APPROACH TO A SIMULATION VIRTUAL MACHINE: OBJECT ORIENTED IMPLEMENTATION OF CDEVS AND PDEVS

A Thesis
Submitted to the African University of Science and Technology
Abuja-Nigeria
in partial fulfilment of the requirement for

MASTER DEGREE IN COMPUTER SCIENCE

By
Aminu Musa Muhammad

Supervisor:

Professor Traore Kaba Mamadou

African University of Science and Technology
www.aust.edu.ng
P.M.B 681,Garki,Abuja F.C.T
Nigeria.

December 2009

APPROACH TO A SIMULATION VIRTUAL MACHINE:

OBJECT ORIENTED IMPLEMENTATION OF CDEVS AND PDEVS

BY

Aminu Musa Muhammad

RECOMMENDED: …….....**.....**

…….....

…….....

Committee Chair

APPROVED: …….....

Chief Academic Officer

.....

Date.

Abstract

In the world today, the size of complex problems and systems has increased, due to the advances in technology and the area of computer design and architecture. Modeling and predicting the behaviour of complex systems (Weather forecast, Fire spreading, River floods, Earthquake, Nanotechnology; design of new materials from molecule scale, decoding the human genome and many others) is becoming complex due to huge amount of data, high statistics need, and the need to serve user communities around the world. Therefore there is a need of exploiting the computing power of nowadays technologies by distributing simulation on multiple processors in order to reduce execution time, perform real time execution, and integrate simulators.

In this work we present an approach to Discrete Event System Specification (DEVS) virtual machine that will take a DEVS model and maps its simulation onto any hardware host like a LAN, a WAN, a Grid, a Cluster, the Internet and so on. The virtual machine is structured in 3 layers; the modeling layer which receives the DEVS model, the simulation layer which simulates the model using either the pessimistic synchronization algorithm or the optimistic synchronization algorithm, and the last layer which is the Middleware layer that allows the mapping of the simulation onto any hardware host.

The kernel of the virtual machine contains a CDEVS implementation of the simulator, PDEVS implementation of the simulator, and the distributed versions of them. Starting with an existing CDEVS simulator we got its PDEVS implementation using a meta modeling approach. And we finally provide the multilayer simulation package.

Dedication

To the almighty Allah, to whom I owe all my entirety and accomplishments.

To my parents who gave me all the support I needed.

Acknowledgements

All praise is due to Allah, and may His peace and blessings be upon the Prophet (peace be upon him) who said: “should not I be a grateful servant”.

It would not have been possible to write this thesis without the help and support of the kind people around me, I can only mention few here but to all of you I say thank you so much.

I owe my deepest gratitude to my supervisor professor Mamadou Kaba Traore, whose guidance, encouragement and support from the beginning till the end of this work enabled me to successfully complete this thesis.

I would like to thank Dr. Cisse Boubou, he has made available his support in a number of ways.

It is a pleasure to thank the IT department especially Mr. Dayo and Mr. Bobby, the chief librarian Ms Alice Olije, and the entire AUST community, thank you all for being kind, helpful and supportive throughout my stay in AUST.

The last but not in anyway the least, I would like to show my gratitude not to my roommate Padiani Kabelu directly but to his flash that I used from the beginning till the end of this thesis.

May Allah bless you all.

Table of Contents

Chapter 1

Introduction

1.1 Introduction to Computer Simulation

A Simulation is a computation that models the behaviour of some real or imagined system over time. And it is referred to as Computer Simulation when the computation is done on a computer. In the world today simulations are widely used to analyse the behaviour of systems such as fire spread, weather forecast, air traffic control, decoding the human genome and the design of new telecommunication networks without physically constructing the system in cases where constructing a prototype may be costly or even infeasible. But modeling and predicting the behaviour of complex systems is becoming complex too, due to huge amount of data, high statistics need, and the need to serve user communities around the world. Therefore there is a need of exploiting the computing power of nowadays technologies by distributing simulation on multiple processors in order to reduce execution time, perform real time execution, and integrate simulators. To see more about parallel and distributed simulation see the book (Fujimoto 2000).

