user has the opportunity to redefine functionalities by overloading the inherited methods without the need of changing kernel's code. Furthermore, WARPED provides a simple definition of time (again can be redefined by the user) and functions to perform consistent I/O operations.

The WARPED API is used to model objects (simulation objects) as entities which exchange messages (time-stamped events) with each other and respond to events by applying them to their internal stats. Thus, the kernel provides functionalities for sending and receiving events by simulation objects. On the other hand, since WARPED kernel is used to present an interface to Jefferson's Time Warp algorithm, it has to offer a mechanism suited for potential rollbacks. The main issue in handling rollbacks is saving and restoring the object's states. To this extent, the WARPED kernel provides the capability of defining each object's state to support periodic state saving during rollbacks and recovery periods.

The WARPED kernel provides two sets of synchronization mechanisms, namely NoTime and TimeWarp. The first one implements a conservative behavior, thus was used in our Conservative PCD++ simulator [Tro01], where the later one implements Jefferson's Time Warp optimistic algorithm, therefore used by our Optimistic PCD++ simulator [Liu06].

3.2.1. WARPED functionalities

Here we will present the major functionalities of the WARPED kernel based on previous studies presented in [Rad98, Mar99]. Figure 8 illustrates a summary of these functionalities. A detailed discussion about each module is given in [Liu06].

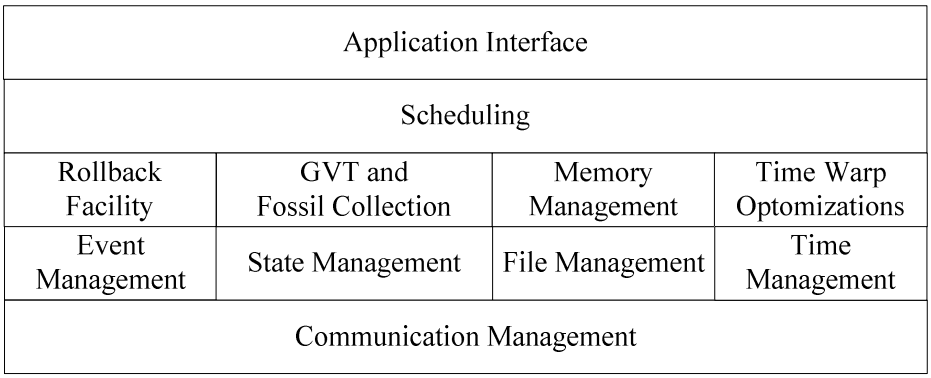


Figure 8. Major functionalities of WARPED kernel [Liu06]

Based on Jefferson's definition, the simulation is carried out by assigning each part of the simulation to one Time Warp process (the *simulation objects*). The WARPED kernel groups simulation objects into partitions called "clusters" [Rad98]. In each *partition* of *cluster*, the simulation objects are assigned to the available physical processors [Low99]. As shown in Figure 9 the WARPED kernel consists of three clustering levels.

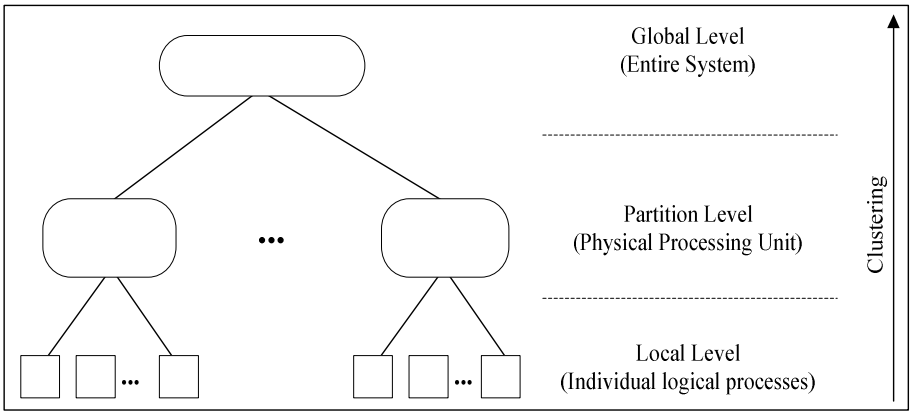


Figure 9. The tree structure of clustering scope levels in WARPED kernel [Low99]

The lowest level is consisted of the simulation objects implementing the Time Warp *local control mechanism*. The level above it is the partition level which is consisted of the physical processors who the simulation objects host on. In Time Warp definition each physical processor is referred to as a Logical Process (LP) which encapsulates one

or more simulation objects. The local simulation objects although grouped by the same LP, are not synchronized. This is due to the fact that each simulation object (Time Warp process) maintains its own LVT [Rad98]. The last (top most) level is the entire system consisting of multiple partitions working together to implement the Time Warp *global mechanism*.

The WARPED kernel provides a variety of optimization strategies to optimize kernel performance. Optimization algorithms implemented in the kernel are: fixed-sized and dynamic message aggregation [Che98] algorithms to minimize inter-LP communication overhead without harming the progress of the simulation; static and adaptive polling [Sha99] algorithms for optimizing the message reception behavior; one anti-message per rollback strategy [Mar99] for reducing the number of anti-messages during rollbacks; lazy and dynamic cancellation algorithms [Lin91] to exploit message independency and take advantage of the parallelism available within a Time Warp logical process; and algorithms for adjustment of runtime parameters using external agents [Rad97] to reduce the overhead of the operations. Many modifications were performed by previous research [Liu06] to incorporate three core Time Warp optimization strategies into the optimistic PCD++ simulator. The following section will describe these major optimizations in details.

3.2.2. Time Warp optimizations of PCD++ simulator

The WARPED layer of our optimistic PCD++ simulator has been modified by previous research [Liu06] to include three major optimization strategies. In general, optimization strategies aim at reducing the operational overhead of the Time Warp mechanism, and exploiting more parallelism than is available in the basic protocol [Low99]. In [Liu06], a flexible user-controlled state-saving mechanism was implemented in the State Management module. Also, the fossil collection algorithm was revised in the GVT and Fossil Collection module to integrate the periodic state-saving strategy. Furthermore, the Rollback Facility was enhanced to allow sending only one anti-message per rollback and reduce the number of anti-messages required to be sent to a certain extent. The following points will describe each of these optimizations:

- **User-Controlled State-Saving Mechanism**

The WARPED kernel provides two types of state-saving strategies, namely the *copy state-saving* (CSS) strategy and the *periodic state-saving* (PSS) strategy. Each of these strategies is handled by a different type of manager: *StateManager* enforcing CSS strategy, and *InfreqStateManager* implementing PSS strategy. The *StateManager* has the responsibility of saving the state of a simulation object after executing each event, while the *InfreqStateManager* only saves a simulation object's state infrequently every number of events. The simulator developer has only the option of selecting one of state managers at compile time. This selection will then apply to all simulation objects and thus all of them will use the same type of state managers throughout the simulation. This restricted selection mechanism has a major drawback which is eliminating the possibility of choosing different type of state-saving mechanism for different simulation objects based on their specific requirements at runtime. The solution presented in [Liu06] introduces a new state-saving mechanism. This new strategy is a two-level *user-controlled state-saving* (UCSS) mechanism in the kernel which provides the simulator developer to utilize more flexible and efficient state-saving strategies at runtime. As shown in Figure 10 the UCSS mechanism has a two-level structure which enables every simulation object to switch to "*skip-state-saving*" mode and as a result skip the state-saving operation. This mechanism allows simulation objects to make state-saving decisions based on application-specific criteria.

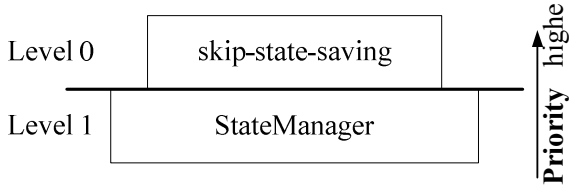


Figure 10. UCSS structure [Liu06]

- **Fossil Collection Algorithm Enhancement**

The GVT manager of WARPED kernel reclaims all but the last saved state older than the GVT along with the messages with timestamps less than the GVT in the input and output queues. As a result, the GVT always indicates the least timestamp of any potential future straggler and anti-message that can be received by any of the existing simulation objects. In other words, it is the minimum time of any rollback that may occur

in one of the processes. When rollback occurs, the state of the process is restored to the last one saved prior to the rollback which is the one with virtual saved time earlier than the GVT. Notice that this state is also the one that was left in the state queue during last fossil collection. Therefore, this mechanism of WARPED kernel works successfully and rollback operation performs exactly as expected even in the case where rollback time is equal to the GVT. However, this mechanism fails to work successfully when the periodic state-saving (PSS) strategy is used [Liu06]. The failure scenario is as follows: when the state of a process is saved infrequently, the restored state which is the last one available in the state queue could be saved at virtual time much earlier than the current GVT. Therefore, although the state restoration is performed correctly, but due to fossil collections, all events with timestamp between the time of the restored state and the GVT are already removed from the queues. As a result, runtime crash occurs. To overcome this problem, the fossil collection mechanism was revised in such a way that fossil collection is no longer performed using computer GVT [Liu06]. Thus, in the new algorithm, a minimum value among the virtual time of the last states saved older than the GVT is calculated for all the processes mapped on a LP. This is the value used to do fossil collection.

- **One Anti-message per Rollback**

During rollback all messages saved in output queue with virtual send time equal to or greater than the rollback time are sent to their original receivers as anti-messages. However, there might be multiple anti-messages with different timestamp that must be sent to the same receiver. This will result in multiple rollbacks at the receiver and consequently a flood of anti-messages exchanged between the processes. The communication overhead associated with these message exchanges is very high. To resolve this issue, when a process has several anti-messages to send to another process, instead of sending them all, it is clearly enough to only send the one with the earliest timestamp [Lub91]. Using this fact, the rollback mechanism was revised to send only one anti-message per rollback and as a result significantly reduce the number of anti-messages that need to be sent to a certain process.

CHAPTER 4 CONSERVATIVE VS. OPTIMISTIC PCD++ SIMULATOR

Part of our research is to analyze the performance of our two existing parallel CD++ simulators, namely Conservative PCD++ simulator [Tro01] and Optimistic PCD++ simulator [Liu06]. In this chapter we will look at the design and implementation of these two simulators and compare their structures as well as functionalities in parallel and distributed simulations. Section 4.1 will introduce the conservative PCD++ simulator, while Section 4.2 will present the optimistic version.

4.1. THE CONSERVATIVE PCD++ SIMULATOR

Conservative PCD++ simulator [Tro01] was the first attempt to reduce simulation time in CD++ using distributed execution of models. Distributed simulation with Parallel CD++ speeds up the execution of both DEVS and Cell-DEVS models in comparison to the stand-alone simulator [Gli04]. The first parallel simulator of CD++ was based on a pessimistic (conservative) approach exploiting 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. Based on this structure, all events with the same timestamp are scheduled to be processed simultaneously on the available nodes. The simulator introduces two different types of coordinators; *master* and *slave* to reduce inter-process communication. The simulator consists of a hierarchical structure creating a one-to-one correspondence between the model components and simulation objects.

4.1.1. Parallel DEVS abstract simulator

The DEVS formalism separates the model from the actual simulation. The *abstract simulator* implements this mechanism by creating a one-to-one correspondence between the model and the simulation entity as illustrated by Figure 11. The abstract simulator for Parallel DEVS was first proposed by [Cho94b] but it lacked in differentiating among intra-process messages and inter-process messages. Thus, the design and implementation was revised by [Tro03] to distinguish among these messaging paradigms and as a result

restrict the number of messages over the network (inter-process messaging) to a minimum.

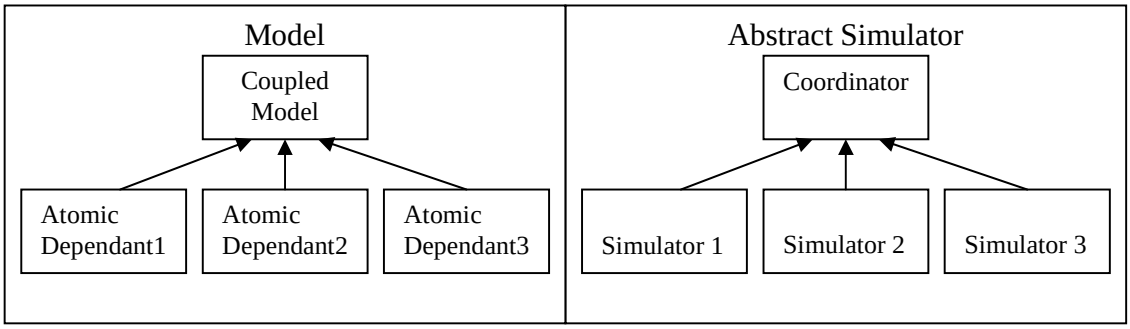


Figure 11. Correspondence between the model and the DEVS processors [Tro01]

The simulation is carried out by DEVS processors which are of two types: *simulator* and *coordinator*. The *simulator* represents an atomic DEVS model, where the *coordinator* is paired with a coupled model. The simulator is in charge of invoking the atomic model's transition and external event function. On the other hand, the coordinator has the responsibility of translating its children's output events and estimating the time of the next imminent dependant(s). As shown in Figure 11 every coordinator has a set of child DEVS processors. At the beginning of the simulation, one logical process will reside on each machine (physical process). Then, each logical process will host one or more DEVS processors. This implies the fact that not all of a coordinator's children are necessarily sitting on the same logical process. Due to the one-to-one correspondence, each coupled model is mapped to only one coordinator. A coordinator communicates with its child processors through intra-process messaging if they reside on the same logical process, and through inter-process messaging if they are sitting on remote logical processes. Figure 12 shows a scenario at which a coupled DEVS model consisting of six atomic components is simulated using this simulator. The coordinator itself and three of its child processors are on the same logical process (LP0), where the other three child processors are hosted on another logical process (LP1). When the number of remote child processors of a coordinator is high, this design mechanism will lead to considerable overheads due to inter-process messages that are sent back and forth among the coordinator and its child processors. To overcome this issue, the concept of *Master and Slave Coordinators* was introduced [Tro01].

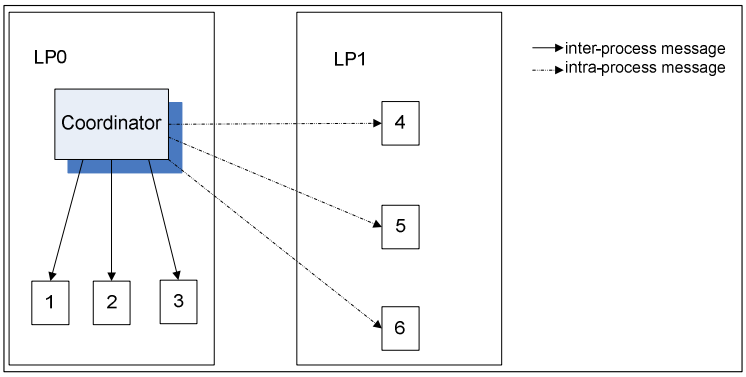


Figure 12. A single coordinator with remote and local child processes

In the new design a coordinator is assigned with each logical process. As a result, all child processors will have a local coordinator through which they can communicate with remote child processors. In such a scenario, there exist two different types of coordinators: *Master Coordinator*, and *Slave Coordinator*. The master coordinator is responsible for synchronizing the model execution, interacting with upper level coordinators, and exchanging messages among the local and remote model components. The slave coordinator is responsible for message exchange among the local model components, and forwarding local components messages to the master coordinator if it resides on another logical process. Figure 13 shows the class diagram of these two coordinators.

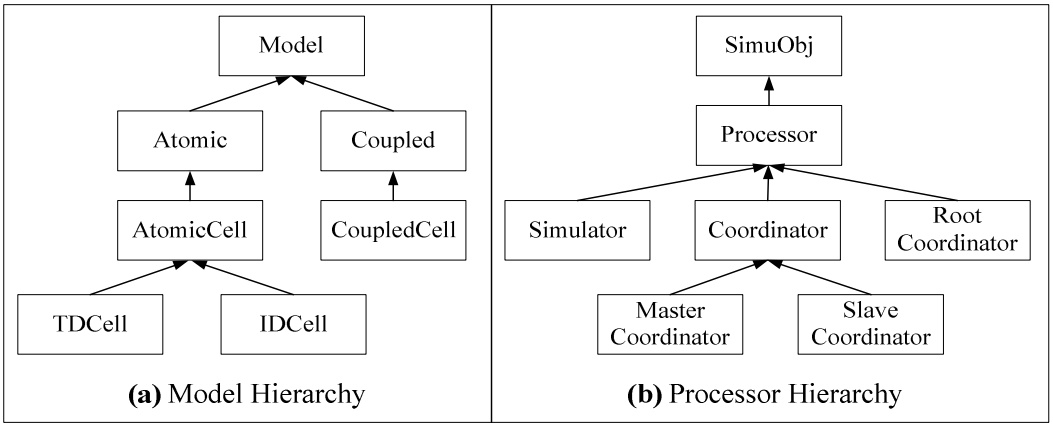


Figure 13. Master and slave coordinators class diagram

Figure 14 illustrates the revised scenario of Figure 12 using the master-slave structure.

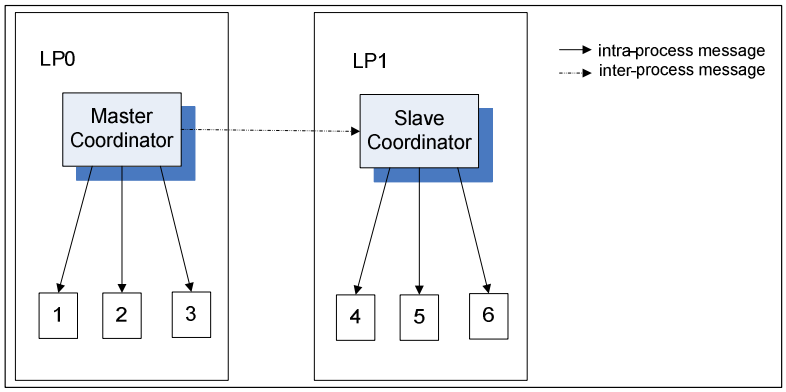


Figure 14. The master-slave coordinator structure

The master-slave coordinator structure organizes DEVS processor into a hierarchy which does not have a one to one correspondence with the model hierarchy. Thus, a parent-child relationship that takes into account the existence of master and slave coordinators must be defined as follows [Tro01]:

i. For each *simulator*, the parent coordinator will be the parent's model local processor.
ii. For each *slave coordinator*, the parent coordinator will be the model's *master coordinator*.
iii. For each *master coordinator*, the parent coordinator will be the parent's model local processor; just as if it was a *simulator*.

Based on this hierarchy, the conservative PCD++ simulator was implemented [Tro01]. Under this design, the simulation advances as a result of exchange of messages in the form of (*type*, *time*) between the parent and child DEVS processors. Two different types of messages exist: 1) the *synchronization messages*: (@, *t*), (*, *t*), and (*done*, *t*), 2) the *content messages*: (*y*, *t*) and (*q*, *t*). The ***collect message*** (@, *t*) is sent from a parent DEVS processor to its imminent children to tell the children to send their outputs. The ***internal message*** (*, *t*) is sent from a parent DEVS processor to its imminent children to tell the children to invoke their transition function (either an external, internal, or confluent transition). The outputs produced by a model are translated into ***output messages*** (*y*, *t*) which are exchanged among a child DEVS processor and its parent. Finally, the ***external messages*** (*q*, *t*) represent the external messages arrived from outside the system or the ones generated as a result of an output message being sent to an influencee.

Each type of message is handled differently by the *simulator*, *master coordinator*, *slave coordinator*, and *root coordinator*. The behaviors of these processors define the abstract simulator. Next we will look at the behavior of each type of DEVS processors at handling different messages.

4.1.2. Message definitions

In the following discussion, the form *(type, t)* is used to denote a message of *type* that has a receive time of *t*. The external, output, collect, internal, and done message are presented as *q*, *y*, @,*, and *done* respectively.

€ **Simulator**

The *simulator* is responsible of invoking the atomic model's $\lambda(s)$, δ_{ext}, δ_{int}, δ_{con} functions. The description that follows is a revised version of the original one presented in [Cho94b] which has been modified by [Tro01].

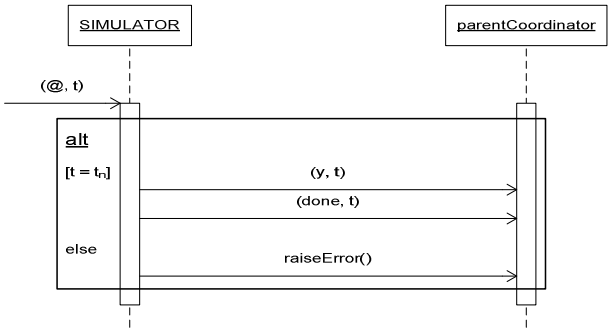


Figure 15. Simulator algorithm for (@, *t*)

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

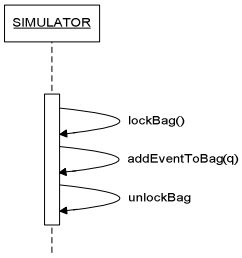


Figure 16. Simulator algorithm for (*q*, *t*)

When an external message is received, it is inserted into the bag and will gets executed at the right time.

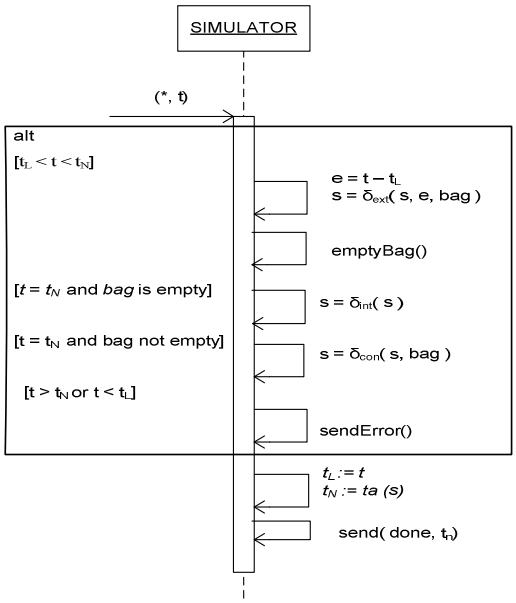


Figure 17. Simulator algorithm for (*, *t*)

Reception of (*, *t*) message indicates that a model's transition function must be executed. The transition function that must be executed is selected based on *t* and the contents of the queue. If $t < t_N$ and the queue is not empty, then δ_{ext} must be executed. If $t = t_N$ then it is the time for an internal transition; either the queue is empty (i.e. no external messages have been received) therefore δ_{int} is executed, or the queue is not empty (i.e. there are external messages) thus δ_{con} is executed.

€ Master Coordinator

A coordinator, whether master or slave, is responsible for the simulation of a coupled model. It translates output events to input events and keeps track of the imminent components. Each coordinator has a set of child processors which correspond with the coupled model components. For a master coordinator the set of child processors consists of: a set of slave coordinators, a set of local child simulators, and a set of local child master coordinators. A DEVS processor is said to be local if it resides on the same processor.

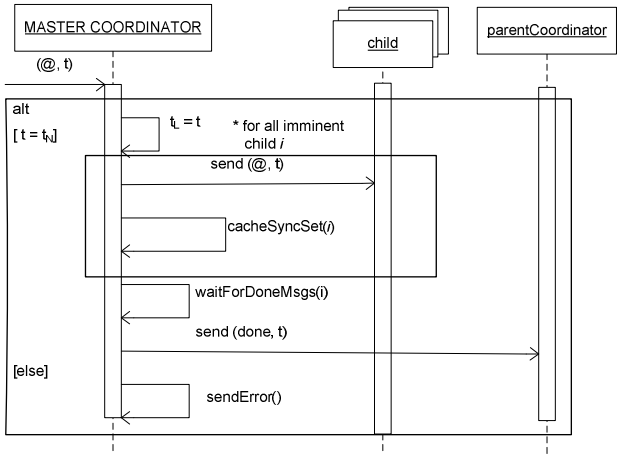


Figure 18. Master coordinator algorithm for (*@*, *t*)

When a (*@*, *t*) is received at the master coordinator, if $t = t_N$, the collect message will be forwarded to all imminent child processors with minimum t_N and the imminent processor will be cached in the synchronize set. The master coordinator will then wait for all imminent processors to send back a *done* message. Then, the master coordinator will send a *done* message to its parent coordinator indicating that it has responded to the received (*@*, *t*) message correctly.