"The study of any physical system to be simulated begins with the creation of a model. Such a model can be in one of several types: 1) Conceptual, 2) Declarative, 3) Functional, 4) Constraints, 5) Spatial or 6) Multi model." (Fishwick and Lin 1996). The conceptual model describes qualitative terms and class hierarchies for the system. In many ways the conceptual model organizes the definition of attributes, methods and general characteristics of each system components without going so far to ascribe dynamics to components. The next four model types reflect an orientation to system construction; a system may be constructed as Petri net, Queuing model or as cellular automaton for instance. The last model type (Multi model) permits the integration of basic model types to create a model composed of component models where each component model represents a level of abstraction for the system (Fishwick and Lin 1996).

After building a model from the real system or imagined system, aspects relevant to simulation are retained and irrelevant aspects are discarded then a simulation model is constructed that can be executed on a computer.

DEVS abbreviating Discrete Event System Specification is a modular and hierarchical formalism for modeling and analyzing general systems that can be discrete event systems which might be described by state transition tables, and continuous state systems which might be described by differential equations and hybrid continuous state and discrete event systems (Wikipedia 2007). We will talk more about DEVS in the next chapter.

1.2 Objective

Our main objective is to present a DEVS virtual machine that will take a DEVS model and maps its simulation onto any hardware host like a LAN, a WAN, a Grid, a Cluster, the Internet and so on. The virtual machine is structured in 3 layers; the modeling layer which receives the DEVS model, the simulation layer which simulates the model using either the pessimistic synchronization algorithm or the optimistic synchronization algorithm, and the last layer which is the Middleware layer that allows the mapping of the simulation onto any hardware host.

The kernel of the virtual machine will contain a CDEVS implementation of the simulator, PDEVS implementation of the simulator, and the distributed versions of them.

1.3 Work Done

Starting with an existing CDEVS simulator we got its PDEVS implementation using a meta modeling approach. The two simulation packages for the CDEVS and the PDEVS are available, but the complete package for the virtual machine is not available being it a research prototype. Both the CDEVS and the PDEVS simulators are implemented using Java, and JVM is used as our virtual machine in this case. We tested the simulators with an example, and make a comparison between the duration of the simulation in each simulator.

1.4 Organisation

Chapter 1 is the introductory chapter. The CDEVS and PDEVS formalisms are described in chapter 2. And their modeling using Unified Modeling Language (UML) is presented in chapter 3, where we show the static and the dynamic view of the simulators. In Chapter 4 we present how we translate the Model into Java code, the packages and classes involved. An example is considered in Chapter 5, where we compare the simulation time between the two simulators. Chapter 6 is the conclusion chapter.

Chapter 2

Discrete Event System Specification (DEVS)

2.1 Discrete Event Simulation

In discrete event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system. There are three common types of world views (simulation strategies) used in discrete event simulation languages – event scheduling, activity scanning, and process interaction (the combination of the first two). From the user's point of view, the specifics of the underlying simulation method are generally hidden.

Discrete event simulations include the following components:

- **Clock:** The simulation must keep track of the current simulation time, in any suitable measurement units for the system being modeled. Here because because events are instantaneous the clock skips to the next event start time as the simulation proceeds.
- **Event List:** An event is described by the time at which it occurs and a type, indicating the code that will be used to simulate that event. Therefore the simulation maintains at least one list of simulation events, sometimes called the pending event set because it lists events that are pending as a result of previously simulated event but have yet to be simulated themselves. The pending event set is typically organized as a priority queue, sorted by event time (Douglas 1986). That is, regardless of the order in which events are added to the event set, they are removed in strictly chronological order.
- **Random Number Generators:** Depending on the system model, the simulation needs to generate random variables of various kinds.
- **Statistics:** Also depending on the system model there are some aspects of interest that needs to be quantified. Therefore the simulation typically keeps track of the system's statistics.
- **Ending Condition:** Theoretically a discrete-event simulation could run forever. So the simulation designer must decide when the simulation will end. Typical choices are “at time t” or “after processing n number of events” or, more generally, “when statistical measure X reaches the value x”.