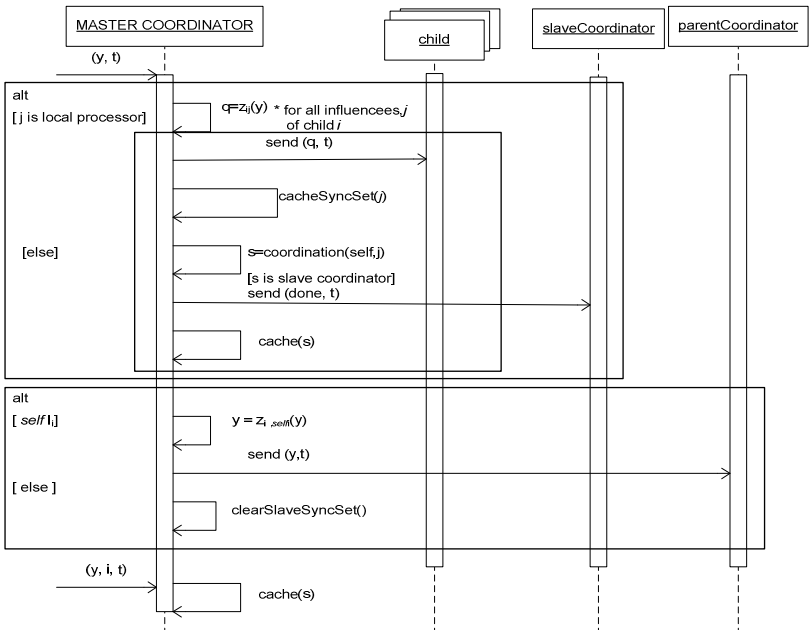


Figure 19. Master coordinator algorithm for (*y*, *t*)

Two different scenarios may occur at the *master coordinator* upon reception of an output message:

1. An output message (*y, t*) is received from a child *i* that is not a *slave coordinator*.
2. An output message (*y, i, t*) is received from a *slave coordinator* which has received a (*y, t*) from a local child *i*.

The *slave-sync* set is used for synchronization so an output message does not get sent twice to the same *slave coordinator*. To reduce the number of inter-process messages sent across the network, instead of forwarding a (*q, t*) message to the slave coordinator, a (*y, i, t*) is sent. A *slave coordinator* might be the parent coordinator for more than one of the influencees of *i*. If (*q, t*) messages were to be forwarded, then there will be one (*q, t*) message for each influencee of *i*. For Cell–DEVS models, this can be a significant overhead. Instead, just one (*y, i, t*) message is sent across the network then the *slave coordinator* will generate the appropriate (*q, t*) messages for each influencee. Then based on [Cho94b] design, all children ready for a transition are cached in a *synchronize* set to later on distinguish active from inactive components.

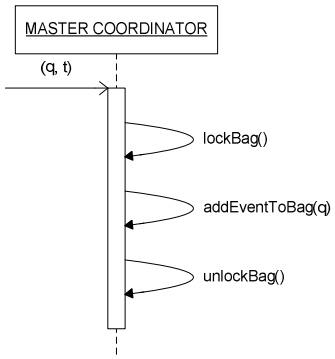


Figure 20. Master coordinator algorithm for (*q, t*)

As in *Simulator*, when an external message is received at the master coordinator, it is inserted into the bag and will get executed at the right time.

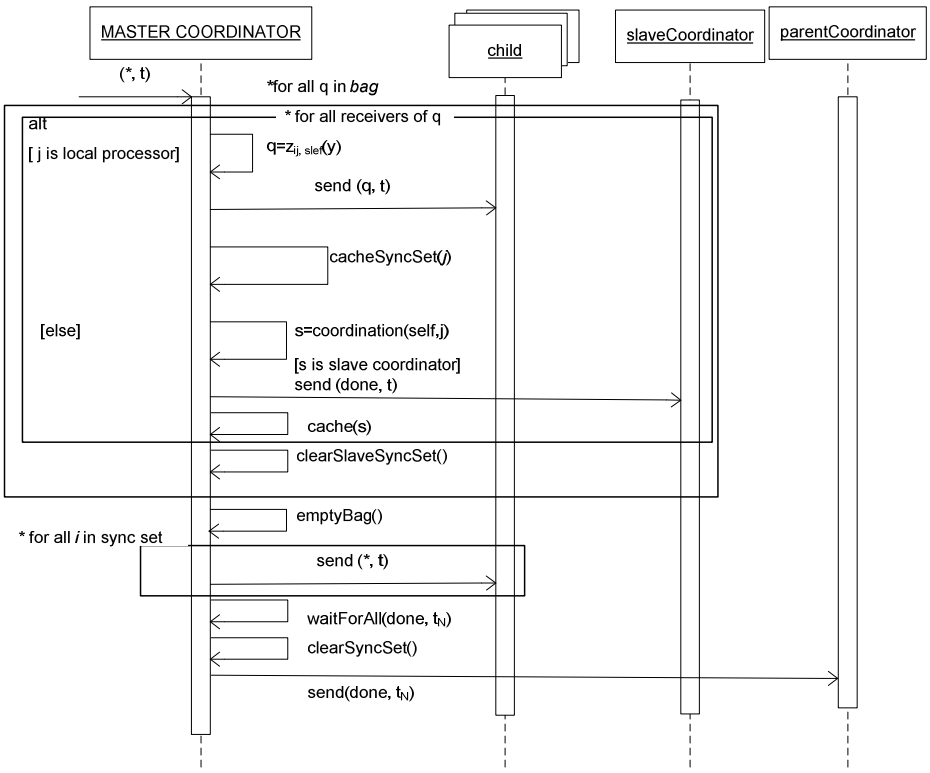


Figure 21. Master coordinator algorithm for (*, *t*)

When the output messages are sent down to child processors, if the message is to be forwarded to a *slave coordinator* the *z* translation will not be applied. Instead, the original *q* message will be sent. This is why must make sure that a message is not forwarded twice to a *slave coordinator*. As mentioned before, the *slave-sync* is used for this purpose.

€ Slave Coordinator

The *slave coordinator* differs from the *master coordinator* in only one aspect: when a message has to be sent to a processor that is not local, it will be sent to the *master coordinator* instead. Both the master and slave coordinators handle a (@, *t*) in a same manner. However, the set of child processor of a slave coordinator differs from that of a master coordinator. For a slave coordinator the set of child processors consists of the set of local child *simulators* plus the set of local child *master coordinators* only.

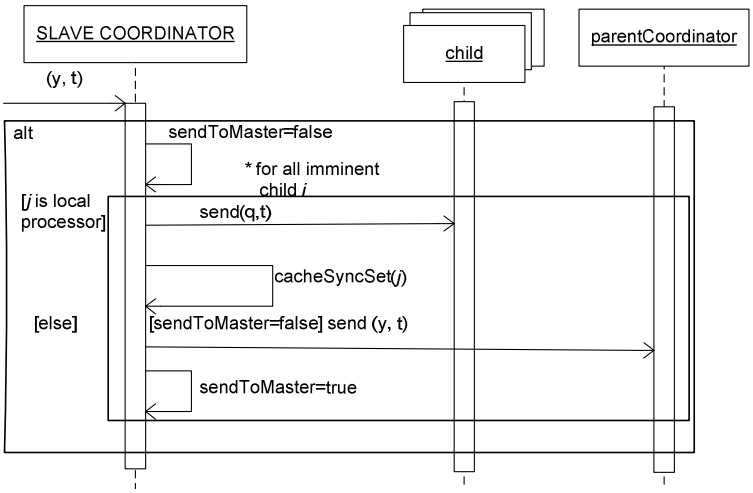


Figure 22. Slave coordinator algorithm for (*y*, *t*)

When an output event is received from a child *i*, the *slave coordinator* sorts the message to the influencees of *i*. If any influencee is local, the *z* function is applied and a (*q*, *t*) message is sent. If there are remote influencees, then the output event is sent to the *master coordinator*, which in turn will sort the message to other *slave coordinators* if necessary. Notice that, only one (*y*, *t*) message must be forwarded to the *master coordinator*. On the other hand, when the *slave coordinator* receives an output event that has been forwarded by the *master coordinator* on behalf of child *i*, it will handle the event as if *i* had been local, but no (*y*, *t*) messages will be forwarded back to the *master coordinator* if there is a remote influencee. This is to avoid infinite loops of messages being forwarded back and forth. The behavior of the slave coordinator upon reception of other messages is identical to the master coordinator, thus will not be investigated here.

€ Root Coordinator

The *root coordinator* is a special processor that is above the topmost coordinator. It is responsible for driving the simulation and advancing the virtual simulation time. The root coordinator is also capable of handling external events which are inserted into a sorted queue of messages.

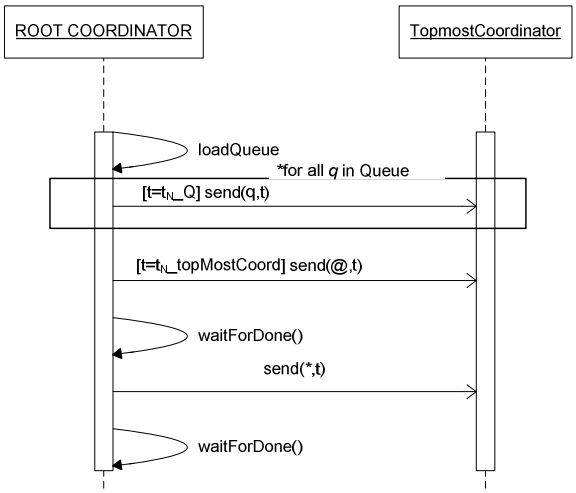


Figure 23. Root coordinator algorithm

4.2. THE OPTIMISTIC PCD++ SIMULATOR

In Section 4.1 we looked at a parallel CD++ simulator which uses a hierarchical structure creating a one-to-one correspondence between model components and simulation object. However, due to the layout of the design, the communication costs associated with this structure is considerable. This led to proposing a flat simulation mechanism rather than the traditional hierarchical one to reduce the overhead in communication by reducing the number of exchange messages (especially inter-process message) to minimum. This is achieved by simplifying the underlying simulator structure, while keeping the same model definition and preserving the separation between model and simulator [Gli04]. Researchers have shown that flat simulators outperform hierarchical ones significantly [Kim00, Gli02a, Gli02d, Glin04, Kim04, Liu06]. Moreover, previous research [Gli02b, Gli02c] showed that although the hierarchical simulator presented in [Tro01] tried to reduce the communication overhead by introducing two specialized DEVS coordinators, but in some cases the communication overhead was still significantly high. After the Conservative PCD++ simulator [Tro01] which was a hierarchical Parallel DEVS simulator, [Gli04] was the first attempt to re-design the parallel CD++ simulator to adopt a flattened structure. The proposed simulator [Gli04] also modified the parallel mechanism to an Optimistic algorithm supported by the use of Time Warp kernel. The

whole abstract simulator was redesigned to reflect the two major modifications; i.e. the departure from conservative-based simulator to an optimistic-based simulator, and flattening the structure of the simulator. As a result, a new Parallel DEVS simulator was implemented which dealt with the communication overhead dilemma by using a flattened structure rather than the old hierarchical approach. However, [Liu06] improved the Optimistic PCD++ simulator further and implemented many optimization strategies and enhanced the parallel and distributed simulation significantly. Thus, we used the latest Optimistic PCD++ simulator [Liu06] for this research.

The following section will describe the abstract simulator in terms of its design layout as well as the functionalities of each DEVS processor.

4.2.1. Parallel DEVS abstract simulator

The flattened architecture of the Parallel DEVS introduces two new types of DEVS processors, namely *Flat Coordinator* (FC) and *Node Coordinator* (NC), to reduce the communication overhead. The flattened structure keeps the modeling framework unchanged and uses a flattened approach for overlaying the coordinators. Figure 24 shows the class hierarchies in the modeling and the simulation framework.

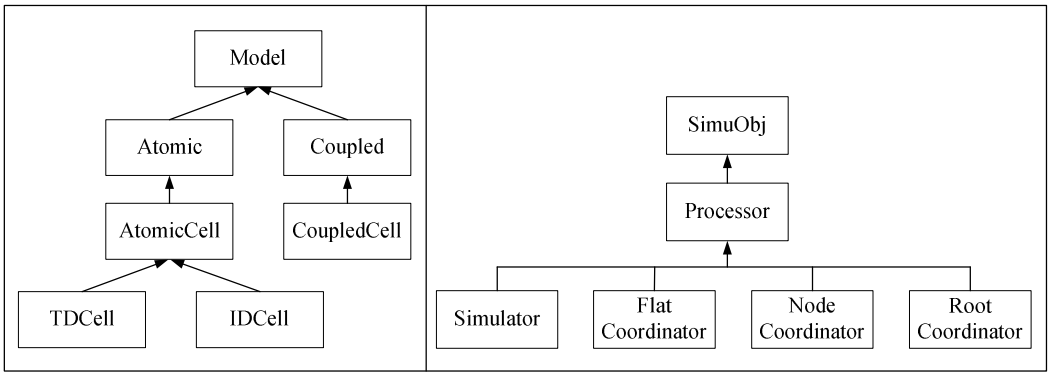


Figure 24. Model and processor hierarchies in Optimistic PCD++ [Liu06]

As shown on the above figure, there are four different types of simulators used in the new design: *Simulator*, *Flat Coordinator*, *Node Coordinator*, and *Root Coordinator*. The simulation is carried out by these processors in a distributed fashion among the available machines. To better illustrate this design, let's consider the scenario that was discussed in Section 4.1 where a coupled DEVS model consisting of six atomic

components is simulated using the flattened PCD++ simulator. Figure 25 shows the model specification and its partitioning on two machines.

The optimistic PCD++ simulator [Liu06] uses simulation techniques that are based on optimistic synchronization protocol which extend the conservative approach used in Parallel CD++ simulator [Tro01, Tro03] that was described in Section 4.1. The optimistic PCD++ simulator uses the optimistic synchronization protocol provided by the WARPED kernel to implement a distributed version of CD++. The flattened architecture used by this simulator outperforms the previous hierarchical simulator [Tro01] by reducing the communication overhead significantly.

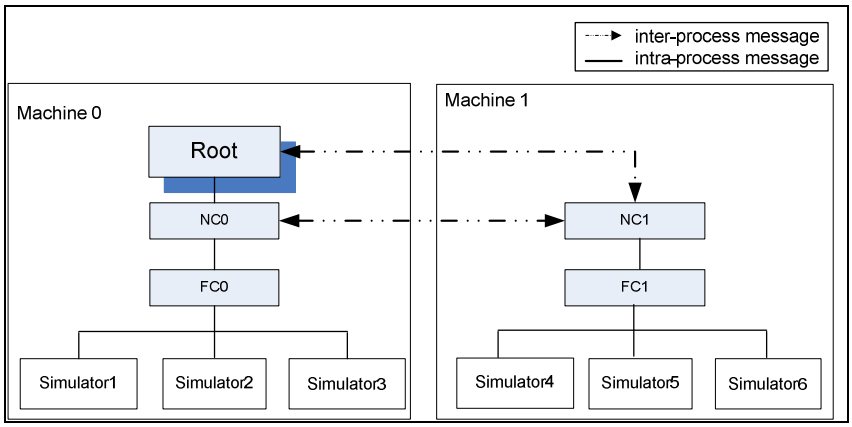


Figure 25. Distributed processor structure

As presented in Figure 25, one LP is created on each machine encapsulating the DEVS processors. Only one Root is created on machine 0 (LP0) which interacts with other NCs using inter-process messaging (for remote NC) and intra-process messaging (for local NC). The Root coordinator is in charge of starting the simulation and performing I/O operations among simulation system and the surrounding environment. Only one NC is created on each machine and acts as the local central controller on its hosting LP. The NC is the parent coordinator for FC and routes remote messages received from the Root or from other remote NCs to the FC. The Simulators are the child processors of the local FC which represent the atomic components of DEVS and Cell-DEVS models and responsible for executing the abstract functions defined in their associated atomic model. When a Simulator needs to communicate with a remote Simulator residing on another LP, it sends the message to its FC, then the message is forwarded to the NC above it. Once the message is at the NC, it will further be routed to

the destination NC. Even if two Simulators are local (sitting on the same LP), they need to forward their messages to their parent FC. There is no direct communication among Simulators; all messages must be forwarded to the parent FC. This is why the FC is known as the local central controller of its hosting LP.

4.2.2. Message definitions

PCD++ processors exchange two categories of messages: *content messages* and *control messages*. The first category includes the external message (*x*) and the output message (*y*), and the second category includes the initialization message (*I*), the collect message (@), the internal message (*), and the done message (*D*). To describe these messages, external and output messages are used to exchange simulation data between the models, initialization messages start the simulation, collect and internal messages trigger the output and the state transition functions respectively in the atomic DEVS models, done messages handle synchronization by carrying the model timing information. The simulation is executed in a message-driven manner. Each type of PCD++ processor, defines its own receive functionality for each type of messages. In this section, we present what happens at each PCD++ processor including the *Simulator*, *Flat Coordinator*, *Node Coordinator*, and *Root Coordinator* upon reception of different types of messages.

- **Simulator**

The Simulator algorithm for initialization message is defined as follows:

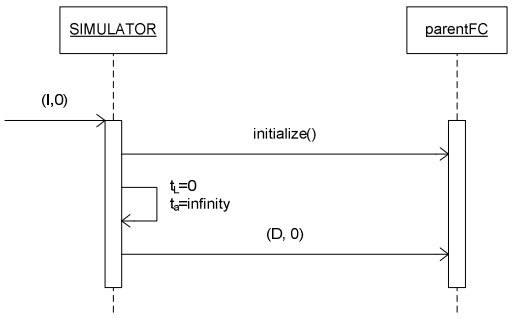


Figure 26. Simulator algorithm for (*I, 0*)

As defined in DEVS formalism, two variables are used in the simulator to record its current simulation time (t_L) and the value of *sigma* (ta). Using these two values, the value of *absolute next time* (denoted as t_N) is calculated as t_L + ta. Upon receiving the initialization message, (*I*, 0), the Simulator resets t_L to the timestamp of the message, therefore the Simulator's virtual time now is equal to zero. Then, the simulator initializes the variables defined in its associated atomic model, and after that, it informs its parent FC of the value of t_a by sending a done message stamped with time 0.

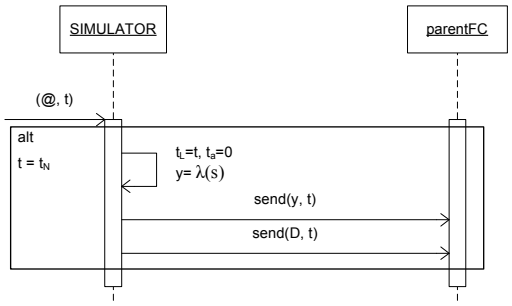


Figure 27. Simulator algorithm for (*@*, *t*)

When a (*@*, *t*) message is received, the Simulator invokes the output function (λ) of the atomic model and as a result an output message (*y*, *t*) is sent to the FC. After this, the Simulator will send (*D*, *t*) to the FC with $t_a = 0$ to indicate that it is imminent.

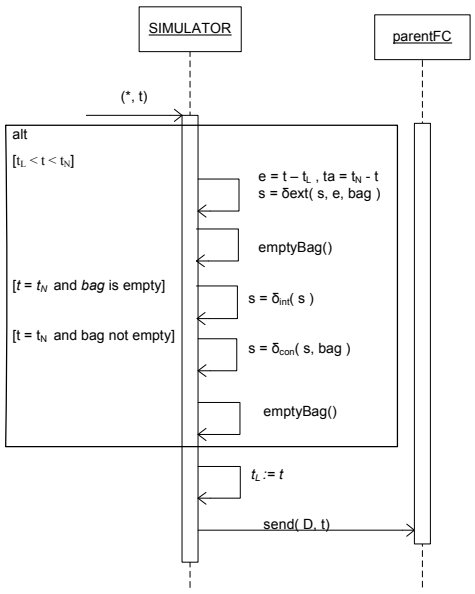


Figure 28. Simulator algorithm for (*, *t*)

Following the collect message, a (*, *t*) will arrive to trigger internal/external/confluent function of the atomic model depending on the timing of the message and the status of the Simulator's message bag.

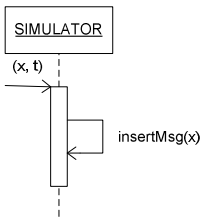


Figure 29. Simulator algorithm for (*x*, *t*)

The last message that may arrive at the Simulator is (*x*, *t*) which is simply inserted into the Simulator's message bag. Note that, only external messages with identical timestamp can be inserted into the message bag at a given simulation time. Before adding further messages with a different timestamp, the existing messages must be processed and the bag be cleared in the receive function for internal message. In other words, an internal message will always arrive in between two consecutive batches of external messages.

€ Flat Coordinator

The FC, sitting in between the NC and the Simulators, performs three tasks: synchronizing the execution of all child Simulators, routing messages exchanged among its children, and delivering to its parent NC those messages that are sent from its children to the environment or to other remote Simulators. To accomplish the first task, the FC finds its imminent children with the minimum absolute next time and records them in a structure called *synchronize set*. It also uses a variable, *doneCount*, to keep track of the number of done messages it should receive from its children. This variable is used to implement a simple barrier. The FC only passes control to its parent NC after these children (the number is given by *doneCount*) have finished their previous computation. The other two tasks rely on the model coupling information that is loaded into the *main administrator* of the simulation administration facility during the bootstrap operation.

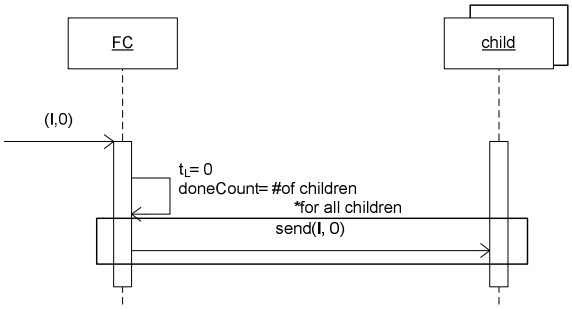


Figure 30. FC algorithm for (*I*, *t*)

When (*I*, 0) is received, the FC records the total number of its children in a variable named as *doneCount* then forwards the (*I*, 0) message to each child. After this, the FC waits for all its children to respond to this initialization by sending back a (*D*, 0). The FC will only pass the control over to the NC if all its children have finished their previous computation and have sent done messages as notification messages.

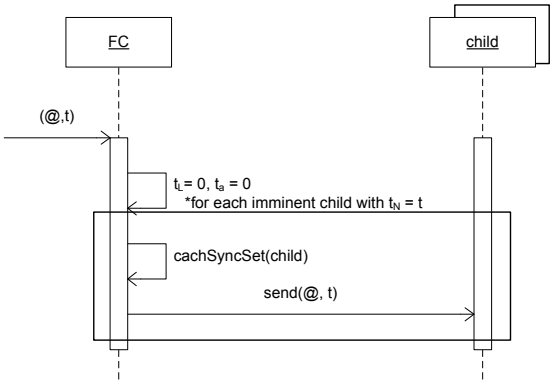


Figure 31. FC algorithm for (@, *t*)

Upon receiving a (@, *t*) message, the FC forwards it to all imminent Simulators and will keep a record of this for later use (to know which children need to do state transitions when (*, *t*) is received).

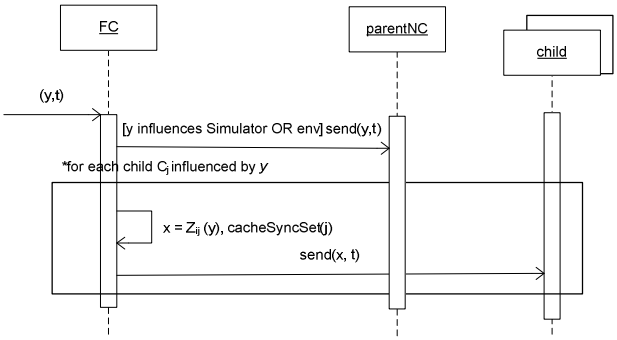


Figure 32. FC algorithm for (*y*, *t*)

Moreover, when (*y*, *t*) is received, the FC searches the model coupling information to find out the correct destination. The destination is either an input port on an atomic model, or an output port on the topmost coupled model.

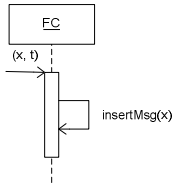


Figure 33. FC algorithm for (*x*, *t*)

In case of receiving (*x*, *t*) message, the FC will simply insert the message into its message bag.

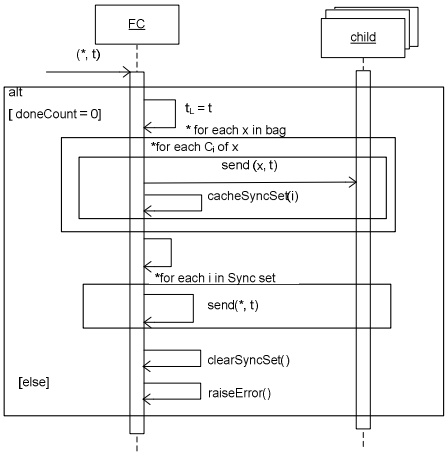


Figure 34. FC algorithm for (*, *t*)

Upon receiving (*, *t*) message, the external messages inside the FC's message bag are flushed to the local receiving Simulators. This will trigger the imminent Simulators to perform a state transition.

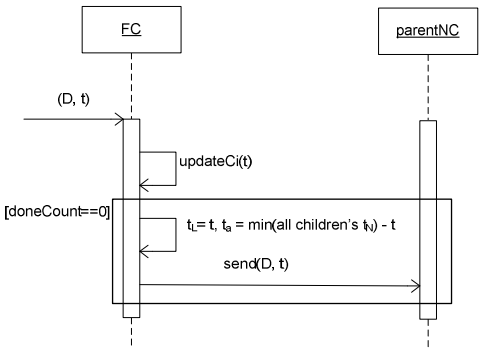


Figure 35. FC algorithm for (*D*, *t*)

Finally, when a (*D, t*) message is received from a child Simulator, the FC updates the child's t_N to the sum of the current simulation time and the *sigma* value carried by the received (*D, t*) message.