2.2 Discrete Event System Specification (DEVS)

DEVS abbreviating Discrete Event System Specification is a modular and hierarchical formalism for modeling and analyzing general systems that can be discrete event systems which might be described by state transition tables, and continuous state systems which might be described by differential equations, and hybrid continuous state and discrete event systems (Wikipedia 2007).

DEVS formalism was invented by Dr. Bernard P. Zeigler, and was introduced to the public in his first book *Theory of Modeling and Simulation* in 1976 when he was an associate professor at University of Michigan. DEVS can be seen as an extension of Moore machine which is a finite state automaton where the outputs are determined by the current state alone (and do not depend directly on the input). The extension was done by

1. associating the lifespan with each state (Zeigler 1976).
2. providing a hierarchical concept with the coupling operation (Zeigler 1984).

Many extended formalism from DEVS have been introduced with their own purposes, after Zeigler proposed a hierarchical simulation algorithm for DEVS model simulation in 1984. Throughput this report we will be using the term CDEVS or Classic DEVS for the Zeigler's formalisms. And DESS short for Discrete Event System. Some of extensions include: DESS(Discrete Event System)/DEVS for combined continuous and discrete event systems, P-DEVS for parallel DESs, G-DEVS for piecewise linear state trajectory modeling of DESS, RT-DEVS for real-time DESS, Cell-DEVS for cellular DESS, Fuzzy-DEVS for fuzzy DESS, Dynamic Structuring DEVS for DESS changing their coupling structures dynamically, and so on.

We used (Zeigler, Herbert, and Tag, 2000) for most of the things followed in this chapter.

2.3 Classic DEVS (CDEVS)

DEVS defines system behaviour as well as system structure. System behaviour in DEVS formalism is described using input and output events as well as states.

In the classic DEVS formalism, Atomic DEVS captures the system behavior, while Coupled DEVS describes the structure of system.

2.3.1 CDEVS Atomic Model

An atomic DEVS model is defined as a 7-tuple

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

where

- X is the set of input values
- Y is the set output values
- S is the set of states (or also called the set of partial states)
- ta: $S \rightarrow R^{+}_{0,\infty}$ is the set of positive reals with 0 and ∞.

- δ_{ext} : Q x X → S is the external transition function, where
  Q = {(s, e) | s Є S, 0 ≤ e ≤ ta(s) is the total state set
  e is the time elapsed since last transition
- δ_{int} : S → S is the internal transition function
- λ: S → Y is the output function

The interpretation of these elements is illustrated in Fig. 2.1.