€ Node Coordinator

Each LP has one NC that acts as the local central controller in charge of the sequential simulation on the hosting machine. It has a single child, the FC underneath. The NC on machine 0 also has a parent, the Root. The NC plays a very important role in the simulation as summarized below:

1) It takes care of the inter-LP communication among the Simulators. The messages exchanged between the NCs is handled using a special structure, the *NC Message Bag*.
2) It is responsible for handling the external events from the environment that are known prior to the start of the simulation and are scheduled by the modeler using a text file, namely *EV file*. These external events are loaded into the NCs during the bootstrap operations by the *main administrator*. Each NC uses a structure called *Event List* to hold those external events it needs to handle during the simulation. Events in the structure are sorted so that they can be processed in increasing timestamp order. The NC uses a

pointer called *event-pointer* to reference the first event that has not yet been sent out. Initially, this pointer points to the first event in the list.

3) It synchronizes the activities of all local processors and drives the simulation on the hosting LP. The local simulation time is advanced by the NC based on three factors: the external events in its Event List, the external messages received in its NC Message Bag, and the closest state transition time provided by the FC.

4) It manages the flow of control messages for the local Simulators in line with the Parallel DEVS formalism. For example, the formalism requires that the output operation must take place just before the state transition in imminent Simulators. Hence, the NC must ensure that the collect message, which triggers the output operation, will be received by imminent Simulators *before* the internal message, which results in the state transition. The correct sequence of these control messages is manipulated using a flag, namely *next-message-type*, which is defined in the state of the NC. It may have a value of collect (@) or internal (*), corresponding to the type of the control message that will be sent out by the NC in the next simulation cycle. The initial value of the flag is set to @.

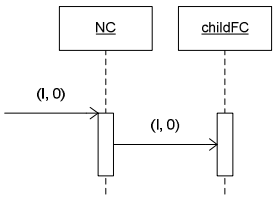


Figure 36. NC algorithm for (*I*, *0*)

Upon receiving (*I*, 0), the NC simply forwards it to the child FC.

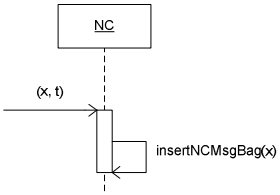


Figure 37. NC algorithm for (*x*, *t*)

In case of receiving (*x, t*), NC will insert this message into the *NC Message Bag*. These external messages contain values sent from remote Simulators to local ones.

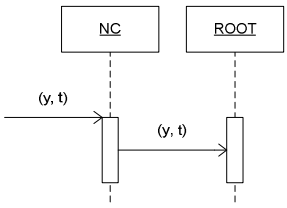


Figure 38. NC algorithm for (*y, t*)

When (*y, t*) is received the NC simply forward it the Root (it has to be sent to the environment).

Finally, reception of a (*D, t*) message by the NC from a child FC indicates that this is a response to a control message that was previously sent out by the NC.

€ Root

This processor only handles environment-oriented output messages during the simulation. Output to the environment is done through a test file called as *output* file or OUT file.

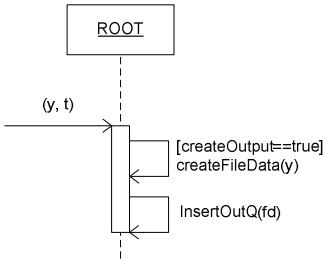


Figure 39. Root algorithm for (*y, t*)

When an output message is received by the Root, it checks to see whether the OUT file is ready. If so, the Root finds out all output ports on the TOP model to which the message will be eventually sent. Then, it creates a *FileData* object from the output message for each of these ports. These data objects are inserted into the file queue corresponding to the OUT file. Finally, the data in the file queue will be flushed out to the physical file by the kernel when GVT advances.

CHAPTER 5 CELL-DEVS MODELS IN THE CD++ TOOLKIT

The CD++ toolkit [Wai02, Tro03] is a modeling and simulation toolkit that implements the original and Parallel DEVS and Cell-DEVS formalisms. Detailed discussion about CD++ toolkit was presented in Section 2.3. In this chapter we will present different models implemented in Cell-DEVS on our CD++ toolkit including: Game of Life (demonstrates the famous Conway's Game of Life), Synapsin-Vesicle Reaction at Nerve Terminal (represents the interaction of synapsin with vesicles at nerve terminal), Fire Spread (illustrates fire propagation in a forest), and Ship Evacuation (an emergency ship evacuation scenario).

5.1. GAME OF LIFE

The Game of Life was created by mathematician John Conway in 1970 [Gar70]. It is the best-known example of cellular automata algorithms. The standard Game of Life uses a two-dimensional grid. We will use this simple example to show the basic facilities of CD++ to define model's rules. Cells can be either on (alive) or off (dead). As presented in Figure 40, the neighborhood of a cell consists of eight cells surrounding it.

(-1, -1)	(-1, 0)	(-1, 1)
(0, -1)	(0, 0)	(0, 1)
(1, -1)	(1, 0)	(1, 1)

Figure 40. Game of life cell neighborhood

The key rule is known as "B3/S23": a new cell is born when it has exactly 3 neighbors; an existing cell (alive cell) survives if it has 2 or 3 neighbors. In all other cases the cell dies, either of overcrowding (with more than three live neighbors) or loneliness (with less than two). At each time step all cells update their state simultaneously. We have modeled the Game of Life using CD++, on a 20x20 cell grid (400 cells). The model definition is shown in Figure 41.

```
1.      [top]
2.      components : life

3.      [life]
4.      type : cell
5.      width : 20
6.      height : 20
7.      delay : transport
8.      defaultDelayTime : 100
9.      border : wrapped
10.     neighbors : life(-1,-1) life(-1,0) life(-1,1)
11.     neighbors : life(0,-1)  life(0,0)  life(0,1)
12.     neighbors : life(1,-1)  life(1,0)  life(1,1)
13.     initialvalue : 0
14.     initialrowvalue :  5     00000011110000000000
15.     initialrowvalue :  7     00000100100100000000
16.     initialrowvalue :  8     00000101110100000000
17.     initialrowvalue :  9     00000100100100000000
18.     initialrowvalue : 11     00000011110000000000
19.     localtransition : life-rule

20.     [life-rule]
21.     rule : 1 100 { (0,0) = 1 and trueCount = 5 }
22.     rule : 1 100 { (0,0) = 0 and trueCount = 3 }
23.     rule : 0 100 { t }
```

Figure 41. Game of life model definition in CD++

As shown on the model definition, the "born" rule is defined by line 22, the "survive" rule is defined by line 21, and the "die" rule is defined by line 23. Figure 42 will show the gird values starting at time `00:00:00:000` until the end of the simulation which is at time `00:00:00:300`. Initially the grid is seeded by a number of live cells by setting the value of the cell to '1'. As the rules are evaluated, more cells are born and finally, at some time they will die until no live cell is left.

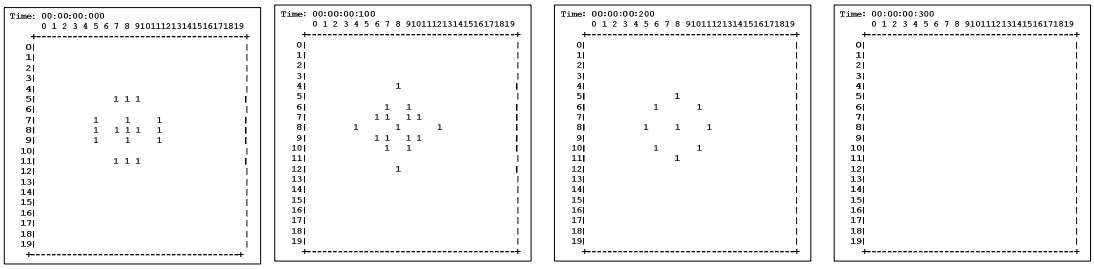


Figure 42. Game of life cell values throughout the simulation

Using CD++, the model has been drawn on a 20x20 cell grid. Figure 43 illustrates the cell gird at four different time stamps of the simulation. The first cell grid shows the initial scenario where seventeen alive cells exist. As the simulation proceeds, either new cells are born or live cells die (based on the "B3/S23" rule). After a while, every live cell

stays alone with no live neighbors. As a result, according to the game rules, the remaining cells die of loneliness.

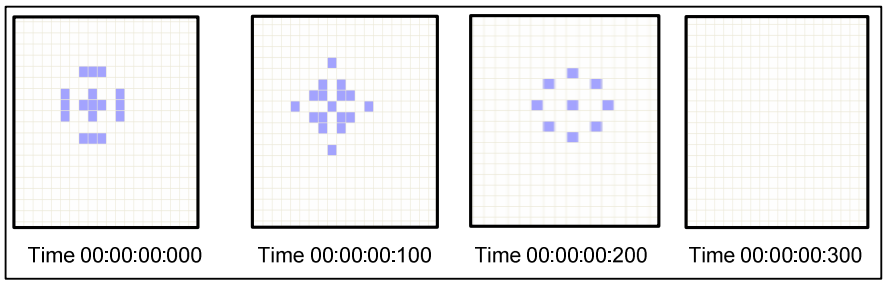


Figure 43. Game of life model at four different time steps throughout the simulation

5.2. SYNAPSIN-VESICLE REACTION AT NERV TERMINAL

We have modeled the reserve pool of synaptic vesicles in a presynaptic nerve terminal, predicting the number of synaptic vesicles released from the reserve pool as a function of time under the influence of action potentials at differing frequencies. Time series amounts for the components are obtained using rule-based methods (the rules defined by Cell-DEVS) [Ala07, Jaf07]. This model was created in collaboration with the Department of Biology at Carleton University. Creating this model in CD++ allows spatial description of synapsin-vesicle interactions. CD++ toolkit makes it possible to have a 2-D graphical representation of this model. As a result a comparable model to the real scene observed in microscopic devices is created.

Synapsin is a neuron-specific phosphoprotein that binds to small synaptic vesicles and actin filaments in a phosphorylation-dependent pattern. Microscopic models have demonstrated that synapsin inhibits neurotransmitter release either by forming a cage around synaptic vesicles (cage model) or by anchoring them to the F-actin cytoskeleton of the nerve terminal [Ben90].

We modeled the molecular interaction of *synapsin* (**S**) with *vesicles* (**V**) which occur inside a nerve cell. The model describes the behavior of synapsin movements until reaching a vesicle and binding to it. Once binding has occurred, depending on *offrate* V and S can again go apart and break their bindings. The *onrate* and *offrate* describe how often bindings occur or break then after. The following formula describes the nature of the reaction:

$$S + V \leftrightarrow SV$$

From the above formula, the left hand side of the equation demonstrates the binding scenario where *synapsin* and *vesicles* perform a bind at a rate specified by *onrate*, while the right hand side of the equation illustrates the bind-break scenario where an *synapsin-vesicle* at an *offrate* which is always smaller than *onrate* breaks apart and again *synapsin* and *vesicles* get released. Then, *synapsin* and *vesicles* can again perform binding and break apart then after. This equation shows an on-going process of "binding" and "breaking apart" which depends on *offrate/onrate*. The larger the *offrate* is, the more bindings get broken apart. Similarly, the larger the *onrate* is, the more V-S binds are produced. Three different scenarios are modeled: 1) V is stationary (with a fixed position on cell space), and S is mobile, 2) V is mobile and S is stationary, and 3) V and S are both mobile (leads to maximum number of total movements and therefore bindings). Binding patterns are in such a way that each S can bind to more than one V, and V can bind to more than one S. Examples of such binding would be:

```
S-V-S-V
|     |
V-S-V-S-V-S-V
```

Each cell space in Cell-DEVS is used to represent one S or V. The neighboring pattern of V and S is in such a way that they can be adjacent cells or diagonal cells, as shown in the following Figure (Gray – Red Cell = S, Black - BlueCell = V).

Diagonal Neighbors **Adjacent Neighbors**

Figure 44. Neighborhood definition

The model uses 100 V and 100 S molecules in a 26x22 cell space. Mobile S or V change position to up, down, left, and right at random. The coupled Cell-DEVS model for this application is described as follows.

$$M=<I,X,Y,Xlist,Ylist,\eta, N,\{m,n\}, C, B, Z, select>$$

Xlist=Φ Ylist=Φ η=9 I=<P^X,P^y>,with P^X={Φ},P^y={Φ};

N={ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1) };

X={0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44};

Y={0,1,2,11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44};

m=26; n=22; B={Φ}; C={Cij/iε[1,26], jε[1,22]}

select ={ (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1) };

Z:

$P_{ij}\ ^{Y}1 \rightarrow P_{i,j-1}\ ^{X}1$ $\quad$ $P_{i,j+1}\ ^{Y}1 \rightarrow P_{ij}\ ^{X}1$

$P_{ij}\ ^{Y}2 \rightarrow P_{i+1,j}\ ^{X}2$ $\quad$ $P_{i-1,j}\ ^{Y}2 \rightarrow P_{ij}\ ^{X}2$

$P_{ij}\ ^{Y}3 \rightarrow P_{i,j+1}\ ^{X}3$ $\quad$ $P_{i,j-1}\ ^{Y}3 \rightarrow P_{ij}\ ^{X}3$

$P_{ij}\ ^{Y}4 \rightarrow P_{i-1,j}\ ^{X}4$ $\quad$ $P_{i+1,j}\ ^{Y}4 \rightarrow P_{ij}\ ^{X}4$

$P_{ij}\ ^{Y}5 \rightarrow P_{ij}\ ^{X}5$ $\quad$ $P_{ij}\ ^{Y}5 \rightarrow P_{ij}\ ^{X}5$

On the cell space, the value 1 is used to represent V, and the value 2 is used to represent S. The number 0 represents an empty cell by which a mobile S can occupy. To give direction to the V (although the model assumes fixed V) or S, a two digit number was used. For example, the following represent:

11	"up"	moving V	21	"up"	moving S
12	"right"	moving V	22	"right"	moving S
13	"down"	moving V	23	"down"	moving S
14	"left"	moving V	24	"left"	moving S

Figure 45. Determining the direction

As mentioned earlier, the model constructed can be further extended to include the movement of both synapsin (S) and vesicles (V) as well as defining different off and on rates. Aside from V-S reactions, the model can also include *Actins*, which bind to *synapsins*. Actins can be represented as a string of cells being fixed at their cell space position. An extract of the model's definition in CD++ is shown in Figure 46:

```
[top]
components : chemCell
[chemCell]
type : cell
dim : (26,22)
delay : transport
defaultDelayTime  : 100
border : wrapped
neighbors :  chemCell(-1,-1) chemCell(-1,0) chemCell(-1,1)
neighbors :  chemCell(0,-1)  chemCell(0,0)  chemCell(0,1)
neighbors :  chemCell(1,-1)  chemCell(1,0)  chemCell(1,1)
initialvalue : 0
initialrowvalue : 0       00102012022010120201 00
initialrowvalue : 1       00010201201010201201 00
...
initialrowvalue : 24      00102012022011012012 00
initialrowvalue : 25      00012020201112022010 00
localtransition : chemCell-rule
[chemCell-rule]
rule : {round(uniform(11,14))} 100 { (0,0) = 1  }
rule : {round(uniform(21,24))} 100 { (0,0) = 2  }
rule : {round(uniform(31,34))} 100
  {((0,0)=21 or (0,0)=22 or (0,0)=23 or (0,0)=24) and
  (((-1,0)- 10 = 1 or (-1,0)- 10 = 2 or (-1,0)- 10 = 3 or (-1,0)- 10 = 4) or
  ((1,0)- 10   = 1 or (1,0)- 10  = 2 or (1,0)- 10  = 3 or (1,0)- 10  = 4) or
  ((0,-1)- 10  = 1 or (0,-1)- 10 = 2 or (0,-1)- 10 = 3 or (0,-1)- 10 = 4) or
  ((0,1)- 10   = 1 or (0,1)- 10  = 2 or (0,1)- 10  = 3 or (0,1)- 10  = 4) or
  ((-1,1)- 10  = 1 or (-1,1)- 10 = 2 or (-1,1)- 10 = 3 or (-1,1)- 10 = 4) or
  ((1,-1)- 10  = 1 or (1,-1)- 10 = 2 or (1,-1)- 10 = 3 or (1,-1)- 10 = 4) or
  ((1,1)- 10   = 1 or (1,1)- 10  = 2 or (1,1)- 10  = 3 or (1,1)- 10  = 4) or
  ((-1,-1)- 10 = 1 or (-1,-1)- 10= 2 or (-1,-1)- 10= 3 or (-1,-1)- 10 = 4))
  and random > 0.10}
...
%moving up
rule : 91 100 {(0,0)=21 and (-1,0)=0 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and (1,0)=91 }
rule : 00 0 {(0,0)=91}
...
%release 0.1 of the S (the offrate is 0.1)
rule : {round(uniform(21,24))} 100 {((0,0)=33 or (0,0)=32 or
                    (0,0)=31 or (0,0)=34) and random < 0.10}
% any others
rule : { (0, 0) } 100 { t}
```

Figure 46. Definition of Synapsin-Vesicle Reaction model in CD++

The detailed explanation of each part of the model rules are defined as follows:

```
rule : {round(uniform(11,14))} 100 { (0,0) = 1  }
rule : {round(uniform(21,24))} 100 { (0,0) = 2 }
```

In the above two rules, the cells are first initialized with 11-14 (for Vesicles) and 21-24 (for Synapsin) to show the scenario at time = 0, where bindings have not yet been performed. Once bindings occur, cells change their values; 11-14 get replaced with 31-34, and 21-24 get replaced with 41-44. Also for Synapsins, four intermediate values 91-94 are used to represent a moving cell that has not yet being settled down. Once it settles down its value changes back to 21-24 (depending on its direction of movement) and gets ready to bind to a vesicle in its neighborhood.

```
rule : {round(uniform(31,34))} 100
  {((0,0)=21 or (0,0)=22 or (0,0)=23 or (0,0)=24) and
  (((-1,0)- 10 = 1 or (-1,0)- 10 = 2 or (-1,0)- 10 = 3 or (-1,0)- 10 = 4) or
  ((1,0)- 10   = 1 or (1,0)- 10  = 2 or (1,0)- 10  = 3 or (1,0)- 10  = 4) or
  ((0,-1)- 10  = 1 or (0,-1)- 10 = 2 or (0,-1)- 10 = 3 or (0,-1)- 10 = 4) or
  ((0,1)- 10   = 1 or (0,1)- 10  = 2 or (0,1)- 10  = 3 or (0,1)- 10  = 4) or
  ((-1,1)- 10  = 1 or (-1,1)- 10 = 2 or (-1,1)- 10 = 3 or (-1,1)- 10 = 4) or
  ((1,-1)- 10  = 1 or (1,-1)- 10 = 2 or (1,-1)- 10 = 3 or (1,-1)- 10 = 4) or
  ((1,1)- 10   = 1 or (1,1)- 10  = 2 or (1,1)- 10  = 3 or (1,1)- 10  = 4) or
  ((-1,-1)- 10 = 1 or (-1,-1)- 10= 2 or (-1,-1)- 10= 3 or (-1,-1)- 10 = 4))
   and random > 0.10}
```

The above rule describes the following scenario: if there exists a synapsin having the value 21, 22, 23, or 24 (a synapsing that can move up/right/down/left) and there is a vesicle in its neighboring which could be an adjacent cell or a diagonal cell, then the synapsin (red cells) will move toward this vesicle and a binding will occur soon, the value of the synapsin gets changed to 31, 32, 33, or 34 (i.e. 21 changes to 31, 22 changes to 32, 23 changes to 33, and 24 changes to 34) to represent a synapsin that is bonded to a vesicle.

```
rule : {round(uniform(41,44))} 100 {((0,0)=11 or (0,0)=12 or (0,0)=13 or (0,0)=14) and
       (((-1,0)- 30 = 1 or (-1,0)- 30 = 2 or (-1,0)- 30 = 3 or (-1,0)- 30 = 4) or
       ((1,0)- 30 = 1   or (1,0)- 30 = 2  or (1,0)- 30 = 3  or (1,0)- 30 = 4)  or
       ((0,-1)- 30 = 1  or (0,-1)- 30 = 2 or (0,-1)- 30 = 3 or (0,-1)- 30 = 4) or
       ((0,1)- 30 = 1   or (0,1)- 30 = 2  or (0,1)- 30 = 3  or (0,1)- 30 = 4 ) or
       ((-1,1)- 30 = 1  or (-1,1)- 30 = 2 or (-1,1)- 30 = 3 or (-1,1)- 30 = 4) or
       ((1,-1)- 30 = 1  or (1,-1)- 30 = 2 or (1,-1)- 30 = 3 or (1,-1)- 30 = 4) or
       ((1,1)- 30 = 1   or (1,1)- 30 = 2  or (1,1)- 30 = 3  or (1,1)- 30 = 4)  or
       ((-1,-1)- 30 =1  or (-1,-1)-30=2   or (-1,-1)-30=3   or (-1,-1)-30 = 4))and
        random > 0.10}
```

Similarly, the above rule describes the following: if there exists a vesicle having the value 11, 12, 13, or 14 (a vesicle that can move up/right/down/left) and there is a synapsin in its neighboring which could be an adjacent cell or a diagonal cell, then since the synapsin will come toward this vesicle and a binding will occur soon, the value of the vesicle gets changed to 41, 42, 43, or 44 (i.e. 11 changes to 41, 12 changes to 42, 13 changes to 43, and 14 changes to 44).

For the movement of synapsin the following four rules are implemented: (each movement is performed in three steps)

```
%moving up
rule : 91 100 {(0,0)=21 and (-1,0)=0 and t}
rule : {round(uniform(21,24))} 0 {(0,0)=0 and 1,0)=91 }
rule : 00 0 {(0,0)=91}
```

step 1: checking to see if there is an empty cell so the synapsin can move into it, for example if the synapsin's direction is upward (value = 21), then at first we need to check if there is an empty cell right above it. (91 is used as an intermediate value to occupy the empty cell)

step 2: once an empty cell is found, it gets occupied by the synapsin (i.e. the cell's value changes from 0 to a random number 21-24).

step 3: the previous position of the synapsin that just moved to an empty cell gets cleared by setting the value of the cell to 0.

Same procedure is used for right, left, and down movement.

```
%release 0.1 of the S (the offrate is 0.1)
rule : {round(uniform(21,24))} 100 {((0,0)=33 or (0,0)=32 or (0,0)=31 or (0,0)=34) and random < 0.10}
```

The above rule is used to break the S-V bindings using an offrate = 0.10. According to this rule, 10% of the bindings get broken and as a result synapsins get released and will be given another direction and they will move around until finding a vesicle and binding to it.