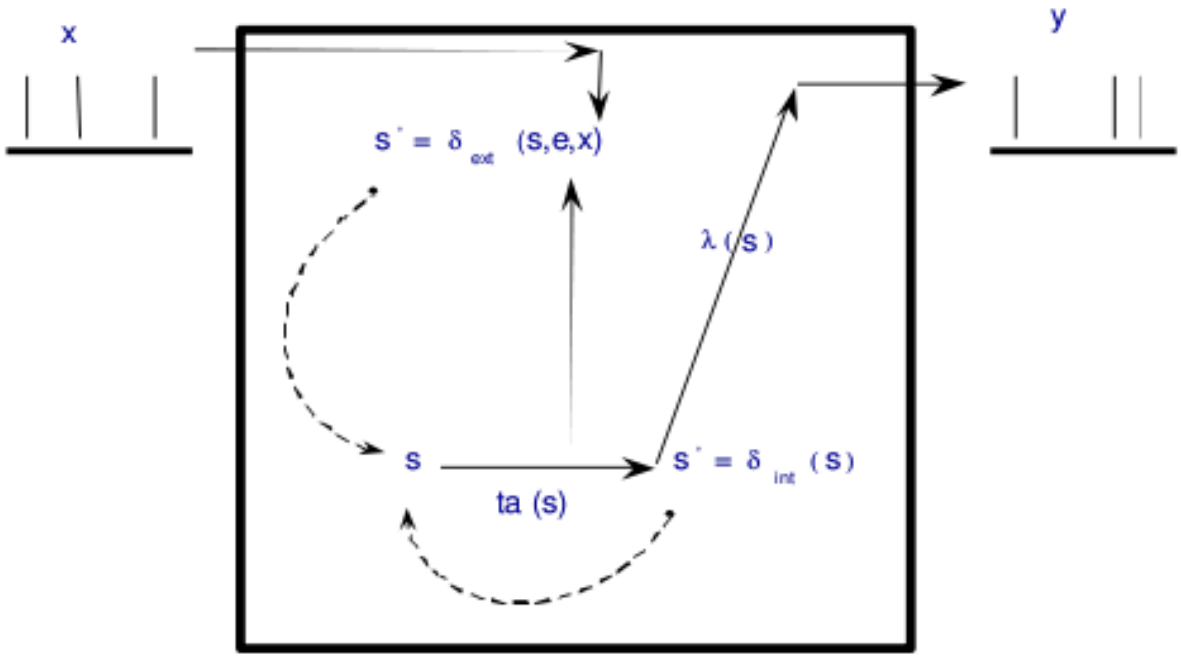


Fig 2.1: DEVS in action.

At any time the system is in some state, s. If no external event occurs, the system will stay in state s for time ta(s). In the first case the stay in s is so short that no external event can intervene- this means s is a transitory state. In the second case, the system will stay in s forever unless an external event interrupts its slumber- s is said to be a passive state. When the elapsed time expires, e = ta(s), the system outputs value, λ(s), and changes state δ_{int} (s). Note that output is only possible just before internal transitions.

If an external event x Є X occurs when the system is in total state (s, e) with e ≤ ta(s),

the system changes to state δ_{ext} (s, e, x). Thus the internal transition function dictates the system's new state when no events occurred since the last transition. The external transition function dictates the system's new state when an external events occurs. In both cases, the system is then in some new state s' with some new resting time, ta(s'), and it continues like that.

The pseudo code algorithm of the simulator that would execute, and generate the behaviour of the semantics of the DEVS model described above is given below and its corresponding flowchart is shown in Fig 2.2.

- Every atomic model has a simulator assigned to it which keeps track of the time of the last event, tL and the time of the next event, tN
- Initially, the state of the model is initialized as specified by the modeler to a desired initial state, s_{init}. The event times, tL and tN are set to 0 and ta(s_{init}), respectively.
- If there are no external events, the clock time, t is advanced to tN, the output is generated and the internal transition function of the model is executed. The simulator then updates the event times, and processing continues to the next cycle.
- If an external event is injected to the model at some time, t_{ext} (no earlier than the current clock and no later than tN), the clock is advanced to t_{ext}.
  - If t_{ext} == tN the output is generated.
  - Then the input is processed by external event transition function.

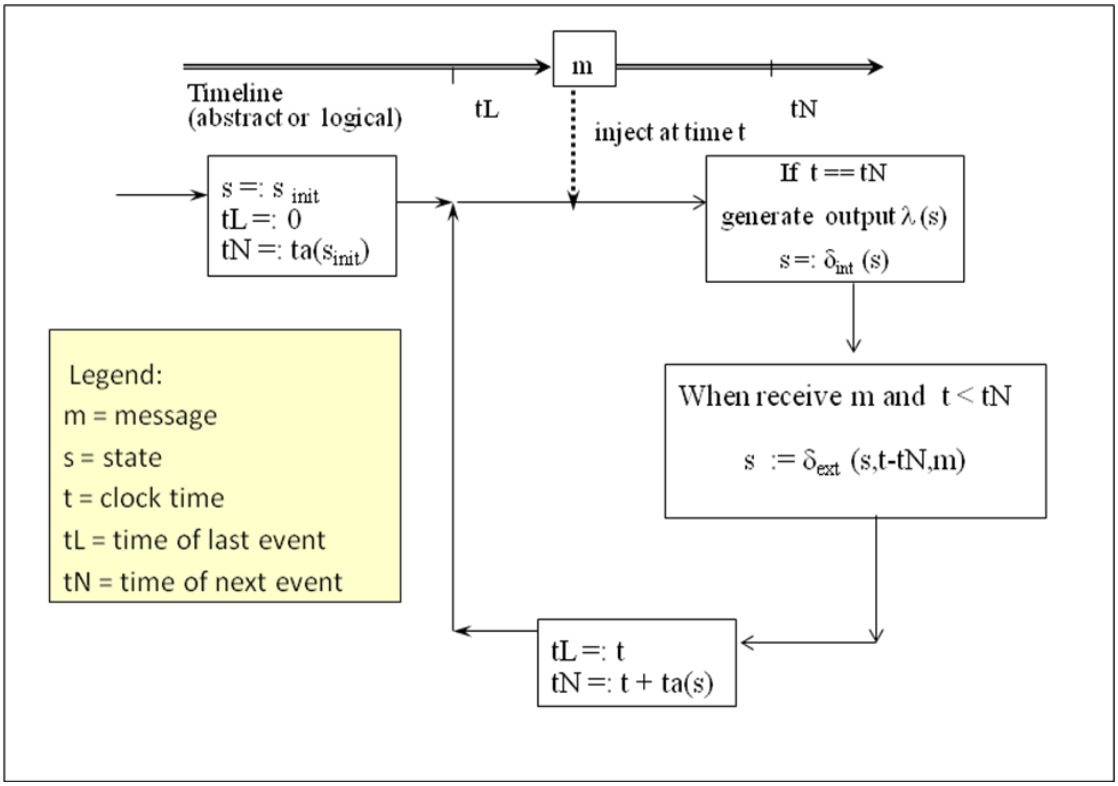


Fig 2.2: CDEVS Atomic Model Simulator Flowchart

2.3.2 CDEVS Coupled Model

The Coupled DEVS defines the components that belong to it and how they are connected with each other. A Coupled DEVS model is defined as a 8 tuple

$$N = (X, Y, D, \{ M_d \mid d \in D\}, EIC, EOC, IC, Select)$$

Where

- X is the set of input ports and values
- Y is the set of input ports and values
- D is the set of component names
- Components are CDEVS models, for d ∈ D, M_d is a sub-components. It can be either Atomic CDEVS model or Coupled CDEVS model
- EIC (External Input Coupling): connect external inputs to component inputs
- EOC (External Output Coupling): connect external outputs of components to external outputs
- IC (Internal Coupling): connect components outputs to component inputs
- Select : $2^D - \{\} \rightarrow D$, the tie-breaking function

The pseudo code algorithm of the simulator that would execute, and generate the behaviour of the semantics of the Coupled DEVS model described above is given below.

1. Coordinator sends nextTN to request tN from each of the simulators
2. All the simulators reply with their tNs in the outTN message to the coordinator
3. Coordinator sends to each simulator a getOut message containing the global tN (the minimum of the tNs)
4. Each simulator checks if it is imminent (its tN = global tN) and if so, returns the output of its model in a message to the coordinator in a sendOut message
5. Coordinator uses the coupling specification to distribute the outputs as accumulated messages back to the simulators in an applyDelt message to the simulators – for those simulators not receiving any input, the messages sent are empty
6. Each simulator reacts to the incoming message as follows:

- If it is imminent and its input message is empty, then it invokes its model's internal transition function
- If is not imminent and its input message is not empty, it invokes its model's external transition function
- If is not imminent and its input message is empty then nothing happens

For a coupled model with atomic model components, a coordinator is assigned to it and coupled Simulators are assigned to its components. In the basic DEVS Simulation Protocol, the coordinator is responsible for stepping simulators through the cycle of activities shown. See Fig 2.3