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
  +-------------------------------------------------------------------+
 0|       13    24    14 24    23 23    11    13 21    22    12       |
 1|          12    23    13 22    14    12    22    12 23    13       |
 2|                                                                   |
 3|       12    12 11 23    14    13 21    22 23    22 23    11       |
 4|          23    11          14 11 22    22 23    13 14 14 23       |
 5|                                                                   |
 6|       12 22    12 23    23    13 22    23    13 13 11 22         |
 7|          13 21    24 23    12    12 22    14 24    13 22         |
 8|       12    22    13 22    23 23    11 13    14 22    13 24      |
 9|                                                                   |
10|          12    13    12    13 23    23    11 23    24    11      |
11|          12 22    24    22    12 13 13 22    24 22    13         |
12|       13    22    13 24    12 14 24    24    12    22    12      |
13|       24    13    24    12 13    22    12 24    12 21    14      |
14|                                                                   |
15|       13 22    21 22    12    14    12    13 24    23    13      |
16|       12 23 11 23 21    22    13 21    14    21    12    13      |
17|          12    13    12    12 22    21    12 21    22    12      |
18|                                                                   |
19|       14    22    13    22    14    11 23    13 24    13         |
20|                                                                   |
21|       14 23    23 24    11 23    22    23    13    12    13      |
22|                      21 12                      24               |
23|                                         22 13                     |
24|       12    24    13 22    21 21    12 14    12 22    12 23      |
25|          12 21    22    23    12 14 12 21    22 23    14         |
  +-------------------------------------------------------------------+
```

Figure 47 shows the grid at the initial case where S and V have not yet interacted to perform a bound (bold boxes represent examples of binding structures). Then, Figure 48 will show how bounds are formed and the corresponding cells change their values to represent the binding.

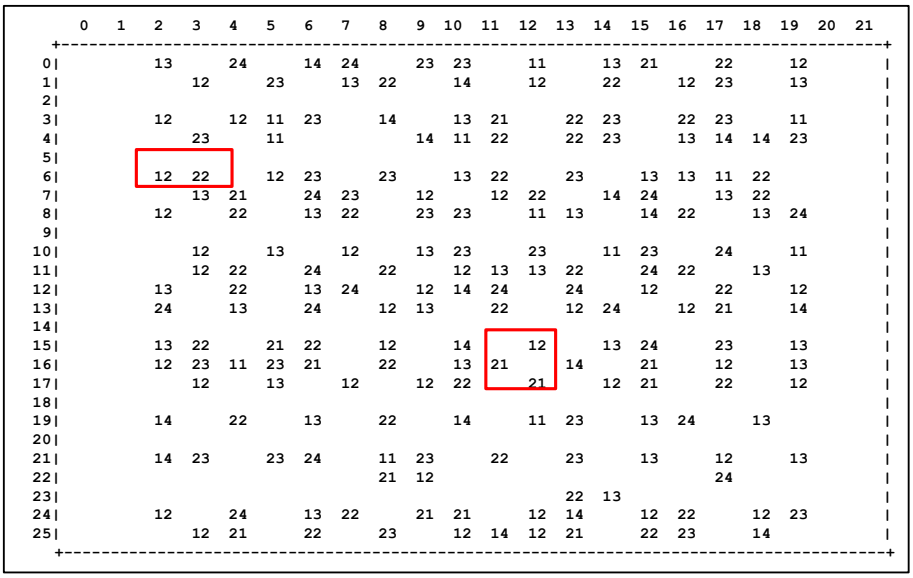


Figure 47. V and S before binding at Time: 00:00:00:100

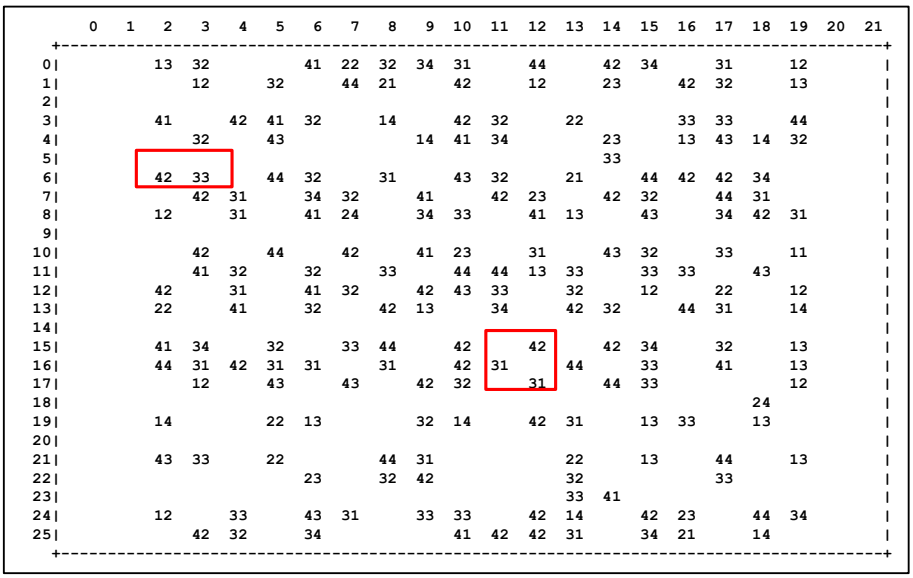
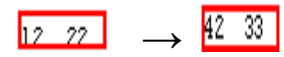


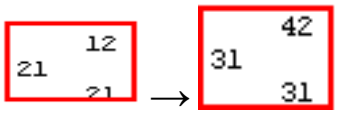
Figure 48. V and S after binding at Time: 00:00:00:300

As illustrated on the above figures, the bold boxes show bindings between synapsin (31-34) and vesicle (41-44). The first illustration (Figure 47) represents the initial scenario where synapsins (21-24) and vesicles (11-14) are free and have not yet performed bindings. Once synapsins walk toward vesicles, the values of the corresponding cells change to 31-34 (bonded synapsins) and 41-44 (bonded vesicles). It is shown that vesicles can be surrounded by more than one synapsin, but each synapsin can bind to only one vesicle at any time.

From the above figure we can see the following possible binding scenarios:

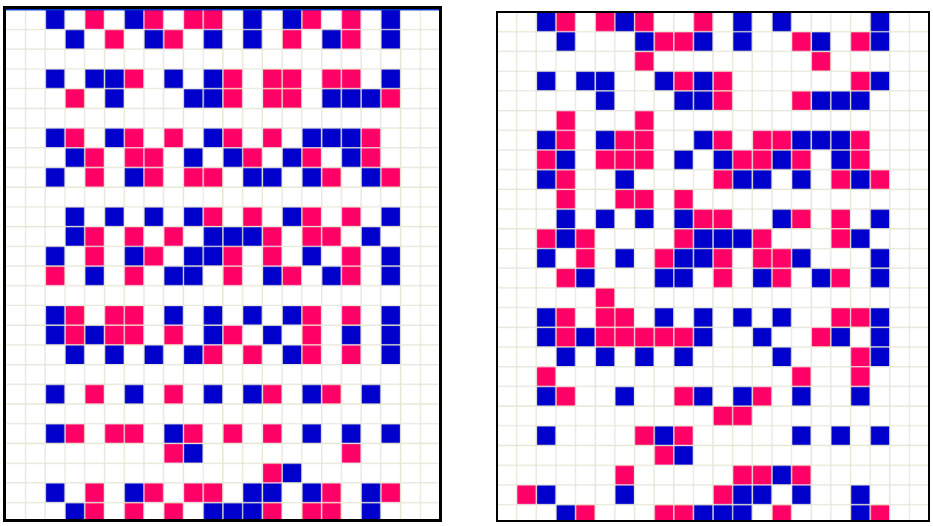


corresponds to: V– S



corresponds to: S – V
|
S

Several initial parameters are tested in order to see the running process of cell nerve with different *offrate*. The case presented in the following figure shows an *offrate* of 0.1.



(a) **(b)**

Figure 49. Model Execution Results: (a) initial values; (b) final execution

The final execution results on Figure 49 present a stable image of synapsin-vesicles bindings where single/double/multiple bindings had occurred within the neuron.

5.3. FIRE SPREAD MODEL

In this section we present Fire Spreading Model introduced in [Ame01], which represents a fire propagation scenario in forest based on Rothermel's mathematical definition [Rot72]. The model computes the ratio of spread and intensity of fire in forest based on specific environmental and vegetation conditions. Three parameter groups determine the fire spread ratio: 1) vegetation type (caloric content, mineral content and density); 2) fuel properties; 3) environmental parameters (wind speed, humidity, and field slope). Figure 50 shows the definition of the model in CD++ using environmental values obtained for a fuel model group number 9, a SE wind of 24.135 km/h and a cell size of 15.24×15.24 m.

```
[top]
components : forestfire

[forestfire]
type : cell          dim : (30,30)          delay : inertial          border : nowrapped
neighbors : (-1,-1) (-1,0) (-1,1) (0,-1) (0,0) (0,1) (1,-1) (1,0) (1,1)
localtransition : FireBehavior

[FireBehavior]
rule : {(1,-1)+(21.552615/17.967136)} {(21.552615/17.967136)*60000} {(0,0)=0 and (1,-1)!=? and 0<(1,-1)}
rule : {(1,0)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and (1,0)!=? and 0<(1,0)}
rule : {(0,-1)+(15.24/5.106976)} {(15.24/5.106976)*60000} {(0,0)=0 and (0,-1)!=? and 0<(0,-1)}
rule : {(-1,-1)+(21.552615/1.872060)} {(21.552615/1.872060)*60000} {(0,0)=0 and (-1,-1)!=? and 0<(-1,-1)}
rule : {(1,1)+(21.552615/1.872060)} {(21.552615/1.872060)*60000} {(0,0)=0 and (1,1)!=? and 0<(1,1)}
rule : {(-1,0)+(15.24/1.146091)} {(15.24/1.146091)*60000} {(0,0)=0 and (-1,0)!=? and 0<(-1,0)}
rule : {(0,1)+(15.24/1.146091)} {(15.24/1.146091)*60000} {(0,0)=0 and (0,1)!=? and 0<(0,1)}
rule : {(-1,1)+(21.552615/0.987474)} {(21.552615/0.987474)*60000} {(0,0)=0 and (-1,1)!=? and 0<(-1,1)}
rule : {(0,0)} 0 { t }
```

Figure 50. Definition of the fire propagation model in CD++

The model consists of 900 cells arranged in a 30×30 mesh, where each cell has the following neighborhood pattern.

(-1, -1)	(-1, 0)	(-1, 1)
(0, -1)	(0, 0)	(0, 1)
(1, -1)	(1, 0)	(1, 1)

Figure 51. Fire cell neighborhood

As mentioned previously, the precondition, delay time, and postcondition rules are based on the mathematical models defined by Rothermel. The fire starts from one cell and propagates throughout the cell grid. Initially, all cells except one are given the value '0' to indicate absence of fire. As the simulation time advances, rules get evaluated to true and fire appears in cells by changing their value to a non-zero number. The fire

sparks from a predefined cell initialized to ‘1’. The initial scenario of the grid at time 00:00:00:000 is presented in Figure 52. The final scenario (Figure 53) shows the fire spread at time 01:59:40:578 where the fire has propagated in the direction of wind.

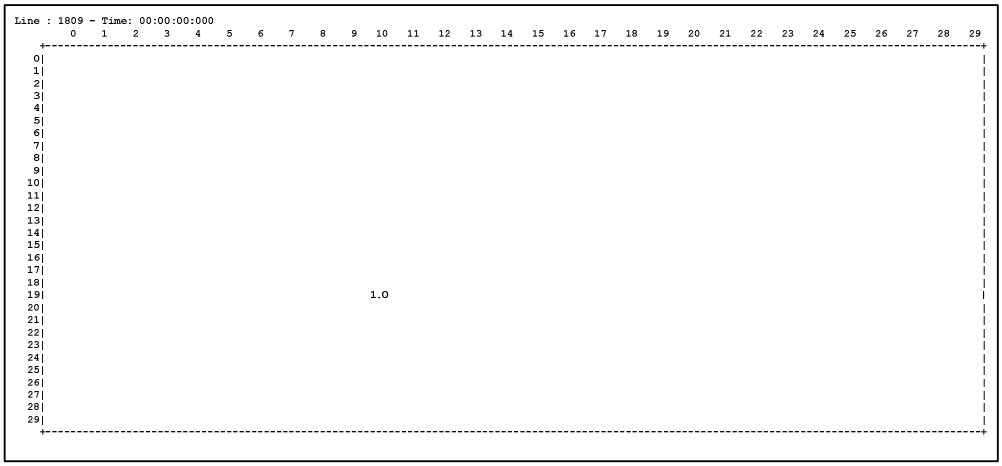


Figure 52. Fire propagation model - initial scenario at time `00:00:00:000`

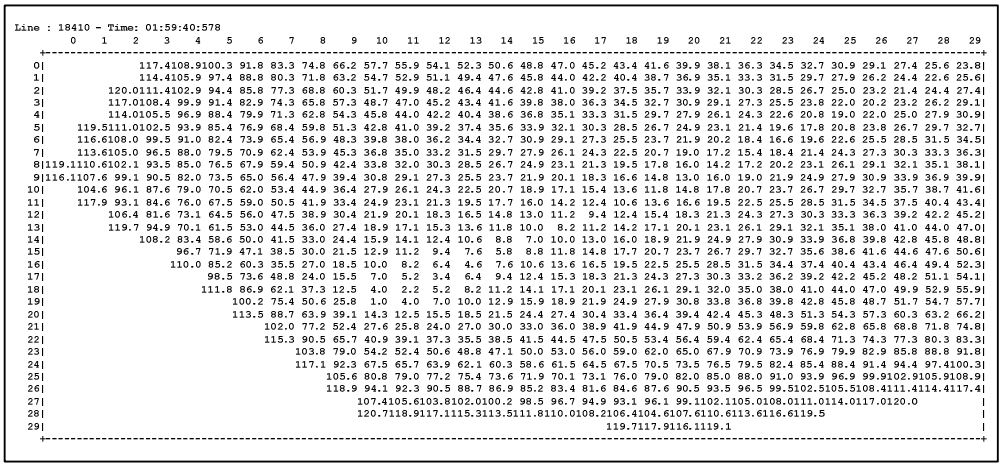


Figure 53. Fire propagation model - final scenario at time `01:59:40:578`

Using CD++, the model has been drawn on a 30x30 cell grid. Figure 54 illustrates snapshots of the simulation results at four different times. Initially, fire starts as fire spot (the dark cell on the grid). Then as time passes by, fire spreads to the neighboring cells in the direction of wind. Therefore, each cell, depending on its position and heat, fires its surrounding cells. As presented on the final scenario of Figure 54, the wind direction leads the fire from the starting point, cell (19, 10), towards southeast of the forest.

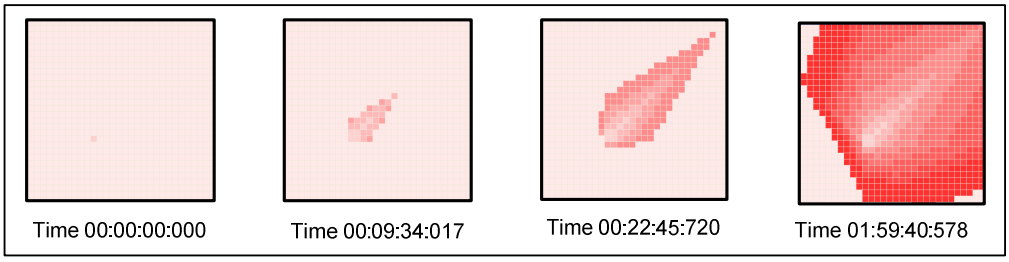


Figure 54. Fire propagation at four different snapshots throughout the simulation

5.4. SHIP EVACUATION MODEL

Based on the model defined in [Klu01], the model we present is the illustration of an emergency ship evacuation scenario. The model consists of 20×20 cell space, using rules with the following restrictions:

1. Each cell representing a person on the ship, calculates its shortest path toward the exit. During the initialization phase, people are placed randomly in any empty cell to imitate real ship evacuation scenario.
2. People run in their initial direction until they encounter another person or an obstacle (e.g. wall).

The neighborhood of each cell consists of 10 cells which will affect the cell's movement (i.e. they can be walls, exit doors, people, or empty cells) as shown in Figure 55.

	UU (-2,0)		
UL (-1, -1)	U (-1,0)	UR (-1, 1)	
L (0, -1)	(0,0)	R (0, 1)	RR (0, 2)
DL (1, -1)	D (1,0)	DR (1, 1)	

Figure 55. Cell neighborhood

From the above figure we can see that the neighborhood consists of 11 cells. Each value on the cell space defines a distinct state, such as the type of the cell: wall, empty, exit door, a moving person. Also each type of movement is given a state value in order to identify the next position of the person. Table 1 summarizes these values.

State Name	Value	Comments
N/A	0	Unknown Empty cell.
Wall	1	Represents an obstacle or a wall.
Exit	2	Represents an exit (e.g. stairs, door).
ED	3	Empty cell and its down (D) cell is the shortest path to the nearest exit.
ER	5	Empty cell and its right (R) cell is the shortest path to the nearest exit.
EU	7	Empty cell and its up (U) cell is the shortest path to the nearest exit.
EL	9	Empty cell and its left (L) cell is the shortest path to the nearest exit.
FD	4	A Full cell (cell with person) and its down (D) cell is the shortest path to the nearest exit.
FR	6	A Full cell (cell with person) and its right (R) cell is the shortest path to the nearest exit.
FU	8	A Full cell (cell with person) and its up (U) cell is the shortest path to the nearest exit.
FL	10	A Full cell (cell with person) and its left (L) cell is the shortest path to the nearest exit.

Table 1. State values and their description

Figure 56 shows an extract of the model's definition in CD++.

```
[top]
components : ship
[ship]
type : cell
width : 20
height : 20
delay : transport
defaultDelayTime : 20
border : nowrapped
neighbors : ship(-2,0)
neighbors : ship(-1,-1) ship(-1,0) ship(-1,1)
neighbors : ship(0,-1)  ship(0,0) ship(0,1) ship(0,2)
neighbors : ship(1,-1)  ship(1,0) ship(1,1)
initialvalue : 0
initialrowvalue :  0     11111111111111111111
initialrowvalue :  1     10000000000000000001
...
initialrowvalue :  18    10000000000000000001
initialrowvalue :  19    11111111111111111111
localtransition : ship-rule
[ship-rule]
%init rules
rule : {3 + randInt(1)} 0 { (0,0) = 0 and (1,0) > 1 and (1,0) < 11}
rule : {5 + randInt(1)} 0 { (0,0) = 0 and (0,1) > 1 and (0,1) < 11}
rule : {7 + randInt(1)} 0 { (0,0) = 0 and (-1,0) > 1 and (-1,0) < 11}
rule : {9 + randInt(1)} 0 { (0,0) = 0 and (0,-1) > 1 and (0,-1) < 11}
%walking rules
rule : 4  100 { (0,0) = 3 and ( (0,1) = 10 or (-1,0) = 4 or (0,-1) = 6 )}
rule : 6  100 { (0,0) = 5 and ( (1,0) = 8 or (-1,0) = 4 or (0,-1) = 6 )}
rule : 8  100 { (0,0) = 7 and ( (1,0) = 8 or (0,1) = 10 or (0,-1) = 6 )}
rule : 10 100 { (0,0) = 9 and ( (1,0) = 8 or (0,1) = 10 or (-1,0) = 4 )}
%exit rules
rule : 3 100 { (0,0) = 4 and (1,0) = 2}
rule : 9 100 { (0,0) = 10 and (0,-1) = 2}
rule : 7 100 { (0,0) = 8 and (-1,0) = 2}
rule : 5 100 { (0,0) = 6 and (0,1) = 2}
%changing direction
rule : 3  100 { (0,0) = 4 and odd((1,0)) }
rule : 9  100 { (0,0) = 10 and odd((0,-1)) and (-1,-1) != 4 }
rule : 7  100 { (0,0) = 8 and odd((-1,0)) and (-2,0) != 4 and (-1,1) != 10 }