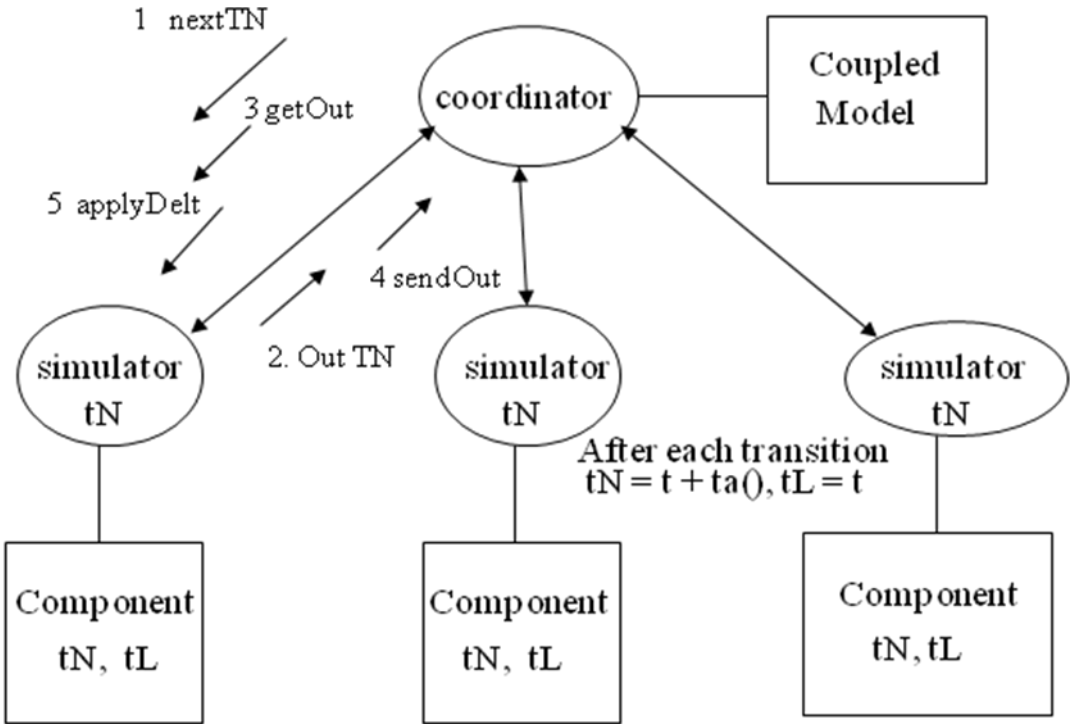


Fig 2.3: The Coupled Model Coordinator for CDEVS

2.4 Parallel DEVS (PDEVS)

About some 15 years after the Classic DEVS was introduced, a revision was introduced called the Parallel DEVS. The Parallel DEVS removes constraints that originated with the sequential operation of the early computers and hindered the exploitation of parallelism.

Parallel DEVS differs from classic DEVS in allowing all imminent components to be activated and to send their output to other components. The receiver is responsible for examining this output and properly interpreting it.

2.4.1 PDEVS Atomic Model

A basic Parallel DEVS is a structure,

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

where

- X is the set of input ports and values
- Y is the set output ports and values
- S is the set of sequential states
- ta: $S \rightarrow R^{+}_{0,\infty}$ is the set of positive reals with 0 and ∞. (time advance function)
- $\delta_{ext} : Q \times X^{b}_{M} \rightarrow S$ is the external transition function, where

  $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)$ is the total state set

  e is the time elapsed since last transition
- $\delta_{int} : S \rightarrow S$ is the internal transition function
- $\delta_{conf} : Q \times X^{b}_{M} \rightarrow S$ is the confluent transition function, where
- $\lambda: S \rightarrow Y$ is the output function

Note:

1. Instead of having a single input, here we have a bag of inputs. A bag is a set with possible multiple occurrences of its elements.
2. The addition of a transition function called confluent. It decides the next state in cases of collision between external and internal events.