rule : 5  100 { (0,0) = 6 and odd((0,1)) and (-1,1) != 4 and (0,2) != 10 and (1,1) != 8 }
```

Figure 56. Definition of ship evacuation model in CD++

The first four rules initialize the model by calculating the shortest path for each undefined cell and placing people on the cell space randomly. The algorithm works as follows: when a cell detects that one of its attached cells has changed its state to "defined", it would know that the attached cell is the shortest path.

Result	Precondition
3 or 4 → ED or FD state	(0,0) = Undefined and (1,0) is defined.
5 or 6 → ER or FR state	(0,0) = Undefined and (0,1) is defined.
7 or 8 → EU or FU state	(0,0) = Undefined and (-1,0) is defined.
9 or 10 → EL or FL state	(0,0) = Undefined and (0, -1) is defined.

Table 2. Initialization rules

The above four rules are implemented in the "init rules" section in Figure 56. The first initialization rule indicates that if the current cell is undefined and the one below it is defined, then the current cell will be randomly changed to an empty or full cell whose down (D) cell is the shortest path to the nearest exit. The second initialization rule indicates that if the current cell is undefined and the one on its right is defined, then the current cell will be randomly changed to an empty or full cell whose right (R) cell is the shortest path to the nearest exit. Similarly, the third initialization rule states that if the current cell is undefined and the one above it is defined, then the current cell will be randomly changed to an empty or full cell whose up (U) cell is the shortest path to the nearest exit. Finally, represented by the fourth init rule, if the current cell is undefined and the one on its left is defined, then the current cell will be randomly changed to an empty or full cell whose left (L) cell is the shortest path to the nearest exit.

Then the second set of rules defines the case when a cell knows that a person will move towards it. The cell knows it will soon be occupied by a person if it is empty and it is the shortest path to at least one cell with a person occupying it.

Result	Precondition
4 → FD state	(0,0) = ED and ((0,1) = FL or (-1,0) = FD or (0,-1) = FR)
6 → FR state	(0,0) = ER and ((1,0) = FU or (-1,0) = FD or (0,-1) = FR)
8 → FU state	(0,0) = EU and ((1,0) = FU or (0,1) = FL or (0,-1) = FR)
10 → FL state	(0,0) = EL and ((1,0) = FU or (0,1) = FL or (-1,0) = FD)

Table 3. Walking rules

The above four rules are implemented in the “walking rules” section in Figure 56. The first walking rule indicates that when the current cell is empty and its down (D) cell is the shortest path to the nearest exit and there is at least one full cell below, or on its right or left, then the current cell is changed to a full cell heading towards down. The second walking rule indicates that if the current cell is empty and its right (R) cell is the shortest path to the nearest exit and there is at least one full cell above, below, or on its right, then the current cell is changed to a full cell heading towards right. Third walking rule represents the case where the current cell is empty and the cell above it is the shortest path to the nearest exit and also there is at least one full cell above, or on its right or left, then the current cell is changed to a full cell heading towards up. Finally, the fourth walking rule states that if the current cell is full and the cell on its left is the shortest path to the nearest exit and also there is at least one full cell above, below, or on its left, then the current cell is changed to a full cell heading towards left.

The third set of rules defines the case when a cell occupied with a person is attached to the exit. Then, the cell knows that a person will leave it and exit.

Resulted State	Input Values
3→ ED state	(0,0) = FD and (1,0) is exit.
5→ ER state	(0,0) = FR and (0,1) is exit.
7→ EU state	(0,0) = FU and (-1,0) is exit.
9→ EL state	(0,0) = FL and (0,-1) is exit.

Table 4. Exit rules

These rules are implemented in the “exit rules” section in Figure 56. The first exit rule indicates the scenario at which the current cell is occupied by a person who is moving downward and the cell below is an exit door, therefore, the person will leave and the current cell’s state changes to an empty cell whose down cell is the shortest path to the exit. The second exit rule indicates that the current cell is occupied by a person who is moving rightward and the cell on the right side is an exit door, therefore, the person will leave and the current cell’s state changes to an empty cell whose right cell is the shortest path to the exit. The third exit rule states that if the current cell is occupied by a person who is moving upward and the cell above it is an exit door, then the person will leave and the current cell’s state changes to an empty cell whose up cell is the shortest

path to the exit. Finally, the last exit rule indicates the scenario at which the current cell is occupied by a person who is moving leftward and the cell on its left is an exit door, therefore, the person will leave and the current cell's state changes to an empty cell whose left cell is the shortest path to the exit.

Then the fourth set of four rules defines when a cell knows that a person will leave it when it is not near an exit. The cell knows that a person will leave it when it is already occupied by a person and its shortest path cell is empty. However, only one person can move to the empty cell when more than one person is trying to move to the same cell. In this case, the priority is first with the person who is in the upper cell, second the one in the right cell, third the one in the down cell, and finally the one in the left cell has the lowest priority.

Result	Precondition
3 → ED state	(0,0) = FD and down (D) cell is empty.
5 → ER state	(0,0) = FR and right cell (R) is empty and UR,RR, and DR cells don't have a person moving to R.
7 → EU state	(0,0) = FU and upper cell (U) is empty and UU and UR cells don't have a person moving to U.
9 → EL state	(0,0) = FL and left cell (L) is empty and UL doesn't have a person moving to L.

Table 5. Changing direction rules

These rules are defined in the "changing direction" section in Figure 56. The first rule indicates that if a cell is occupied with a person whose direction is down (D), then if the cell below the current cell is empty, the person can move down by one cell. The second rule indicates the case where the current cell is full with a person who is wishing to move right, therefore if the cell on the right side of the current cell is empty, the person can move right. The third rule describes the situation where a cell is occupied by a person with upward direction, therefore, if the cell above the current cell is empty the person will move up. Finally, if the current cell is full, being occupied by a person whose direction is leftward, if the cell on the left side of the current cell is empty then the person will move left by one cell.

The ship evacuation model can be modified by adding more exit doors or changing the position of these cells. Using CD++ Modeler [Wai02], the initial cell space at time zero and at the end of simulation when everybody has left through the exit doors

are illustrated in Figure 57. Initially four different types of cells appear on the grid: empty spaces, walls, people, and exit doors, while the final result of the simulation shows no presence of people, i.e. the ship is evacuated.

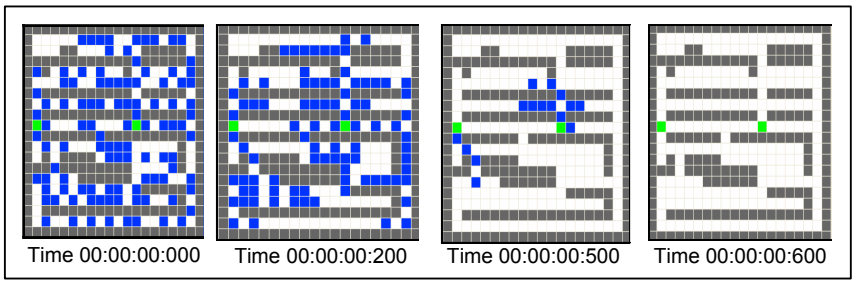


Figure 57. Model Execution Results

CHAPTER 6 THE NEW OPTIMISTIC PCD++ SIMULATOR

In Chapter 4 we introduced our two existing parallel simulators, namely the conservative PCD++ simulator [Tro01] and the optimistic PCD++ simulator [Liu06]. We have modified the WARPED kernel to handle rollbacks in a more efficient way. In this Chapter, we will present two new algorithms that we have implemented in WARPED kernel. Section 6.1 will introduce the rollback mechanism of the optimistic PCD++ simulator. Then, in Section 6.2 the *Near-perfect State Information* protocol will be discussed. Finally, our new algorithms; Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GFRM) will be presented in Section 6.3 and 6.4 respectively.

6.1. ROLLBACK MECHANISM OF OPTIMISTIC PCD++

The WARPED kernel mechanisms are based on a set of standard settings such as: Least-Time-Stamp-First (LTSF) scheduling, copy state saving, passive response GVT (pGVT) algorithm, and fossil cancellation. Using these properties, the scheduling, rollback, and GVT facilities are simplified. The reliable communications over First-in, First-out (FIFO) channels improve Jefferson's definition [Jef85] where he did not assume this order preservation in the communication medium. Furthermore, the kernel uses a predefined ordering scheme for simultaneous events. Thus, events with the same timestamps are ordered based on the identities of their receivers. The input events at the same virtual receive time are ordered based on their *arriving order* (i.e. the sequence by which they are received at), while the output events with the same virtual send time are ordered based on their *sending order* (i.e. the sequence by which they are sent at) [Liu06]. Another restriction is that the timestamp of each event in a process must be less than or equal to the timestamp of the next event in that process [Liu06].

The major internal structures defined in the WARPED kernel include such entities as Logical Process (LP), LTSF scheduler, communication manager, GVT manager, state manager, and simulation objects. Moreover, there are three types of queues used by these

entities: input queue (IQ), output queue (OQ), and state queue (SQ). These structures and the relationship among them are shown in Figure 58.

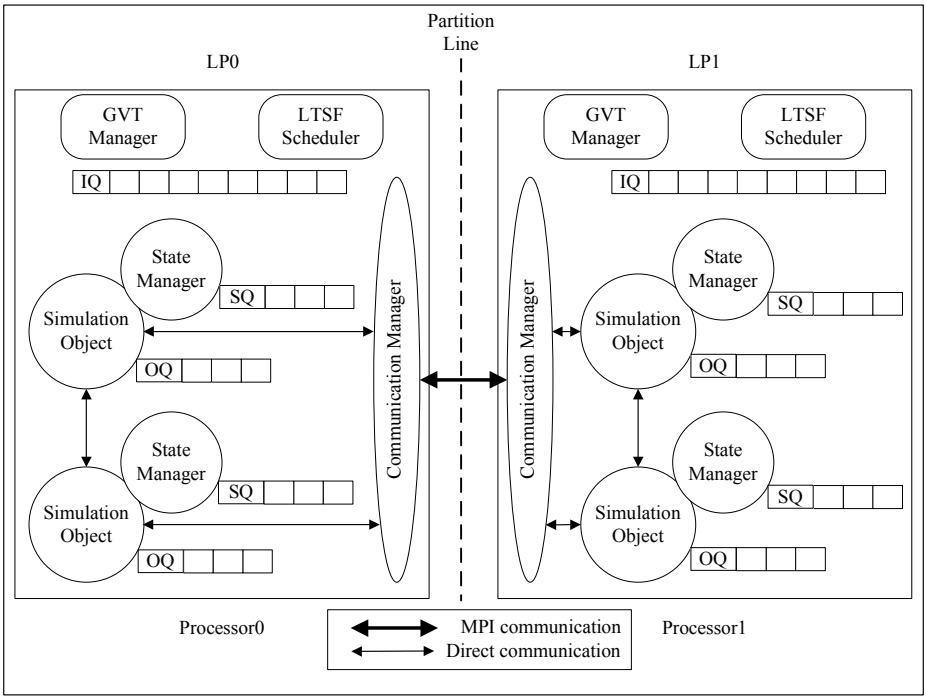


Figure 58. Internal structures in the WARPED kernel

From the diagram we can see that the simulation system (global level) in this example is partitioned into two parts, with each part mapped onto a dedicated processor. All simulation entities on a processor are grouped together by a LP (partition level) that is bound to a physical process. On each processor, the main event processing loop is performed sequentially by the corresponding LP. Processor parallelism occurs at the partition or LP level. Simulation objects (local level) within a LP share a communication manager, a LTSF scheduler, and a GVT manager.

Each simulation object has its own state manager, which manages a state queue on behalf of the object. Each simulation object also has an output queue containing messages the object has recently sent, kept in virtual send time order. All simulation objects within the same LP share a single input queue holding all recently arrived messages sorted in virtual receive time order. Arrangements have been made so that each simulation object seems to have its own logically dedicated input queue. Furthermore, each simulation object may optionally create one or more file queues, which are not shown in the diagram, corresponding to the output files used by the object during the

simulation. The output data are placed temporarily as data objects in these file queues. Physical output activity can only be committed when GVT exceeds the virtual time of these data objects. The simulation objects on all processors must form a complete partitioning of the system. As required by the Time Warp protocol, each simulation object also has a local clock whose value is the current simulation time on that object.

The scheduler selects an event from the input queue in each simulation cycle based on the LTSF algorithm, it then invokes a "process-event" callback provided by the application programmer for the receiving simulation object to execute this event. Maintaining all input events in a single input queue greatly simplifies the scheduling operations as the scheduler has a single access point to all these events.

Inter-LP communications are done by the communication manager over MPI, while intra-LP communications between any two simulation objects and between a simulation object and the communication manager are performed by direct function invocations. Thus when a simulation object on LP0 needs to send an event to another simulation object on LP1, it passes the event to the local communication manager, which then wraps the event into a MPI message and forwards it to the destination LP. On the receiving end, LP1 polls the communication channel regularly and once an incoming MPI message is detected, the communication manager on LP1 receives the message, retrieves the event in it, and delivers the event to the destination simulation object on LP1.

The GVT manager is the entity that operates at the global or system level. Special kernel messages are passed among the GVT managers in the system, implementing a specific GVT calculation algorithm. Each GVT manager performs the fossil collection operations for all simulation objects within the same LP once the GVT goes forward. Implementing rollback is the task of the local control mechanism in Time Warp and the rollback operations are performed by the simulation objects at the local level.

6.1.1. Types of rollbacks

The kernel provides two control mechanisms to handle rollbacks: 1) a rollback mechanism which is implemented at each individual simulation object at the local level, and 2) GVT calculation and fossil collection which is implemented by the LPs at the

global level [Liu06]. Rollbacks are triggered on a simulation object by incoming straggler messages and anti-messages at the time when these messages are inserted into this object's input queue. There are three types of rollbacks that can happen in the kernel: *primary*, *secondary*, and *cascaded* which are discussed in the following points.

€ **Primary rollback**

A typical scenario of the runtime representation of a simulation object before primary rollback is depicted in Figure 59.

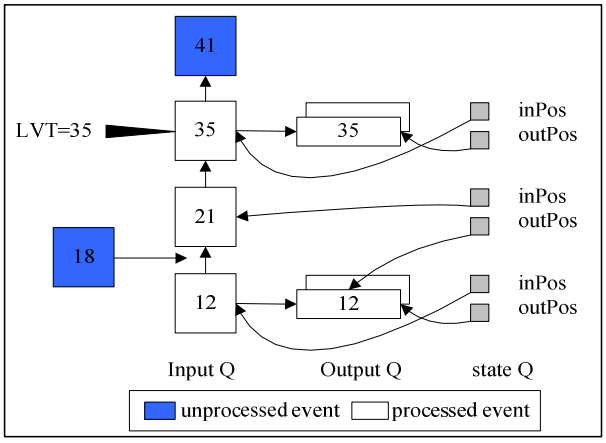


Figure 59. Runtime representation of a simulation object

In the diagram, an input event is depicted as a block with the *receive time* of the event shown on it. On the other hand, an output event is shown as a block with its *send time*. States saved on the state queue are pictured as boxes with three values: the Local Virtual Time (LVT) of the simulation object (as shown in the box), a pointer, *inPos*, pointing to the event just executed, and another pointer, *outPos*, identifying the last message sent by the simulation object. Let's denote the states shown in the diagram as S(12), S(21), and S(35).

The figure shows that this simulation object has executed events with receive time 12, 21, and 35, notated as E(12), E(21), and E(35) respectively. After executing these events, the simulation object's LVT is set to 35, the receive time of the last event executed. The simulation object sent three messages when E(12) was executed, while two messages were sent out when E(35) was executed. These output messages are recorded in the output queue in increasing send time order. Notice that there is no output message generated as the result of executing E(21). Hence, the outPos of S(21) still points to the last output message in the previous cycle, which is also referred by the outPos of S(12).

Now let's consider an event E(18) arriving at this simulation object. When it is inserted into the input queue, the kernel rollback facility finds out that the receive time of this incoming event is strictly less than the current LVT, thus the event is identified as a straggler event, and a primary rollback will be performed on this simulation object. The receive time of the straggler event is referred to as *rollback time* in our following discussion. Here, the rollback time is 18.

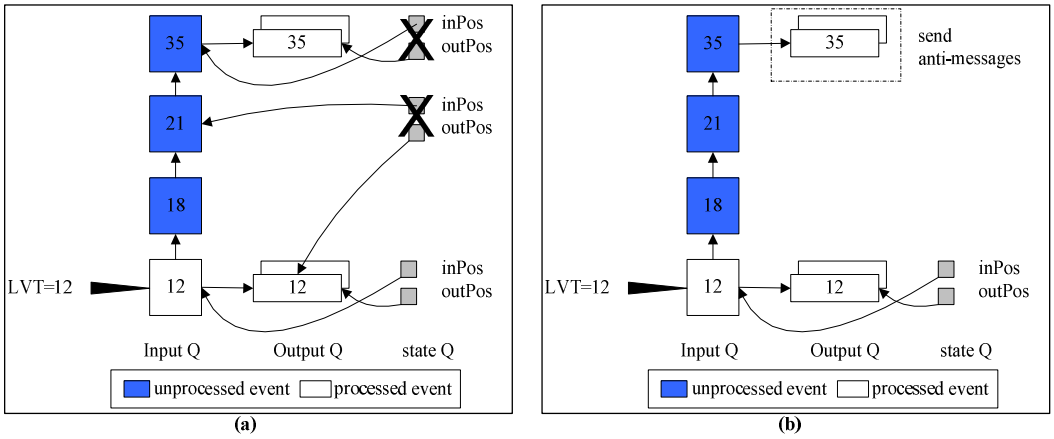


Figure 60. Actions performed during primary rollback

The actions performed by the kernel rollback facility during primary rollback are illustrated in Figure 60 and described as follows:

1) Insert the straggler event E(18) into the input queue.
2) Undo those previously processed events following E(18) in the input queue. As shown in Figure 60(a), E(21) and E(35) are unprocessed.
3) Search the state queue to find the last state saved before the rollback time, and then restore the simulation object's current state based on that state. Therefore, the content of the object's current state is an exact duplicate of S(12). Notice that when copying a state, all data contained in the state are copied, including the values of the inPos and outPos.
4) Discard from the state queue all states after S(12).
5) Reset the object's LVT to the recorded value in the object's current state. Since the object's current state is now a copy of S(12), the LVT is reset to 12.
6) Rollback file queues, if any, associated with this simulation object, which is not shown in the diagram. The simulation object may create output files and the output data are contained in the corresponding file queues in increasing

virtual time order. Rollback of these file queues are simply done by removing all data with virtual time greater or equal to the rollback time.

7) Find the last output message sent during executing E(12) from the output queue. This can be done with ease in constant time since the outPos of the restored state, S(12), is pointing exactly to that message.
8) Send all messages in the output queue after the last output message found in step 5 as anti-messages to their receivers, as shown in Figure 60(b).

After these operations, the kernel resumes normal execution forward again.

We can see from step 6 that anti-messages may be emitted at the end of the primary rollback to remove the effects of incorrect computations on other simulation objects, both locally within the same LP and remotely on other processors in which case the anti-messages are sent as MPI messages through the communication manager.

€ **Secondary rollback**

Secondary rollback on a simulation object is caused by receiving an anti-message. Depending on whether the corresponding positive message is processed or not, there are two kinds of scenarios that could happen here:

The first scenario is when the positive event has already been processed. Thus, the simulation object receives an anti-message tagged with a negative time. For example, the anti-message of E(21) is denoted as E(-21). The actions performed by the kernel for this type of rollbacks are described as follows.

(1) Perform a message annihilation operation to delete both the original message and its anti-message counterpart.
(2) Follow step 2 to step 8 of the primary rollback operations as presented earlier but using the timestamp of the anti-message as the rollback time.

We can see that the operations performed here largely remain the same as those of primary rollbacks except that a message annihilation operation replaces the previous enqueue operation.

In the second scenario, the positive event has not yet been processed by the simulation object. The only action that needs to be done is a message annihilation operation for the anti-message. The simulation object continues to execute the next available event after the annihilation.

As in the case of primary rollback, there is the issue of identifying the type of the straggler event. There has to be dedicated mechanism to deal with the positive events, processed and unprocessed, that have the same timestamp as the anti-message during secondary rollbacks.

€ **Cascaded rollback**

The primary rollback triggered by a straggler message is the root cause of rollbacks in Time Warp. Hence, rollback propagation starts with one primary rollback and, probably, multiple rounds of secondary rollbacks occurring upon the arrival of anti-messages at the destination simulation objects. The hosting simulation object of the primary rollback is thus named as *rollback originator*, and the original primary rollback of the propagation is called the *root of the propagation*. The levels of secondary rollbacks may be to any depth, and there may even be circularity in the graph of anti-message paths, but the propagation eventually terminates [Jef85].

During either type of rollback, there may be zero, one, or more anti-messages sent out from the simulation object. The spreading of the anti-messages may happen with a partition on the same processor or across the border of partitions to other processors. Each of these anti-messages will trigger a further secondary rollback on the destination simulation object. The secondary rollback is performed *immediately* upon the arrival of the anti-message at the destination simulation object. Notice that a simulation object A sends an anti-messages to simulation object B triggering a secondary rollback on B, and then during the secondary rollback, B may send back anti-messages to A causing further secondary rollbacks on A, which forms a circle in the graph of anti-message paths. However, if we consider the propagation process in terms of rollbacks instead of simulation objects, these circles disappear since the further secondary rollbacks on object A are simply a deeper level of rollbacks resulting from the earlier secondary rollback on object B. This suggests using a tree structure to depict the propagation process.

The propagation process can be described as traversing the whole tree from the primary rollback (root of the tree). This operation backtracks, by returning from the rollback function, to the most recent node that did not have further anti-message from that rollback or a node that represents a remote subtree. Under this technique, the time measured for the simulation object where the primary rollback takes place includes not

only the time for the primary rollback itself, but also the time for all other local rollbacks in the tree. This is one example that we should be careful about rollback operations.

6.2. NEAR-PERFECT STATE INFORMATION PROTOCOLS

The Near-perfect state information (NPSI) protocols proposed by Srinivasan [Sri98] are a new class of synchronization for parallel discrete event simulation which outperform Time Warp, both temporally and spatially. NPSI-based protocols dynamically control the rate at which processes exploit parallelism achieving a more efficient parallel simulation. In optimistic protocols such as Time Warp, logical processes execute events *aggressively* assuming freedom from errors. Thus, the aggressive event execution would include *risk* which is the potential at which erroneous results propagate to other LPs [Rey88]. The NPSI protocols aim at controlling both aggressiveness and risk of optimism adaptively by computing an error potential (EP). The EP of a process is defined as a function of the states of other LPs participating in the simulation. It works as an elastic force which sometimes blocks and sometimes frees the progress of the LP.

The optimism implemented by Time Warp protocol incurs three time costs: state saving, rollback, and memory management [Sir98]. Furthermore, by restricting optimism a forth time costs gets introduced; the lost opportunity cost which is defined as the potential loss in performance when an LP is suspended form execution while it was safe for it to continue. Thus, protocols that control optimism define the cost function as follows:

Total cost = state saving cost + *rollback cost + memory management cost + lost opportunity cost*.

Since the time cost of state saving can be a function of the size of the state and the frequency rate at which it is saved, the *Total cost* function can be rewritten by omitting the sate saving cost resulting in:

Total cost = rollback cost + memory management cost + lost opportunity cost.

As mentioned above, by limiting optimism the first two costs can be reduced. However, as a result of this limitation, the lost opportunity cost would increase in return. Figure 61 illustrates this tradeoff.

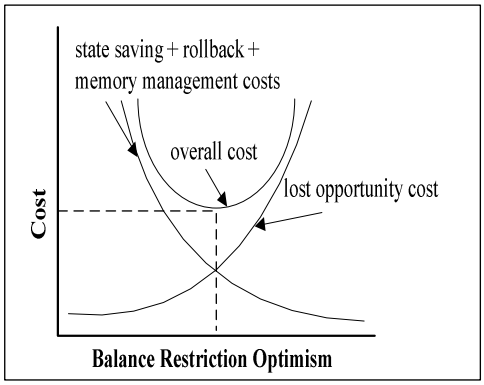


Figure 61. Tradeoff introduced by limiting optimism [Sri95]

From the graph we can see that the best performance is attained when the controlled optimism eliminates both rollback and memory management cost and in return adds zero lost opportunity cost [Sri98]. Optimism limiting protocols can achieve a good balance by precisely identifying the incorrect computations and avoiding their propagation. This can only be done by providing each LP with *perfect* state information about other LPs. The issue is that due to various latencies in computing distributed snapshots, it is impossible to obtain perfect state information. The NPSI mechanism approximates perfect state information by using a dynamic feedback system that operates asynchronously with respect to LPs. Hence, providing LPs with near-perfect state information at low-cost.

The NPSI protocols are optimistic protocols that control the aggressiveness and risk of LPs by dynamically computing near-perfect sate information. Design of such protocols includes two phases:

1. Identifying the state information that is required for controlling optimism; and
2. Designing a mechanism that translates this information into control over an LP's optimism.

As mentioned previously, EP_i is the value which controls the optimism of LP_i. Figure 62 represents a general framework for adaptive protocols.

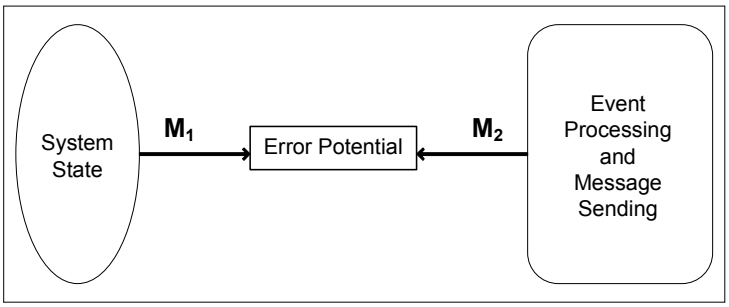


Figure 62. General framework for adaptive protocols [Sri98]

The NPSI protocol keeps each EP_i up-to-date by evaluating the function M_1 using the near-perfect sate information it receives from the feedback system. On the other hand, M_2 function dynamically translates new values of EP_i in the event execution and communication rates.

In order to control the optimism of PCD++, we have modified WARPED [Mar99] to implement a NPSI mechanism based on the number of rollbacks. The idea is to reduce the number of rollbacks by suspending the simulation object within LP that has large number of rollbacks and therefore blocking it from flooding the net with anti-messages. However, the LP will still be able to receive input events and they will be inserted into the corresponding message bags. After a predefined duration, the suspend simulation object is released and will go on simulating. Based on previous research [Szu00], we have implemented two new protocols, namely *Local Rollback Frequency Model* (LRFM) and *Global Rollback Frequency Model* (GRFM) to limit the optimism of PCD++ simulator.

The main concept is to associate each LP with an *error potential* (EP_i) to control the optimism of LP_i. During the simulation run, the value of each EP is kept updated by evaluating *M1* function which uses state information that is received from the feedback system. Then, the function *M2* dynamically translates every update of EP_i in delays in the execution events. The next two sections will provide more details about the design and implementation of LRFM and GRFM.

6.3. LOCAL ROLLBACK FREQUENCY MODEL

The Local Rollback Frequency Model (LRFM) protocol is only based on local information of the logical processes. That is, the simulation object within a LP will be suspended or allowed to continue simulating only based on the number of rollbacks it had. First, M_1 and M_2 functions must be defined:

- **Function M_1:** The error potential of a simulation object is the number of rollbacks that the object had from a time *T1* until the actual time *T2*, having that $T2 - T1 \leq T$, where *T* is the interval after which the local number of rollbacks of the simulation object gets restarted back to zero.
- **Function M_2:** If the number of rollbacks for a simulation object at the interval *T* is greater than a specified value, then the object is suspended, adopting a conservative behavior. By suspending the simulation object, the LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object is again given the permission to resume. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates aggressively, adopting its usual optimistic behavior (as in Time Warp).

To implement this protocol each LP has to be informed about two values: *max_rollback*, and *period*. Where *max_rollback* is the maximum number of allowed rollbacks before suspension of the simulation object, and *period* is the duration for which the simulation object will stay suspended. The algorithm is presented in Figure 63.

```
1. In each LP, at the beginning predefine:
   max_rollbacks and period
2. In each simulation object, at the simulation start:
   previous_time = 0
3. In each object, when the LP is scheduled to run:
   actual_time = Warped.TotalSimulationlTime ()
        if (actual_time - previous_time >= period)
                simulateNextEvent()
                previous_time = actual_time
                rollbacks = 0
        else
                if (rollbacks < max_rollbacks)
                simulateNextEvent()
                /* else, SUSPEND the simulation object */
```

Figure 63. LRFM algorithm

From the LRFM algorithm we see the following three possible scenarios:

1. The LRFM period has expired, therefore the simulation object starts a new period, its number of rollbacks gets reset to zero, and it is given the permission to continue its execution.
2. The LRFM period has not yet expired, if the number of rollbacks of the simulation object is less than the allowable range (i.e. *max_rollbacks*), then the simulation object continues its normal execution.
3. The LRFM period has not yet expired, but the number of rollbacks within the simulation object has exceeded *max_rollbacks*, thus the simulation object gets suspended for the entire duration of the current LRFM period.

With the inclusion of this protocol, in every simulation cycle an object will simulate the lowest timestamp event (as WARPED does originally) if the number of its rollbacks in the period *T* is smaller than the maximum allowable rollbacks; if not, the object suspends executing until the new period of time *T*, after which Warped restarts the rollbacks number to zero.

In order for an LP to be able to simulate objects that mustn't be delayed, we have modified the scheduler policy to choose the next object to simulate. It chooses the first object of the input event list (that is, the object with the lowest input event timestamp)

only if its rollbacks count does not exceed *max_rollbacks*; else, the scheduler checks the next object of the input event list and so on, until it finds an object in condition to be simulated or until it reaches the end of the list.

6.4. GLOBAL ROLLBACK FREQUENCY MODEL

In Global Rollback Frequency Model (GRFM) protocol each simulation object uses global information in such a way that among all the simulation objects residing on all LPs, the one with greatest number of rollbacks must be suspended for the duration of time defined at the beginning of the simulation. Therefore, at each simulation cycle all the LPs must broadcast the information regarding the rollback counts of all of their simulation objects. As in LRFM, M_1 and M_2 functions must first be defined:

- **Function M_1**: The error potential of a simulation object is the number of rollbacks that the object had minus the maximum number of rollbacks of the other simulation objects (both local and remote ones) participating in the simulation, from a time *T1* until the actual time *T2*, having that $T2 - T1 \leq T$, where *T* is the interval after which the local number of rollbacks of the simulation object gets restarted back to zero.
- **Function M_2**: If the number of rollbacks for a simulation object at the interval *T* is greater than other number of rollbacks of the other simulation objects, then the object is suspended, adopting a conservative behavior. By suspending the simulation object, the LP where the object resides on will still be able to receive incoming events, but the events are not processed until the simulation object is again given the permission to resume. However, if the number of rollbacks of the simulation object is less than the predefined value, then the object simulates aggressively, adopting its usual optimistic behavior (as in Time Warp).

This algorithm is implemented as follows:

```
1. In each LP, at the beginning predefine: period
2. In each simulation object, at the beginning predefine:
      previous_time  = 0
      max_rollbacks = 0
3. In each simulation object, when the LP is scheduled to run:
     actual_time = Warped.TotalSimulationlTime ()
            if (actual_time - previous_time >= period)
                  simulateNextEvent()
                  previous_time  = actual_time
                  rollbacks = 0
            else
                  if  (rollbacks < max_rollbacks)
                  simulateNextEvent()
            /* else, SUSPEND the simulation object */
4. For i from 1 until the number of LPs
            if (i is NOT this LP id)
            send to LP i the number of rollbacks of the objects of the LP id
            Subroutine that receives the number of rollbacks from other LP:
            For j from 1 until the numbers received
                  If (rollbacks[j] > max_rollbacks)
                  max_rollbacks = rollbacks[j]
```

Figure 64. GRFM algorithm

As in LRFM, the GRFM algorithm yields three different scenarios:

1. The GRFM period has expired, therefore the simulation object starts a new period, its number of rollbacks gets reset to zero, and it is given the permission to continue its execution.
2. The GRFM period has not yet expired, if the number of rollbacks of the simulation object is less than the allowable range (i.e. *max_rollbacks*), then the simulation object continues its normal execution.
3. The GRFM period has not yet expired, but the number of rollbacks within the simulation object has exceeded *max_rollbacks*, thus the simulation object gets suspended for the entire duration of the current GRFM period.

The main difference of GRFM and LRFM is the way *max_rollbacks* is initialized. In LRFM, maximum allowable rollbacks is predefined by the user at run time, while in GRFM maximum allowable rollbacks is set to the largest number of rollbacks of all

participating simulation objects. That is, whenever a simulation objects is scheduled to execute, it must send the number of rollbacks it had so far to all other simulation objects, both local and remote ones. As a result, at any time *max_rollbacks* is the largest number of rollbacks among all the existing simulation objects.

By implementing LRFM and GRFM protocols in our optimistic PCD++ simulator, different simulation results can be collected since the RFM *period* (and in case of LRFM the *max_rollbacks*) can be modified very easily at the beginning of the simulation. This is done by changing these values in the configuration files right before the simulation starts and therefore, there is no need to rebuild the whole simulator in order for these modifications to have effect. Chapter 7 will discuss the performance of these two algorithms by testing those models which were presented in Chapter 5.

CHAPTER 7 EXPERIMENTS AND PERFORMANCE ANALYSIS

As part of the contribution of this research, we have run a variety of tests to analyze the performance of our existing PCD++ simulators; the optimistic and the conservative as well as our LRFM and GRFM Time Warp-based protocols. The main goal of this section is to show the capability of PCD++ simulators in terms of handling the number of nodes driving the simulation, complexity of the model, and the size of the model. As was mentioned earlier in Chapter 5, we have selected different models with distinguishable functionality, complexity, and size to better judge the capability of the simulators. Our experiments were carried out on a HP PROLIANT DL Server, a cluster of 32 compute nodes (dual 3.2GHz Intel Xeon processors, 1GB PC2100 266MHz DDR RAM) running Linux WS 2.4.21 interconnected through Gigabit Ethernet and communicating over MPICH 1.2.6. A description on how to run Cell-DEVS models is given in Section 7.1. The performance metrics are presented in Section 7.2. Finally, Section 7.3 will present the execution results of the Cell-DEVS models and the improvements achieved by using different simulators/protocols.

7.1. RUNNING CELL-DEVS MODELS

Each Cell-DEVS model consists of a number of necessary and optional files grouped together in a package. Since the simulation can be distributed among 1 to 32 nodes of the cluster, we used a partitioning mechanism implemented earlier in [Tro01, Liu06] which evenly divides the cell space into horizontal rectangles, as illustrated in Figure 65 which represents the partitioning of a 30×30 (900 cells) model over 3 nodes. Different partitioning strategies can be implemented which in return result in a significant impact on the performance of the simulation.

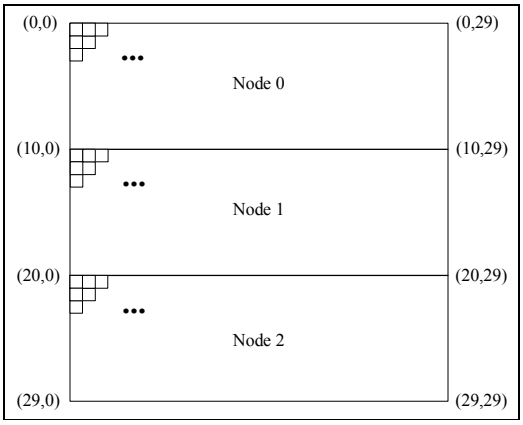


Figure 65. A simple partition strategy for a 30x30 Cell-DEVS model [Liu06]

PCD++ simulators enable user to modify a partition configuration file prior to the simulation by entering the desired number of nodes that the simulation will be carried on. Figure 66 illustrates the content of the partition file of the fire propagation model where the user had selected 3 nodes for running the simulation.

```
0 : forestfire(0,0)..(9,29)
1 : forestfire(10,0)..(19,29)
2 : forestfire(20,0)..(29,29)
```

Figure 66. The partition file of fire propagation model for simulation on 3 nodes

Once the partitioning is performed, the model simulation can start by running the execution script. An extract of the simulation results of the 30x30 fire model execution on two nodes is illustrated in Figure 67. As shown on the figure, the statistical details of each logical process (LP0 and LP1) are presented. The important statistics for each LP include: number of rollbacks (RB), length of rollback (LRB), bootstrap time (BT), running time (RT), events received (ER), events executed (EE), and the number of local objects. Once the simulation is over, the start time, end time, and the total elapsed time is printed out. The value shown as “total elapsed time” is the total simulation time which includes the bootstrap time as well as the running time (i.e. T = BT + RT).

```
[sjafer@node01 fire_30_2M]$ runfire.sh
      Run on ->node02
      Run on ->node01
PCD++: A Tool to Implement n-Dimensional Cell-DEVS models
------------------------------------------------------------
Stop at time: 05:00:00:000
[1] Local objects  : 453
[0] Local objects  : 455
Total objects : 908 / Total machines : 2
Using a period of 15000 to calculate GVT

      -------- Configuration for this simulation run --------
            --> LTSF Scheduling
            --> Saving state information every event
            --> Message type-based state saving
            --> pGVT algorithm for GVT estimation
            --> System's memory manager
            --> Aggressive cancellation strategy
      ------------------------------------------------------------

      -LY is off, create log files for NodeCoordinators only.
Create output file for Root.
GVT [0]: 00:00:00:000
GVT [1]: 00:00:00:000
GVT [1]: 00:37:00:561
GVT [1]: 02:00:28:888
GVT [0]: 02:00:28:888
Simulation complete!
------------------------- Statistics ----------------------------
LP[0] (ER) = 24471 / (EI) = 2954 / (PR) = 25 / (SR) = 459 / (RB) = 484 / (RBL) = 2451
LP[0] localObjs = 455 / (SS) = 12749 / (SK) = 12082 / (SR) = 0 / (EE) = 24376 / (CFL) = 0
LP[0] (LH) = 0 / (LM) = 0
LP[0] (ST) = 7.54898e+08 ns  / (ET) = 2.61378e+09 ns  / (CT) = 0 ns / (RT) = 5.6514e+07 ns
LP[0] (BT) = 8.5956e+08 ns
LP[0] (DT) = 2.86839e+08 ns
(Inter-LP Message Size) = 116 bytes
Simulation ended!
------------------------- Statistics ----------------------------
LP[1] (ER) = 26844 / (EI) = 5487 / (PR) = 34 / (SR) = 953 / (RB) = 987 / (RBL) = 4447
LP[1] localObjs = 453 / (SS) = 14183 / (SK) = 12957 / (SR) = 0 / (EE) = 26687 / (CFL) = 0
LP[1] (LH) = 0 / (LM) = 0
LP[1] (ST) = 7.3481e+08 ns  / (ET) = 3.09227e+09 ns  / (CT) = 0 ns / (RT) = 1.10897e+08 ns
LP[1] (BT) = 7.99848e+08 ns
LP[1] (DT) = 0 ns
------------------------------------------------------------
START_STRING = 48:56:004186000
Start time: min=48, sec=56, neno=4186000
END_STRING = 49:01:602353000
End time  : min=49, sec=1, neno=602353000
Total elapsed time (seconds):5.5981
[sjafer@node01 fire_30_2M]$
```

Figure 67. Execution results of running 30x30 fire model using the optimistic PCD++ simulator

7.2. PERFORMANCE METRICS

The *total elapsed time* value was collected from the execution environment to measure the performance of the simulators in terms of execution time. Also, the speedups with respect to changing the number of simulating nodes were calculated to show how the parallel simulation outperforms the sequential one (using only one node). The *overall speedup* for N nodes is given as follows.

$$\text{Overall Speedup} = \frac{T(1)}{T(N)}$$

Where T (1) represents the serial execution time measured on one node, and T (N) is the total execution time taken by the simulation running on N nodes. Each of the models which were presented in Chapter 5 is executed on four different simulators:

- The optimistic PCD++ simulator [Liu06];
- The conservative PCD++ simulator [Tro01];
- The optimistic PCD++ simulator implementing LRFM protocol; and
- The optimistic PCD++ simulator implementing GRFM protocol.

The goal is to identify the execution performance of each simulator as we increase the number of participating nodes. Due to the partitioning mechanism that is used by our optimistic and conservative simulators, we can only increase the number of nodes to a certain limit. That is, the maximum number of nodes that a model can be simulated on is equal to the number of rows of the cell gird for that particular model. For instance, if we have a model of 400 cells arranged in a 20x20 mesh, we can run the model on 1 to 20 nodes. In order to obtain stable results, for each model, simulations were run on 1 to N nodes and for each scenario five trials were collected. The execution results which will be presented in the next section reflect the average of these five trials which are within a confidence interval of 95%.

7.3. SIMULATION RESULTS

In the following points we will present the simulation results of executing our four models discussed in Chapter 5.

- **Game of Life Model**

This model consists of 1200 cells arranged in a 30x40 mesh with a total execution time of 4.6723 seconds when run on standalone CD++. Figure 68 represents the execution time resulting from running the model with four different simulators on 1 to 6 nodes.

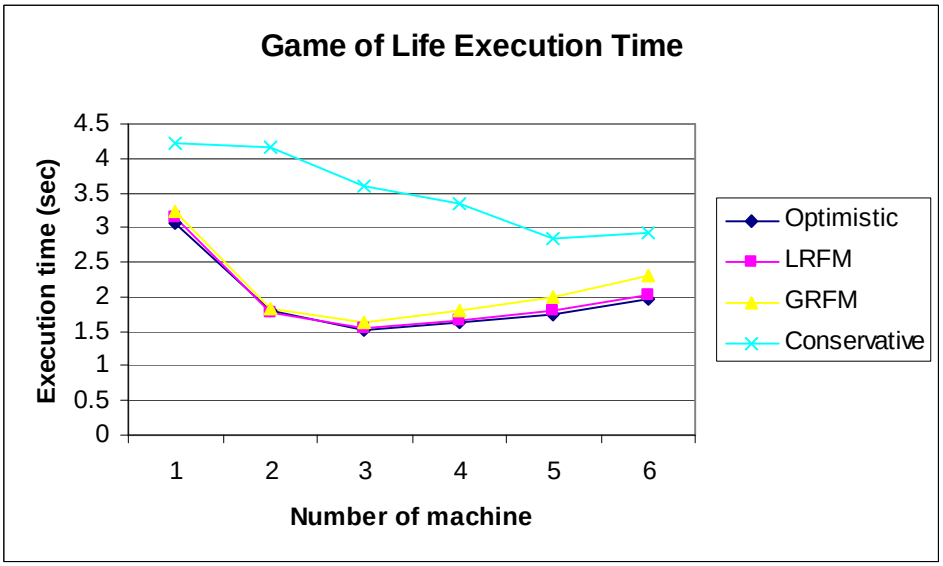


Figure 68. Game of life model execution time on 4 different simulators

From the execution time graph, we can see that the optimistic, LRFM-based, and GRM-based simulators outperform the conservative one on 1 to 6 nodes and at the same time produce very close results. However, as the number of machines goes beyond 3, the conservative simulator starts dropping down the execution time. Among the three optimistic simulators, the GRFM-based simulator takes longer time due to its time consuming mechanism in broadcasting information about each LP's rollbacks among the participating nodes.

The speedups of the model execution times with respect to execution on one node for each particular simulator are given in Figure 69. The speedups graph only represents the performance of 1 to 3 nodes which showed significant performance.

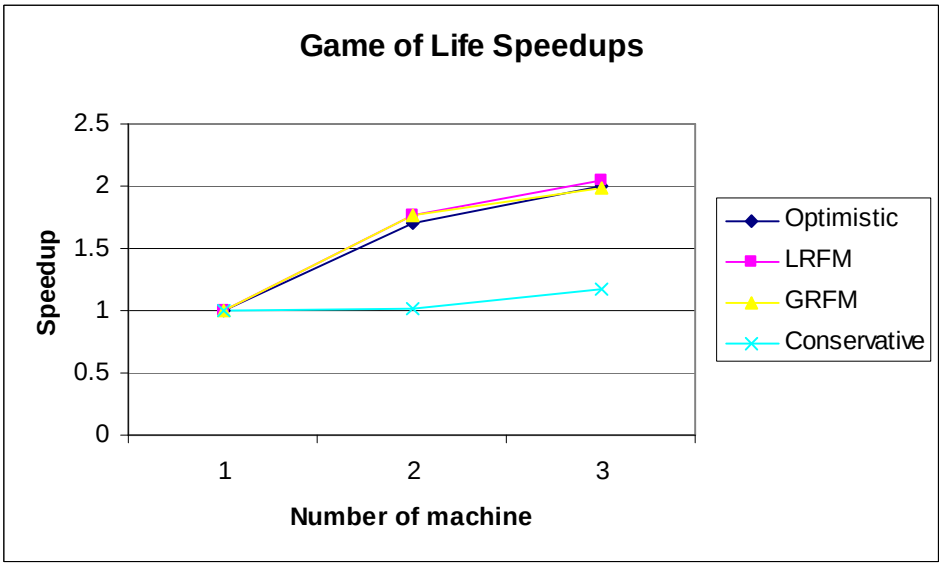


Figure 69. Game of life model speedups with regards to execution on one node

€ **Synapsin-Vesicle Reaction Model**

This model consists of 676 cells arranged in a 26x26 mesh with a total execution time of 3.7621 seconds when run on standalone CD++. Figure 70 represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

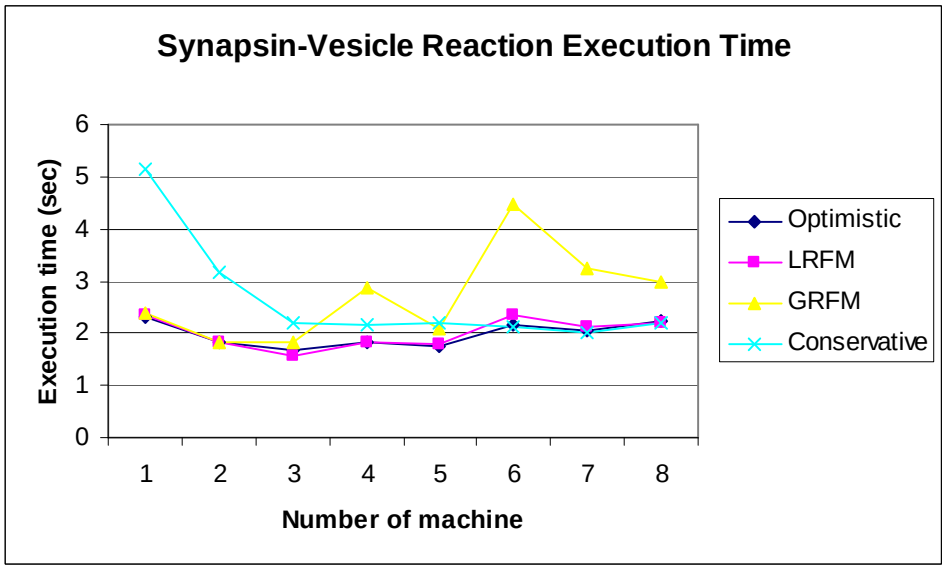


Figure 70. Synapsin-vesicle model execution time on 4 different simulators

From the graph, we can see that the optimistic and LRFM-based simulators, produce very close results on 1 to 8 nodes. Also, the GRFM-based simulator has similar results for 1, 2, 3, and 5 nodes. However, it degrades the performance when 4, 6, 7, and 8 nodes are participating. On the other hand, the conservative simulator shows different behavior as the number of nodes increases. As seen on the graph, the conservative simulator improves the total execution time significantly when more than 2 nodes are available. Again, as in the previously discussed models, as the number of computing nodes increases, the GRFM-based simulator has the lowest performance among other ones. The main reason is communication overhead among the participating LPs which leads in a noticeable time added to the duration of the model execution.

Figure 71 represents the speedups of the model execution times with respect to execution on one node for each particular simulator.

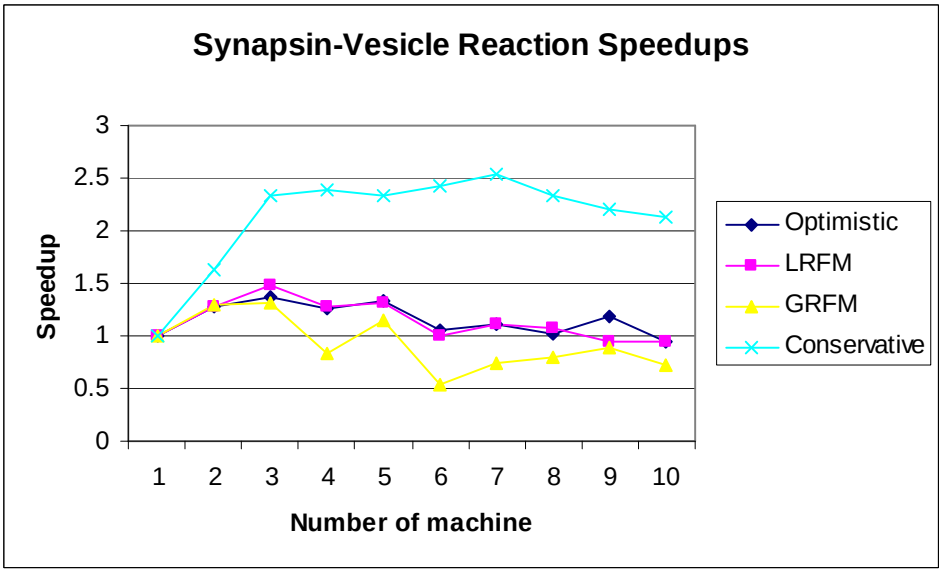


Figure 71. Synapsin-vesicle model speedups with regards to execution on one node

€ Fire Propagation Model

This model consists of 900 cells arranged in a 30x30 mesh with a total execution time of 6.2145 seconds when run on standalone CD++. Figure 72 represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

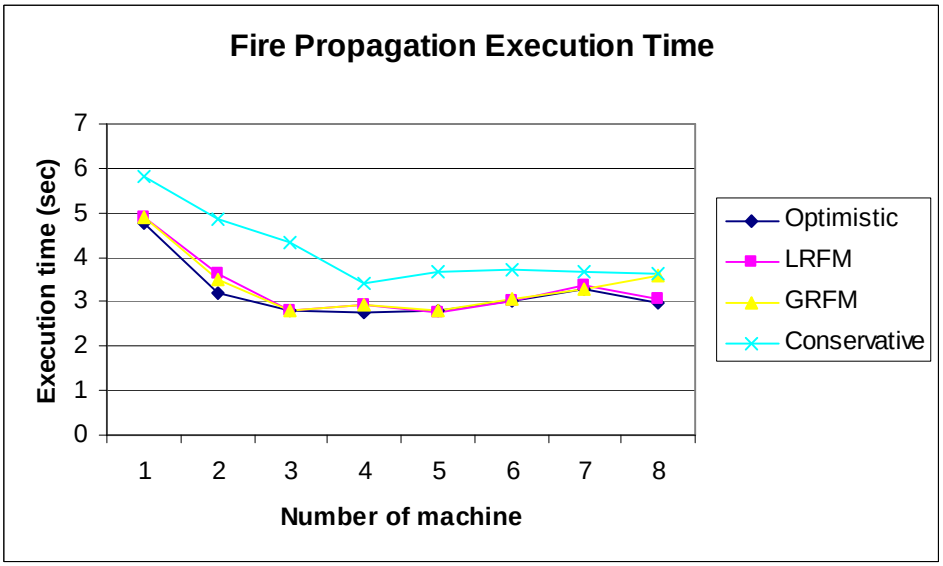


Figure 72. Fire propagation model execution time on 4 different simulators

As seen on the graph, our parallel simulators significantly improved the execution time of the fire propagation model. The three optimistic simulators produced very similar results on 1 to 7 nodes. For this model, we can definitely remark that the optimistic simulators outperform the conservative one. For the optimistic simulators the best results were achieved on 5 nodes, while the conservative one had its lowest execution time on 4 nodes. The speedups of the model execution times with respect to execution on one node for each particular simulator are given in Figure 73, which provides a better explanation of the performances achieved.

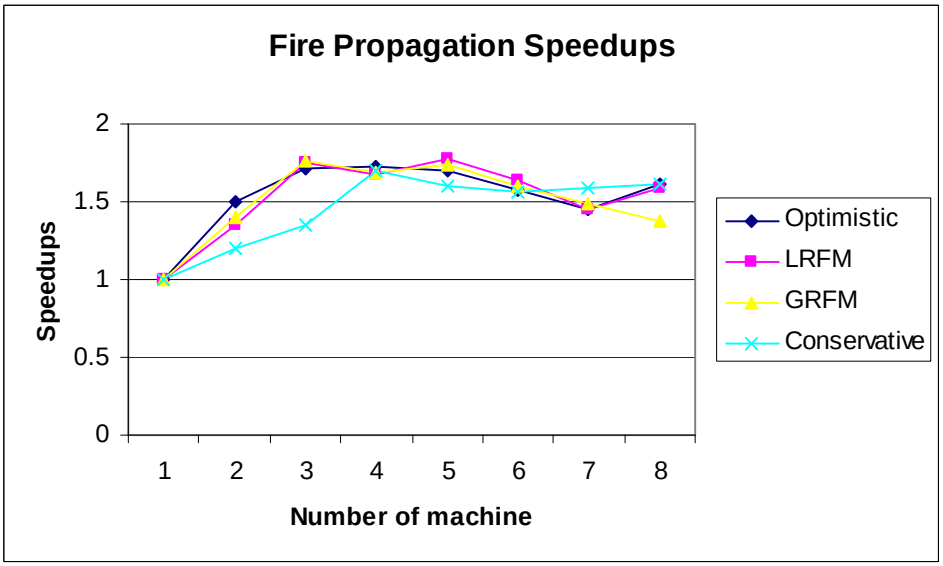


Figure 73. Fire propagation model speedups with regards to execution on one node

- **Ship Evacuation Model**

This model consists of 400 cells arranged in a 20x20 mesh with a total execution time of 6.4327 seconds when run on standalone CD++. Figure 74 represents the execution time resulting from running the model with four different simulators on 1 to 8 nodes.

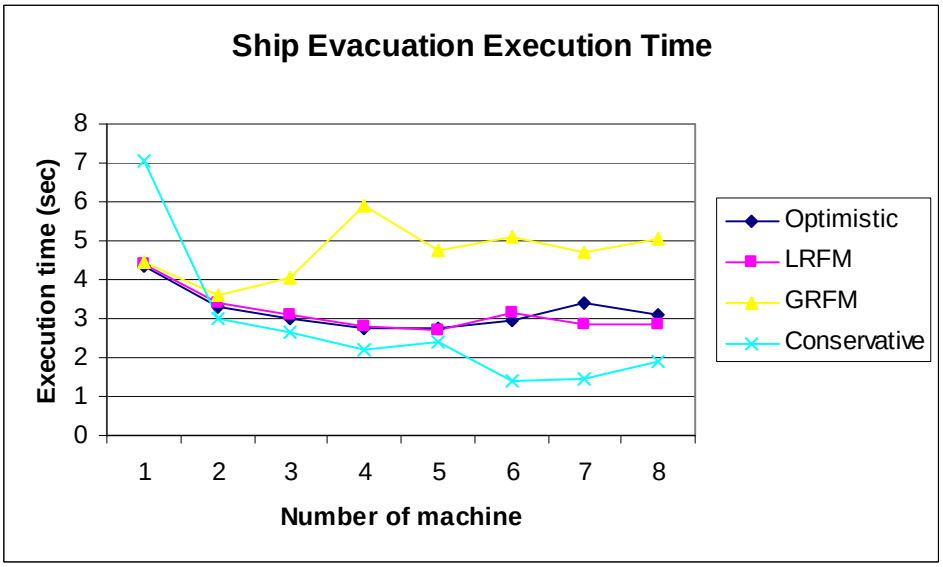


Figure 74. Ship evacuation model execution time on 4 different simulators

From the execution time graph, we can see that the conservative simulator outperforms the other three simulators. This is due to the causality-error avoidance

mechanism of this simulator which avoids rollbacks and anti-message flows. The optimistic and LRFM-based simulators produce very similar results for 2 to 6, and 8 nodes. However, the GRFM-based simulator does not present good results. This is mainly due to the huge message-passing mechanism among the LPs who are sending messages back and forth reporting information about their rollbacks. To prove this, we can see that the GRFM-based simulator reduces the execution time when there are two computing nodes, but as the number of nodes increases, the performance degrades. The speedups of the model execution times with respect to execution on one node for each particular simulator are given in Figure 75. The speedups graph shows that except for the GRFM-based simulator, the other simulators have improved the execution time.

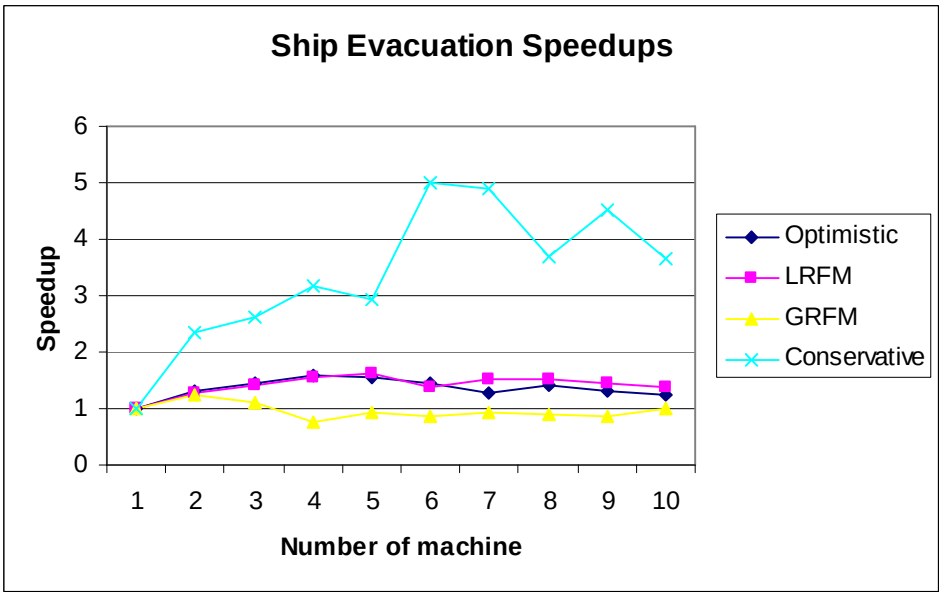


Figure 75. Ship evacuation model speedups with regards to execution on one node

7.4. ADDITIONAL TESTINGS OF PCD++ SIMULATOR

In this section we show a different sort of tests that we used. For this purpose, we built a simple model consisting of an *n* by *n* grid with initial value of zero for all the cells (except the one located in the center of the grid). As the simulation runs, the value "1" propagates through all cells starting from the central cell towards four directions (N/S/E/W) until the value of all cells are changed from "0" to "1". We collected similar tests for the model but changing the size to 5x5, 10x10, and 30x30 cell space. Figure 76

and Figure 77 Represent the 5x5 Grid model definition in CD++ and the propagation of "1" throughout the grid respectively.

```
[grid-rule]
type : cell
dim : (5,5)
delay : transport
defaultDelayTime : 100
border : nowrapped
neighbors :                          grid-rule(-1,0)
neighbors : grid-rule(0,-1)   grid-rule(0,0)  grid-rule(0,1)
neighbors :                          grid-rule(1,0)
initialvalue : 0
initialrowvalue :  0     00000
initialrowvalue :  1     00000
initialrowvalue :  2     00100
initialrowvalue :  3     00000
initialrowvalue :  4     00000
localtransition : myrule

[myrule]
rule : 1 100 { trueCount>1 }
rule : {(0,0)} 100 { t }
```

Figure 76. 5x5 Grid model definition in CD++

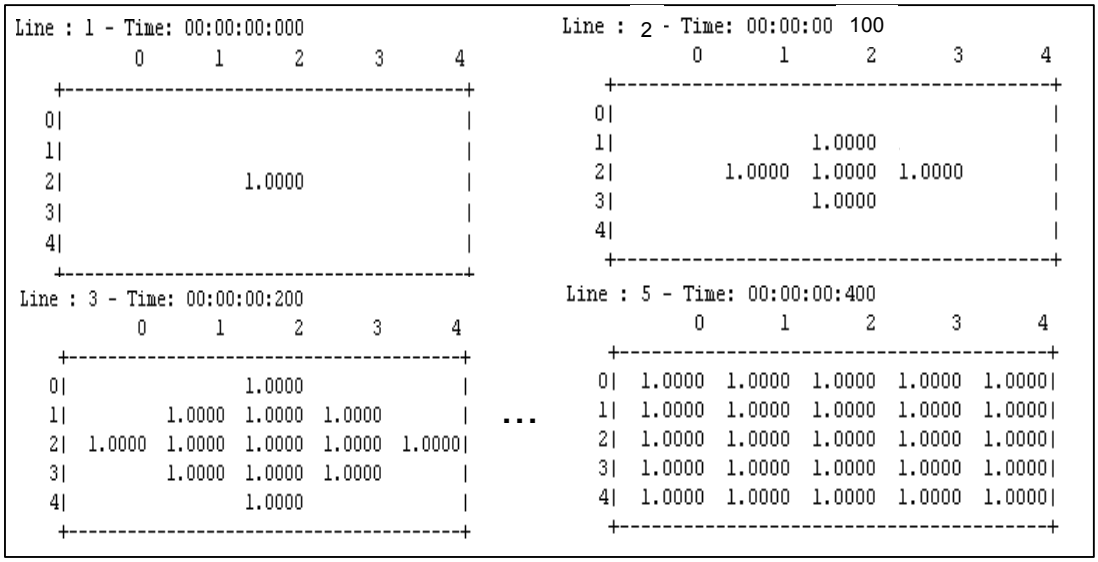


Figure 77. Propagation of "1" throughout the grid

This model allows observing the performance of the simulator when the grid's size is increased incrementally. Two different types of tests were collected: 1) analyzing the performance of the simulator by introducing fixed and variable delays into the cells' evaluation rules, 2) testing the robustness of the simulator as the complexity of the model is increased. The following sections explain these two testing scenarios in details.

7.4.1. Performance of PCD++

The first testing scenario was running the Grid model on *1* to *n* machines where *n* is the size of the model (*n x n*). Every set of simulation was performed on optimistic PCD++, LRFM-based PCD++, and GRFM-based PCD++ simulator. The following figures will illustrate the simulation time behavior as well as the speedups. For very *n x n* Grid model first the simulations were collected by inserting fixed delay time in the rules evaluating cells' states. Secondly, the same sets of simulations were performed but this time modifying the delay to be variable.

```
rule : 1 100 {trueCount > 1}
```

(a)

```
rule : 1 {(100 * truecount)}  {trueCount > 1}
```

(b)

Figure 78 shows the difference between fixed and variable delayed rules.


```
rule : 1 100 {trueCount > 1}
```

(a)

```
rule : 1 {(100 * truecount)}  {trueCount > 1}
```

(b)

Figure 78. (a) Grid rule with fixed delay, (b) Grid rule with variable delay

- **5x5 Grid Model with fixed and variable delays**

The model consisted of 25 cells arranged in a 5x5 mesh. Figure 79 and Figure 80 represent the execution time resulting from running the model with three different simulators on 1 to 5 computing nodes with fixed and variable delays.

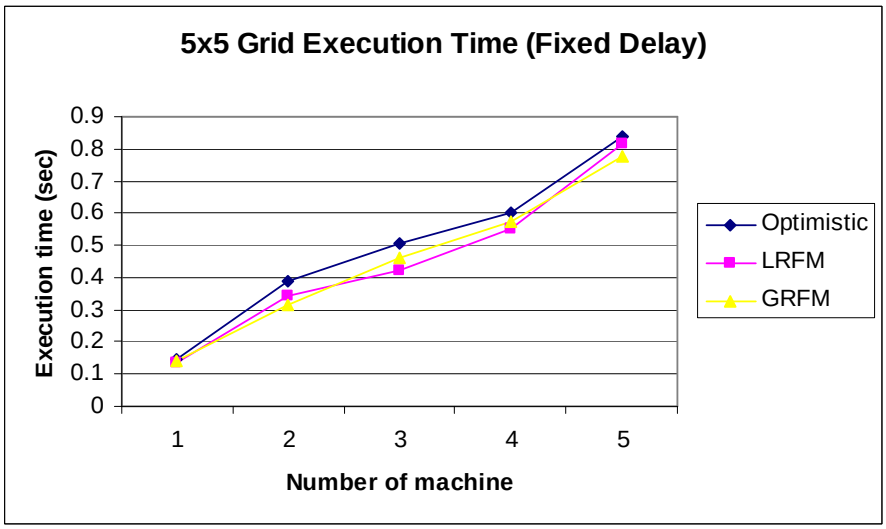


Figure 79. Simulation results of 5x5 Grid model with fixed delay

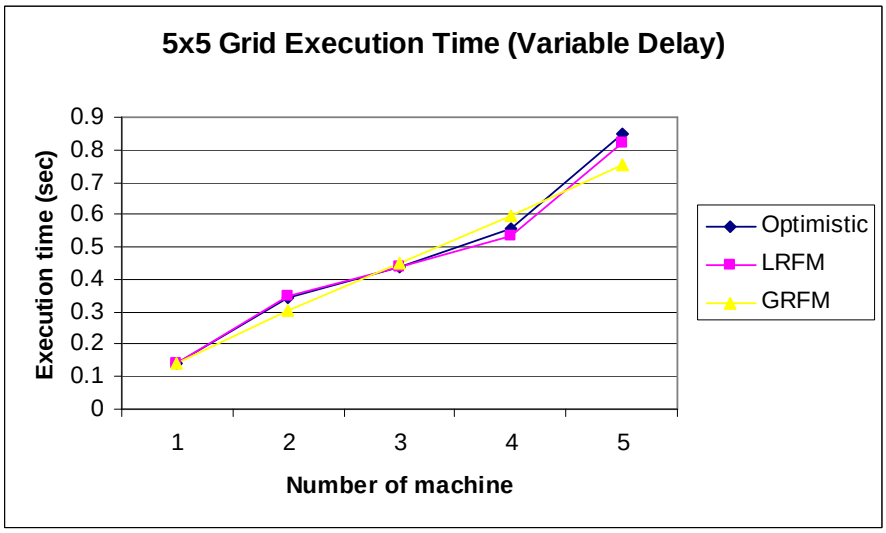


Figure 80. Simulation results of 5x5 Grid model with variable delay

As the graphs show, the execution times of the 5x5 Grid model with variable and fixed delay are very close to each other. This is mainly due to the small size of the model where does not get affected by changing the delay type at which cells' states are changed. Aside from this, again due to the small size of the model increasing the number of computing nodes does not improve the execution time, rather it worsens the situation. The reason is that the actual execution time of the model is noticeably smaller than the time needed to initialize the additional nodes plus the significant communication overhead among them. The following speedup graphs clarify this fact.

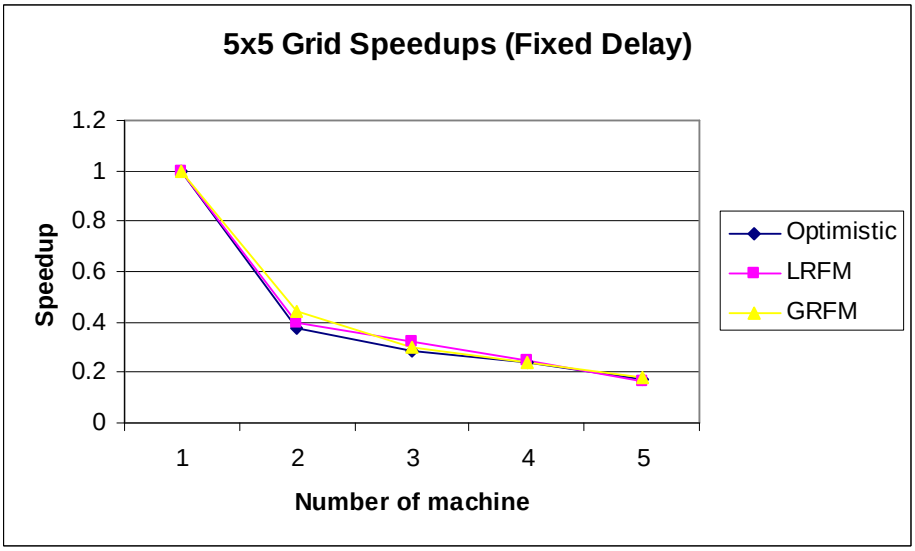


Figure 81. Speedup results of the 5x5 Grid model with fixed delay

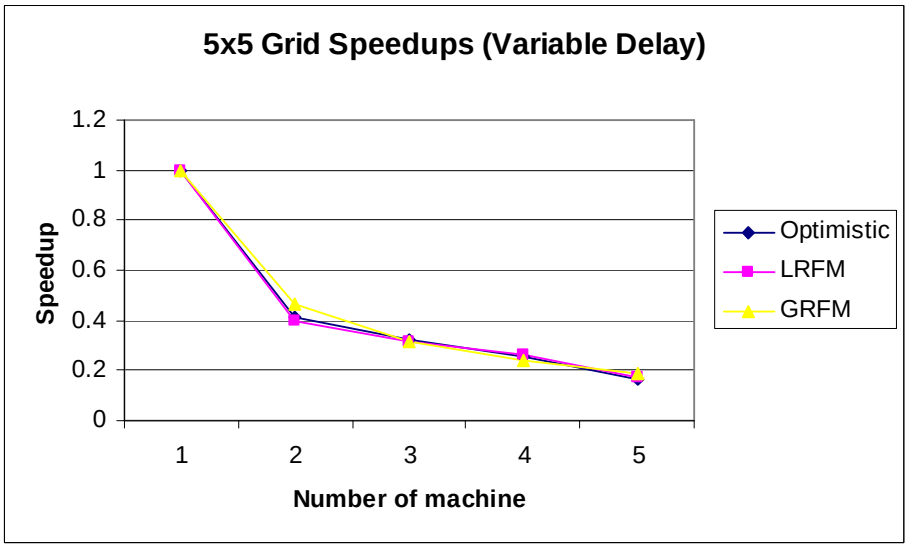


Figure 82. Speedup results of the 5x5 Grid model with fixed delay

€ 10x10 Grid Model with fixed and variable delays

By expanding the Grid model into a 10x10 mesh consisting of 100 cells, the effect of different type of delay can be observed. However, the model's size is still small and the additional computing nodes do not improve the execution time. For the 10x10 Grid model we have only run the simulation on the optimistic PCD++. Figure 83 represents these results.

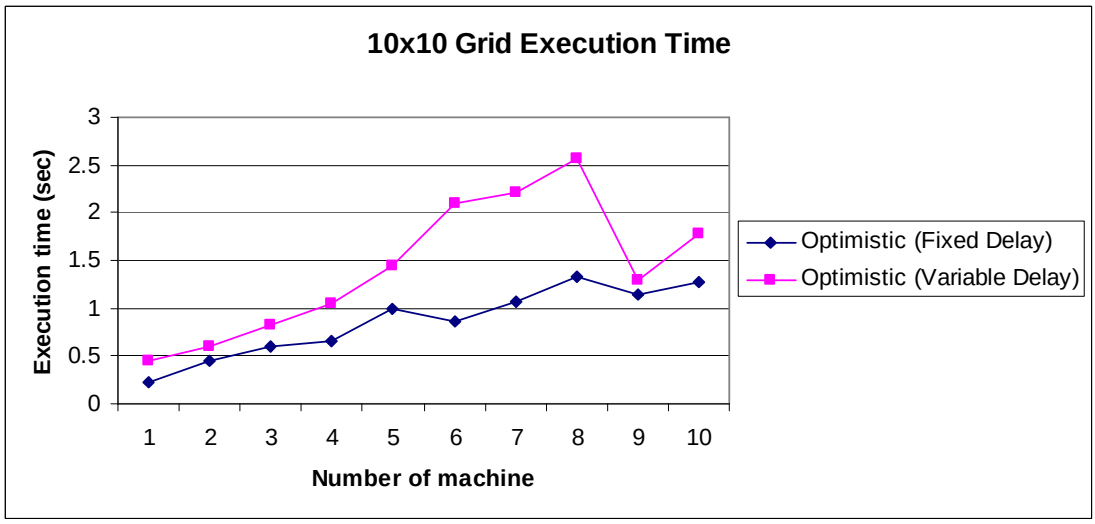


Figure 83. Simulation results of 10x10 Grid model with fixed and variable delay

As seen on the graph, the variable delay adds up a noticeable computing time to the simulation. Figure 84 illustrates the resulting speedups which prove that the model is too small to be executed on more than one machine.

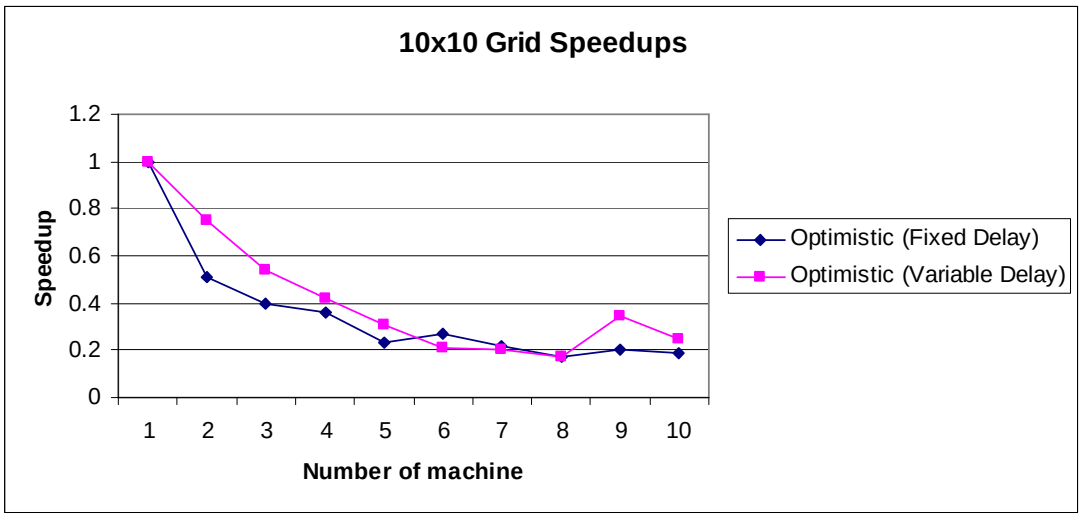


Figure 84. Speedup results of the 10x10 Grid model with fixed and variable delay

€ 30x30 Grid Model with fixed and variable delays

To observe the performance of the optimistic PCD++ simulator, we repeated the simulation scenarios of the 10x10 Grid model by expanding the model into 900 cells arranged in a 30x30 mesh. Figure 85 illustrates the execution results.

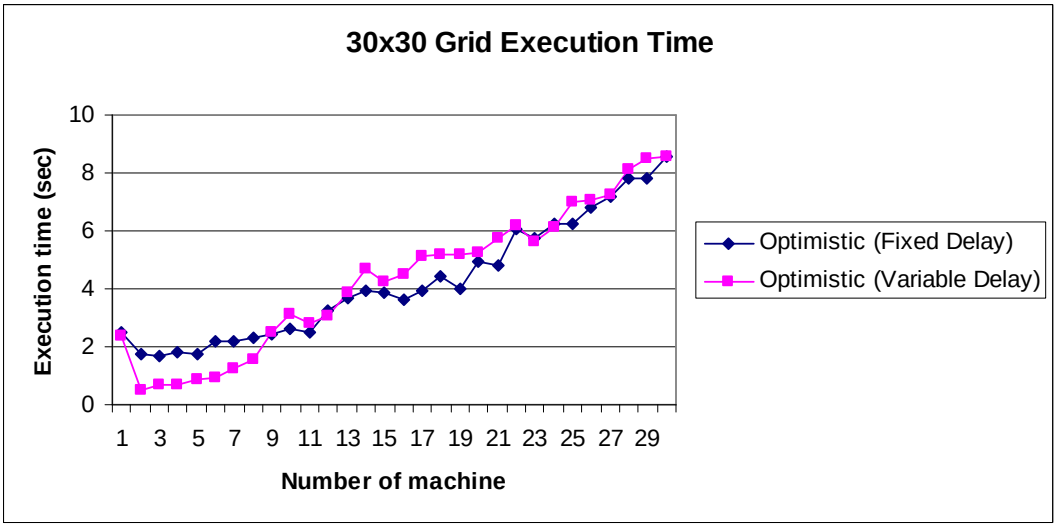


Figure 85. Simulation results of 30x30 Grid model with fixed and variable delay

From the above graph we can see interesting results where the model with variable delays outperforms the one with fixed delay. The size of the model looks ideal enough to be executed on multiple nodes in contrast to the 5x5 and 10x10 Grid model. For both types of delays, running the model on 2 to 8 nodes reduces the execution time compared to simulation on single node. Almost after 8 nodes, adding extra computing nodes increased the execution time. As mentioned before, this is due the significant startup time and communication overheads which exceed the execution time on single machine. The speedup graphs prove these results.

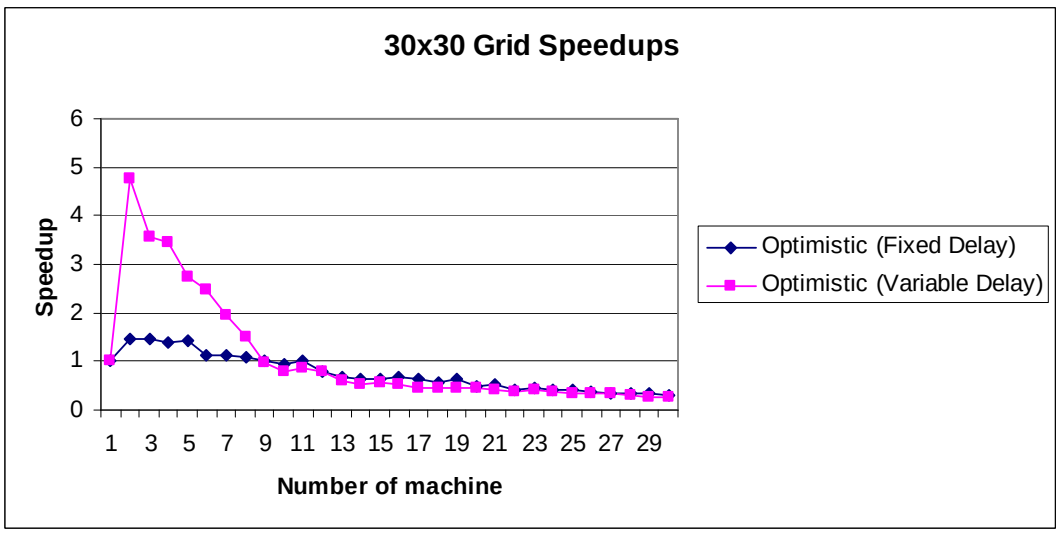


Figure 86. Speedup results of the 30x30 Grid model with fixed and variable delay

To better illustrate the result achieved from analyzing the performance of PCD++ in handling different model sizes we are presenting the execution results of the 5x5, 10x10, and 30x30 Grid model on single graphical presentations. The following 3-D graphs reflect these combinations for both fixed and variable delay model.

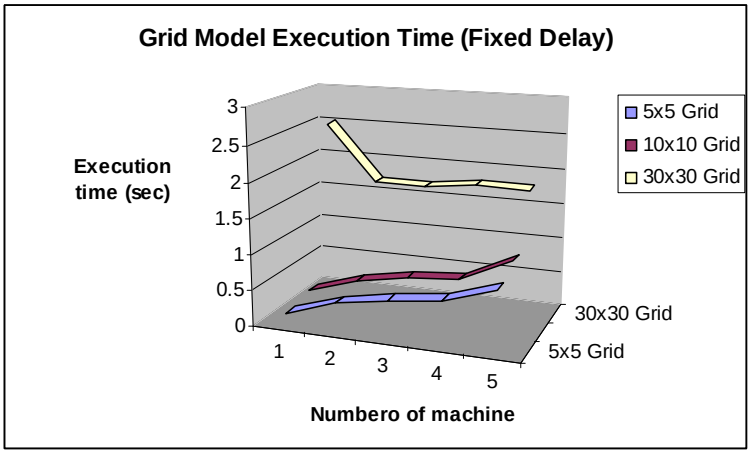


Figure 87. 3-D representation of Grid model with fixed delay

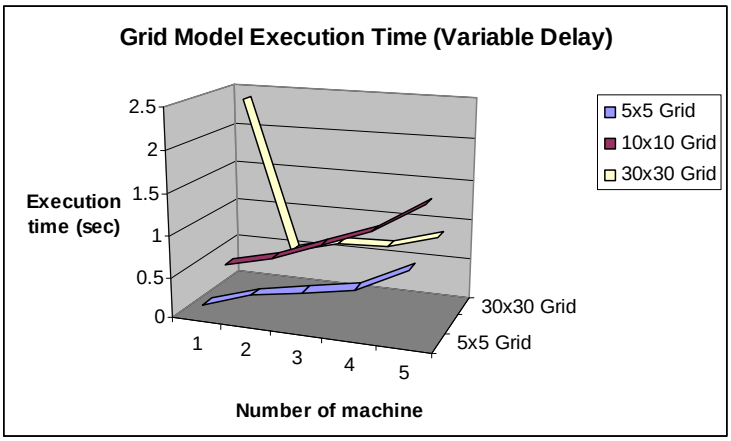


Figure 88. 3-D representation of Grid model with variable delay

7.4.2. Robustness of PCD++

The second testing scenario was running the 5x5, 10x10, and 30x30 Grid model on single machine and each time changing the complexity level of the model. The complexity level was modified by adding extra computations within the cells' evaluation rules. For example, in the Grid model when a cell has one or more neighbors holding the value "1", after a fixed or variable delay the cell's value is changed from "0" to "1". In normal case this can be done by simply changing the cell's state value instantly. However, to add

complexity to the model we added extra computation to increase the total time at which a cell's state is changed from "0" to "1". We have increased the complexity level by introducing a function in the CD++ simulator that loops for *n* times. The complexity level (i.e. value of *n*) can be modified by the user at run time. The following snapshot represents the Grid model's rule that uses complexity levels. The function *ComplexityFunc* takes two parameters: *n* is the complexity level (represents the number of nested loops which will add extra computation time), the second parameter defined the new value of the cell for which the rule is evaluated (i.e. modifying the cell's state to "1" if it has neighbors with state equal to "1").

```
rule : {ComplexityFunc(n, 1)} 100 {trueCount > 0}
```

Figure 89. Adding complexity level to cells' evaluation rules

The purpose of using complexity levels on one machine was to increase the execution time without modifying the model's size. Although we have run the complexity tests on 5x5, 10x10, and 30x30 Grid model, but we wanted to take a deeper look at the performance of the optimistic PCD++ simulator and observe how it performs in the presence of complex and long-time simulations. As in the first testing scenario, we have used both types of delays (i.e. fixed and variable). The following graphs provide the simulation results.

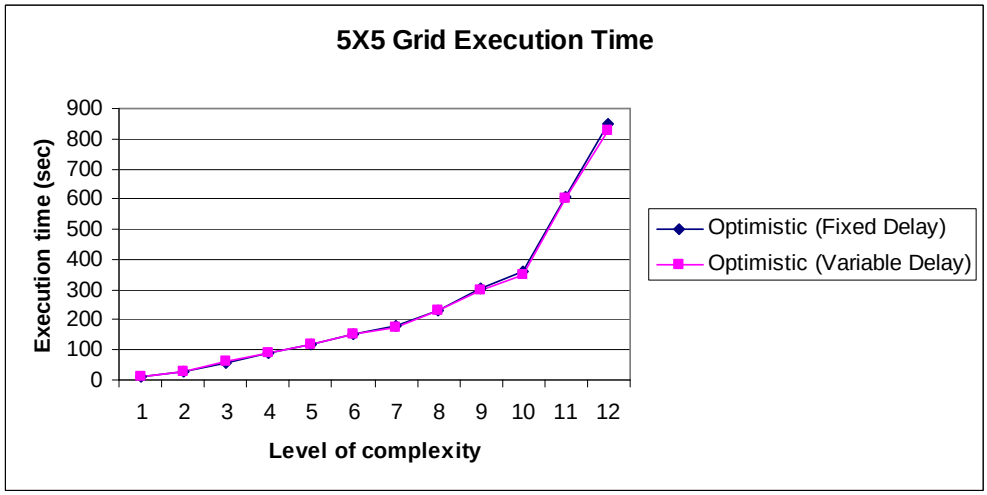


Figure 90. Execution results of 5x5 Grid model under 12 different complexity levels

As seen on Figure 90, due to the small size of the model, both types of delays produce similar results. The complexity level is increased gradually and as a result the

execution time is increased from a couple of seconds to almost 850. Next we present the 10x10 Grid model's simulation results.

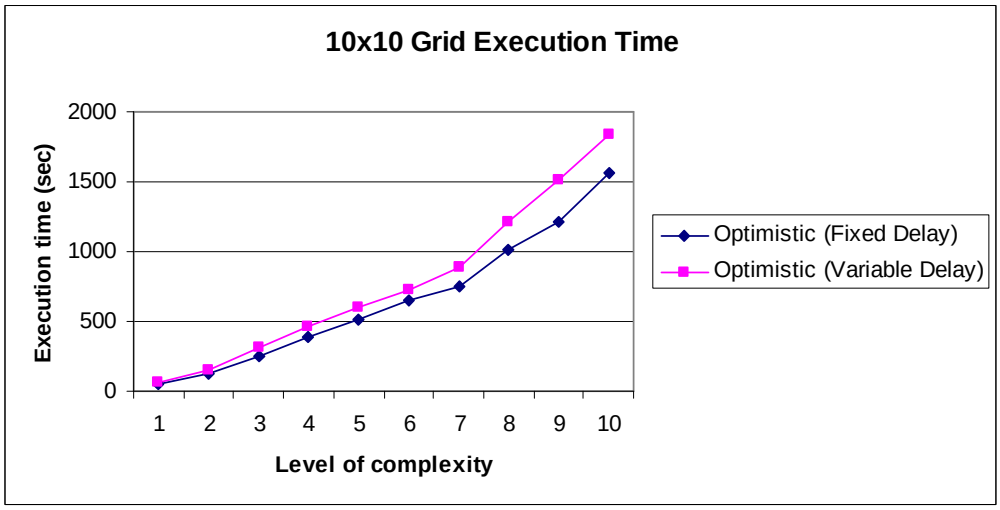


Figure 91. Execution results of 10x10 Grid model under 10 different complexity levels

The execution time of the 10x10 Grid model reflects the effect of complexity levels more clearly than the 5x5 model. Since the model is four times larger, therefore the execution time is noticeably higher and more sensitive to complexity levels.

Finally the complexity simulations were tested for the 30x30 model. Results are presented in Figure 92.

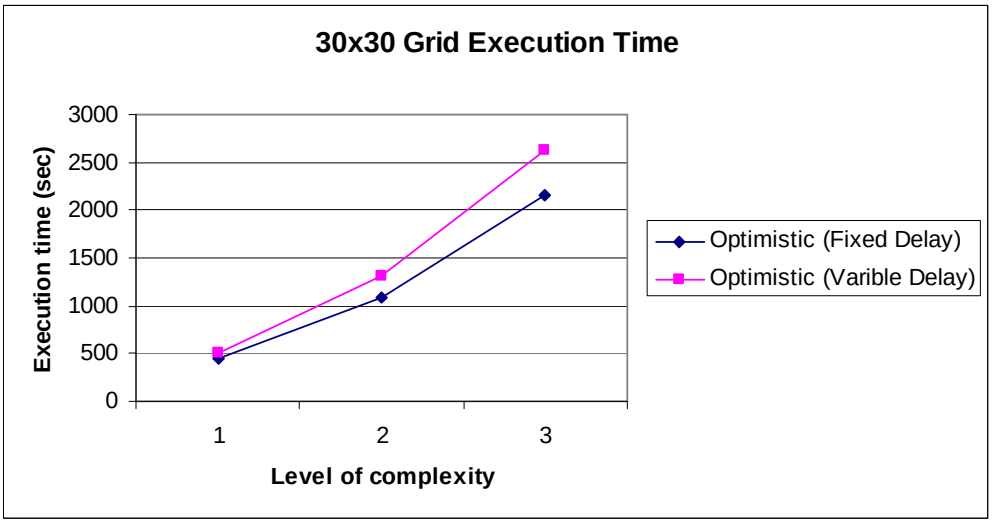


Figure 92. Execution results of 30x30 Grid model under 3 different complexity levels

As seen on the graph, due to the large size of the model the effect of complexity on increasing the execution time is very sensible and only three levels of complexity present the behavior that was obtained in 10x10 Grid model in 10 complexity levels.

The following graphs represent the 3-D illustration of the complexity-based tests. Figure 93 shows the scenario of fixed delay Grid model, while Figure 94 represents the variable delay version of the model with three different complexity levels.

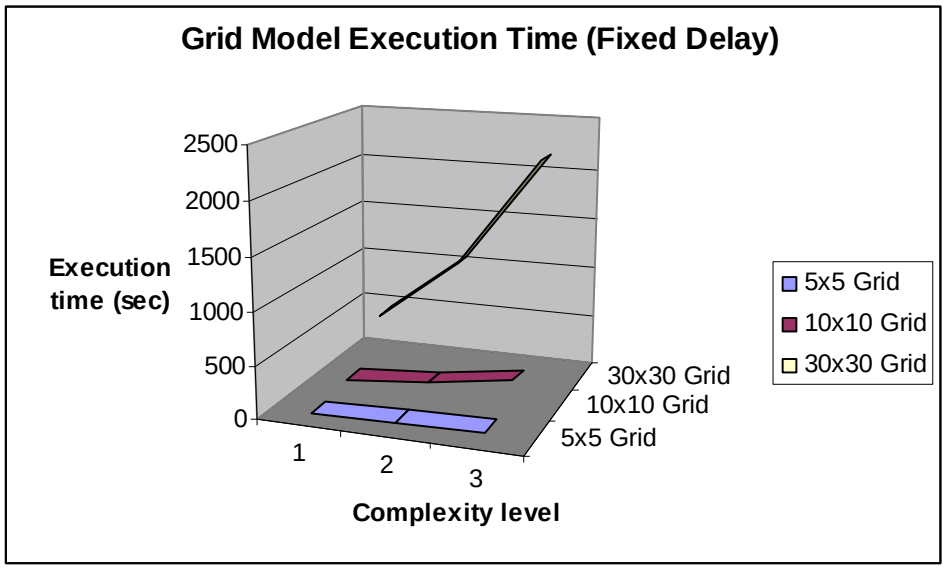


Figure 93. 3-D representation of fixed delay Grid model with three levels of complexity

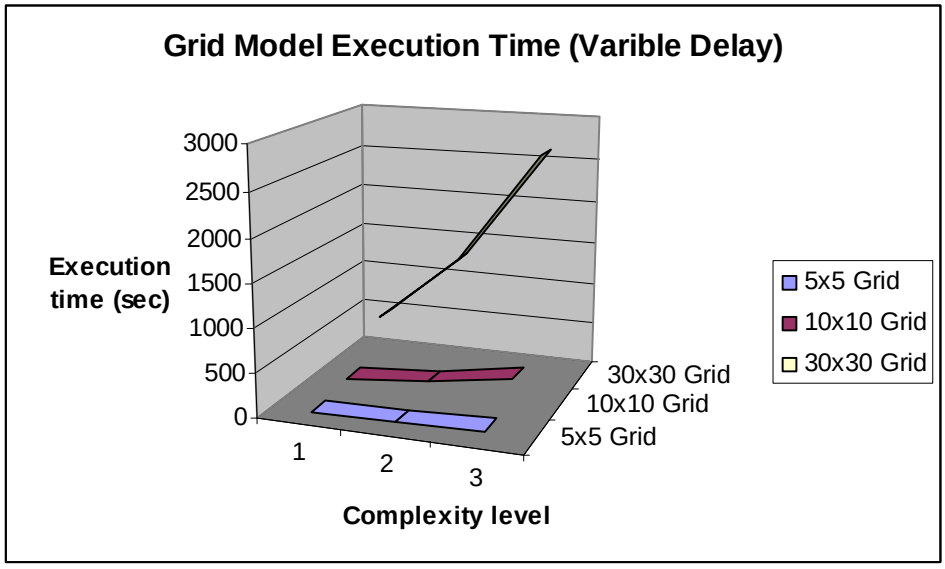


Figure 94. 3-D representation of varible delay Grid model with three levels of complexity

CHAPTER 8 CONCLUSIONS AND FUTURE WORK

This work presented the parallel simulation of DEVS and Cell-DEVS models using PCD++, a parallel and distributed environment based on the Time Warp optimistic synchronization protocol. PCD++ serves as an extension to the CD++ toolkit which was developed by previous researcher [Liu06] aiming at exploiting parallelism for the purpose of fast and efficient simulation of complex models. The concept of Parallel and Distributed Simulation was presented.

The most challenging problem of parallel and distributed simulation i.e. Synchronization among nodes (LPs) was discussed by over viewing the three different types of synchronization strategies for event driven simulations: no synchronization at all, pessimistic (conservative) synchronization, and optimistic synchronization.

We illustrated the software architecture of the purely optimistic parallel CD++ simulator (PCD++). The layered architecture of the optimistic PCD++ simulator consists of five layers (from top to bottom): model, PCD++, Time Warp - WARPED, and the operating system, where ach layer was explained in details. A variety of optimization strategies of the Time Warp kernel were pointed out and discussed thoroughly. Some optimizations in terms of GVT calculation, dynamic memory management, and state management were mentioned.

We have analyzed the performance of our two existing parallel CD++ simulators, namely Conservative PCD++ simulator [Tro01] and Optimistic PCD++ simulator [Liu06]. We looked at the design and implementation of these two simulators and compared their structures as well as functionalities in parallel and distributed simulations.

The hierarchical structure of the conservative PCD++ simulator was compared against the flattened structure of the optimistic PCD++ simulator. The migration from a hierarchical structure to a flattened structure was illustrated as two major modifications; i.e. the departure from conservative-based simulator to an optimistic-based simulator, and flattening the structure of the simulator. Then it was illustrated how the optimistic PCD++ simulator deals with the communication overhead dilemma by using the flattened structure.

A set of models were designed and implemented in CD++, and they were presented to illustrate the capability of Cell-DEVS formalism in building such models. The Ship Evacuation model illustrated an emergency ship evacuation scenario. Synapsin-Vesicle Reaction model presented the reserve pool of synaptic vesicles in a presynaptic nerve terminal. The Fire Spread model represented a fire propagation scenario in forest based on Rothermel's mathematical definition. The Game of Life model defined the standard Game of Life using a two-dimensional grid.

Aiming at improving the performance of the optimistic simulator, we modified the WARPED kernel to handle rollbacks in a more efficient way. We presented two new algorithms that we have implemented in WARPED kernel. The *Near-perfect State Information* protocol was discussed and after that our new algorithms; Local Rollback Frequency Model (LRFM) and Global Rollback Frequency Model (GFRM) were presented.

Finally, we have run a variety of tests to analyze the performance of our existing PCD++ simulators; the optimistic and the conservative as well as our LRFM and GRFM Time Warp-based protocols. The main goal of these tests was to show the maximum capability of the two mentioned PCD++ simulators in terms of handling the number of nodes driving the simulation, complexity of the model, and the size of the model for the models introduced in Chapter 5.

7.1. FUTURE WORK

With regard to testing the performance of PCD++, there are several topics of interest for future research:

- Models with longer execution time are required to be run on PCD++. This gives the chance to catch unexpected errors especially in terms of timeouts and broken pipes. In most of the cases, the MPI communication interface was not able to handle timings (long waits due to the size and complexity of the model) properly. Thus, a deeper investigation at the MPI level is suggested.

- Testing different partitioning strategies as opposed to the one used by current PCD++ would provide lots of feedback about the capability of the simulator.
- A dynamic load balancing mechanism which allows for run-time balancing of the load would be a great solution to ensure that load is divided equally among the available nodes.
- Incorporating algorithms such as moving time windows (MTW) [Fuj00, Fuj03] and the Filter algorithm [Pra91] and comparing the resulted simulators with LRFM- and GRFM-based simulators gives key ideas on how to control optimism efficiently.
- Aside from different testing strategies, profiling the simulator would provide very detailed information and could be used to modify the underlying C++ codes to reduce runtime.

REFERENCES

[Ala07] Al-aubidy, B.; Dias, A.; Bain. R.; Jafer, S.; Dumontier, M.; Wainer, G.; Cheetham, J. ”Advanced DEVS models with applications to biomedicine”. *Artificial Intelligence, Simulation and Planning*. Buenos Aires, Argentina. AIS 2007.

[Ahm05] Ahmed M.; Yonis, K.; Elshafei, M.; Wainer, G. "Building a tool for modeling and simulation of computer networks". Proceedings of the 38thIEEE/SCS Annual Simulation Symposium. San Diego, CA. U.S.A. 2005.

[Ben90] Benfenati, F., Valtorta, F., Greengard, P.; “Computer Modelling of Synapsin 1 Binding to Synaptic Vesicles and F-actin”. Implications for Regulation of Neurotransmitter Release. 1990.

[Bry77] Bryant, R. E. "Simulation of packet communication architecture computer systems". Massachusetts Institute of Technology. Cambridge, MA. USA. 1977.

[Cas93] Cassandras, C. G. "Discrete event systems: Modeling and performance analysis". Boston, MA: Aksen Associates, 1993.

[Cha78] Chandy, K. M.; Misra J. "Distributed simulation: A case study in design and verification of distributed programs". IEEE Transactions on Software Engineering. pp.440-452. 1978.

[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.

[Che98] Chetlur, M.; Abu-Ghazaleh, N.; Radhakrishnan, R.; Wilsey, P. A. "Optimizing Communication in Time-Warp Simulators". Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS'98). 1998.

[Che04] Cheon, S.; Seo, C.; Park, S.; Zeigler, B. "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.

[Dav00] Davila, J.; Uzcagegui, M. "GALATEA: A multi-agent, simulation platform". Proceedings of the International Conference on Modeling, Simulation and Neural Networks. Merida, Venezuela. 2000.

[Del02] de Lara, J.; Vangheluwe, H. "ATOM3: A tool for multi-formalism modeling and meta-modeling". European Joint Conference on Theory and Practice of Software. Grenoble, France. 2002.

[Dia01] Diaz, A.; Veronica, V.; Wainer, G. "Defining congestion control mechanism in ATLAS". In proceedings of Summer Computer Simulation Conference, Orlando, FL. 2001.

[Fil02] Filippi, J. B.; Bernardi, F.; Delhom, M. "The JDEVS modeling and simulation environment". Proceedings of the Integrated Assessment and Decision Support Conference (IEMSS'02). pp. 283-288. Lugano, Switzerland. 2002.

[Fre02] Frey P.; Radhakrishnan, R.; Carter, H.W.; Wilsey, P. A.; Alexander, P. "A formal specification and verification framework for Time Warp-based parallel simulation". IEEE Transactions on Software Engineering. Vol. 28. No. 1. 2002.

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

[Fuj00] Fujimoto, R. M. "Parallel and Distributed Simulation Systems". A Wiley-Interscience publication. ISBN 0-471-18383-0. 2000.

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

[Fuj03] Fujimoto, R. M. "Distributed simulation systems". Proceedings of the 2003 Winter Simulation Conference. pp. 124-134. 2003.

[Gar70] Gardner, M. " Mathematical games: The fantastic combinations of John Conway's new solitaire game "life"". *Scientific American* 223: 120 - 123. 1970.

[Gia76] Giambiasi, N.; Miara, A. "SILOG: A practical tool for digital logia circuit simulation". Proceedings of the 16thD.A.C. San Diego. 1976.

[Gli02a] Glinsky, E.; Wainer, G. "Definition of Real-Time simulation in the CD++ toolkit." Proceedings of the Summer Computer Simu lation 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. "New Techniques for Parallel Simulation of DEVS and Cell-DEVS Models in CD++". M. A. Sc. Thesis. Carleton University. Canada. 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.

[Gru93] Grunwald, D.; Zorn, B. "CustoMalloc: Efficient Synthesized Memory Allocators". Software - Practice and Experience. Vol. 23, pp. 851-869. 1993.

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

[Jaf07] Jafer, S.; Wainer, G. "An Environment for Advanced Parallel Simulation of Cellular Models". Proceedings of the International Conference on Unconventional Computation 2007 – UC'07.

[Jan06] Janousek, V., Kironsky, E. "Exploratory Modeling with SmallDEVS". Proceedings of ESM 2006, pp. 122-126, ISBN 90-77381-30-9, Toulouse, France, 2006.

[Jef85] Jefferson, D. "Virtual Time". ACM Transactions on Programming Languages and Systems. 7(3):405-425. 1985.

[Kim00] 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.

[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.

[Klu01] Klüpfel, W.; Meyer-König, T.; Wahle, J.; Schreckenberg, M. "Microscopic Simulation of Evacuation Processes on Passenger Ships", Theoretical and Practical Issues in Cellular Automata, pp. 63-71, Springer-verlag 2001.

[Knu73] Knuth, D. E. "Fundamental Algorithms". Vol. 1. The Art of Computer Programming. Second Edition. Addison-Wesley. 1973.

[Kof03] Kofman, E.; Lapadula, M.; Pagliero, E. "PowerDEVS: a DEVS-based environment for hybrid system modeling and simulation". Technical Report LSD0306. LSD, University Nacional de Rosario. 2003.

[Lin91] Lin, Y. B.; Lazowska, E. D. "A study of Time Warp Rollback Mechanisms". ACM Transactions on Modeling and Computer Simulations. Vol. 1, No. 1. January 1991.

[Lin93] Lin, Y. B.; Preiss, B. R.; Loucks, W. M.; Lazowska, E. D. "Selecting the Checkpoint Interval in Time Warp Simulation". Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'93). 1993.

[Liu06] Liu, Q. "Distributed Optimistic Simulation of DEVS and Cell-DEVS Models with PCD++". M. A. Sc. Thesis. Carleton University. Canada. 2006.

[Low99] Lowry, M. C.; Ashenden, P. J.; Ha wick, K. A. "Distributed High-Performance Simulation using Time Warp and Java". Technical Report, Department of Computer Science. The University of Adelaide. South Australia. 1999.

[Lub91] Lubachevsky, B.; Weiss, A.; Shwartz, A. "An analysis of rollback-based simulation". ACM Transactions on Modeling and Computer Simulation. Vol. 1, No. 2, pp. 154-193. April 1991.

[Mad05] Madhoun, R.; Wainer, G. "Modeling battlefield scenarios in Cell-DEVS". Proceedings of SISO Fall Interoperability Workshop. San Diego, CA. U.S.A. 2005.

[Mad06] Madhoun, R. "Web Service-Based Distributed Simulation of Discrete Event Models". M. A. Sc. Thesis. Carleton University. Canada. 2006.

[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.

[Mar99] Martin, D. E.; McBrayer, T. J.; Radhakrishnan, R.; Wilsey, P. A. "WARPED - A Time Warp Parallel Discrete Event Simulator (Documentation for version 1.0)". Available at: http://www.ececs.uc.edu/~paw/warped/doc/index.html. 1999. [Accessed July, 2007].

[MPI95] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report. Available via: <http://www.mpi-forum.org>. [Accessed July, 2007].

[Nut06] Nutaro, J. ADEVS website. Available at: http://www.ece.arizona.edu/~nutaro. [Accessed July, 2007].

[Pra91] Prakash, A.; Subramanian, R. "Filter: An algorithm for reducing cascaded rollbacks in optimistic distributed simulations". Proceedings of the 24th Annual Simulation Symposium. pp. 123-132. 1991.

[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.

[Rad97] Radhakrishnan, R.; Moore, L.; Wilsey, P. A. "External Adjustment of Runtime Parameters in Time Warp Synchronized Parallel Simulators". Proceedings of the 11th International Parallel Processing Symposium. 1997.

[Rad98] Radhakrishnan, R.; Martin, D. E.; Chetlur, M.; Rao, D. M.; Wilsey, P.A. "An Object-Oriented Time Warp Simulation Kernel". Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98). Vol. LNCS 1505, pp. 13-23. Springer-Verlag. 1998.

[Rey88] Reynolds, P. "A Spectrum of Options for Parallel Simulation". Proceedings of the 1988 Winter Simulation Conference, 325-332. 1988.

[Rod99] Rodriguez D.; Wainer, G. "New Extensions to the CD++ Tool". Proceedings of the 32nd SCS Summer Computer Simulation Conference. Vancouver, Canada. 1999.

[Ron94] Ronneren, R.; Ayani, R. "Adaptive checkpointing in Time Warp". Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS'94). 1994.

[Rot72] Rothermel, R. "A mathematical model for predicting fire spread in wild-land fuels". Research Paper INT-115. Ogden, UT: U.S. Department of Agriculture, Forest Service, Intermountain Forest and Range Experiment Station. 1972.

[Sar98] Sarjoughian, H. S.; Zeigler, B. "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.

[Sha99] Sharma, G. D.; Abu-Ghazaleh, N. B.; Rajasekaran, U. K. V.; Wilsey, P. A. "Optimizing Message Delivery in Asynchronous Distributed Applications". Proceedings of the 5th International Euro-Par Conference on Parallel Processing. Lecture Notes In Computer Science. Vol. 1685, pp. 1204-1208. 1999.

[Sri95] Srinivasan, S.; Reynolds, P.; "Adaptive Algorithms vs. Time Warp: An Analytical Compariosn". Proceedings of the 1995 Winter Simulation Conference. 1995.

[Sri98] Srinivasan, S.; Reynolds, J., “Elastic Time”, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 2. 103-139. April 1998.

[Szu00] Szulsztein, E.; Wainer, G. “New Simulation Techniques in WARPED Kernel” (in Spanish). Proceedings of JAIIO, Buenos Aires, Argentina, 2000.

[Tro01] Troccoli, A.; Wainer, G. "CD++, a tool for simulating Parallel DEVS and Parallel Cell-DEVS models". Técnica Reporta. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. Argentina. 2001.

[Tro03] Troccoli, A.; Wainer, G. "Implementing Parallel Cell-DEVS". Proceedings of the 36th Annual Simulation Symposium (ANSS'03). IEEE. 2003.

[Uhr01b] Uhrmacher, A. M.; Kullick, B. G. "Interacting multi-agent and simulation systems - an exploration into Mole and James". Proceedings of the 5th International Conference on Autonomous agents. pp. 122-123. 2001.

[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.

[Wai99] Wainer, G.; Giambiasi, N. "Avoiding serialization in Timed Cell-DEVS". Proceedings of the 31st SCS Summer Computer Simulation Conference. Chicago. USA. 1999.

[Wai00] Wainer, G. "Improved cellular models with Parallel Cell-DEVS". Transactions of the Society for Computer Simulation International. Vol. 17, No. 2, pp. 73-88. 2000.

[Wai01a] Wainer, G.; Christen G.; Dobniewski, A. "Defining models with the CD++ toolkit". Proceedings of the European Simulation Symposium. Marseille, France. SCS Publisher. 2001.

[Wai01b] Wainer, G.; Giambiasi, N. "Timed Cell-DEVS: modeling and simulation of cell spaces". Invited paper for the book Discrete Event Modeling & Simulation: Enabling Future Technologies. Springer-Verlag. 2001.

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

[Wol86] Wolfram, S. "Theory and applications of cellular automata". Vol. 1. Advances Series on Complex Systems. World Scientific. Singapore. 1986.

[War07] Warped: A Time Warp Simulation Kernel. *Warped Documentation for version 1.0.* Available via <www.ececs.uc.edu/~paw/warped/>. [Accessed July, 2007.]

[Zei76] Zeigler, B. "Theory of modeling and simulation". First Edition. Wiley. 1976.

[Zei93] Zeigler, B.; 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.; Kim, J. G. "DEVS-C++: A high performance modeling and simulation environment". The 29th Hawaii International Conference on System Sciences. 1996.

[Zei99a] Zeigler, B.; Kim, D.; Buckley, S. "Distributed supply chain simulation in a DEVS/CORBA execution environment". Proceedings of the 1999 Winter Simulation Conference. 1999.

[Zei99b] Zeigler, B.; Sarjoughian H. S. "Support for hierarchical modular component-based model construction in DEVS/HLA". Simulation Interoperability Workshop. 1999.

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

[Zha06] Zhang, M.; Zeigler, B.; Hammonds, P. "DEVS/RMI - An auto-adaptive and reconfigurable distributed simulation environment for engineering studies". DEVS Integrative M&S Symposium (DEVS'06). Huntsville, Alabama, USA. 2006.