The flowchart of the simulator that would execute, and generate the behaviour of the semantics of the PDEVS model described above is given below in Fig 2.4

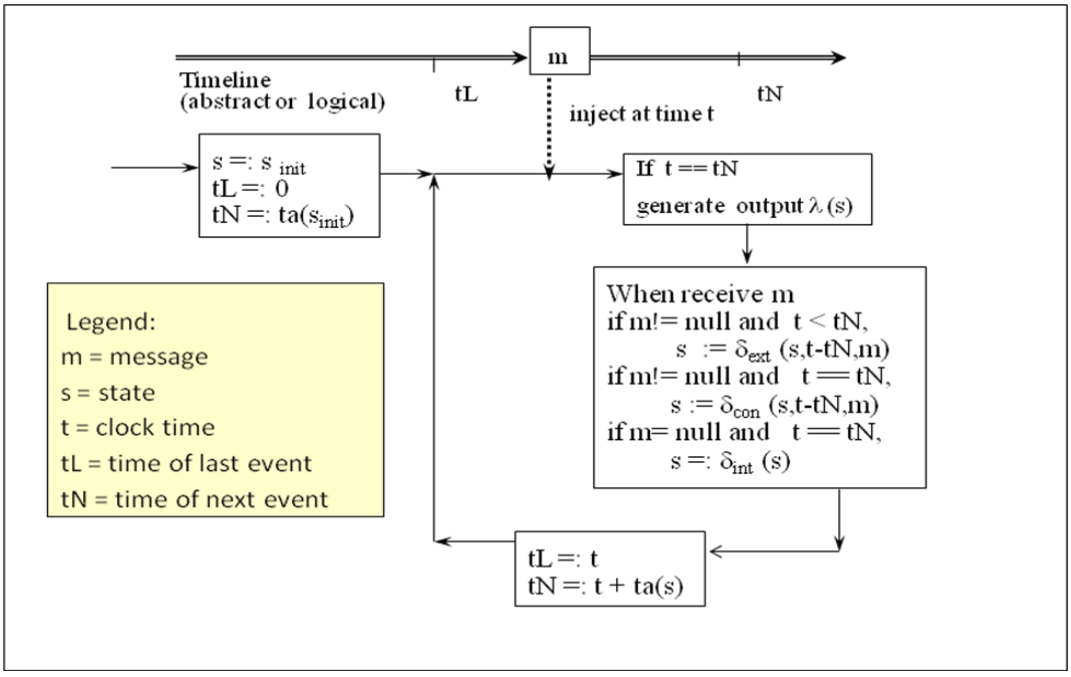


Fig 2.4: PDEVS Simulator Flowchart

2.4.2 PDEVS Coupled Model

The Parallel DEVS coupled models are described the same way as in Classic DEVS models except that the select function is removed. While this seems to be a simple change, its semantics differ significantly in how imminent components are handled. In PDEVS all imminent components generates their outputs which are distributed to their destinations using the coupling information.

A Coupled PDEVS model is defined as a 7 tuple

$$N = (X, Y, D, \{ M_d \mid d \in D\}, EIC, EOC, IC)$$

Where

- X is the set of input ports and values
- Y is the set of input ports and values
- D is the set of component names
- Components are PDEVS models, for $d \in D$, M_d is a sub-components. It can be either Atomic PDEVS model or Coupled PDEVS model
- EIC (External Input Coupling): connect external inputs to component inputs
- EOC (External Output Coupling): connect external outputs of components to external outputs
- IC (Internal Coupling): connect components outputs to component inputs

Chapter 3

Modeling the Simulation System

This section presents the modeling of both CDEVS and the PDEVS simulation systems, using the Unified Modeling Language (UML). The static view is shown by a class diagram and the dynamic view by a sequence diagram. We start with the CDEVS Paradigm.

3.1 The Static View of the CDEVS Simulator:

Two figures are shown here, Fig 3.1 which shows the Class Diagram of the CDEVS Model and the Class Diagram for the Simulator is shown in Fig 3.2.

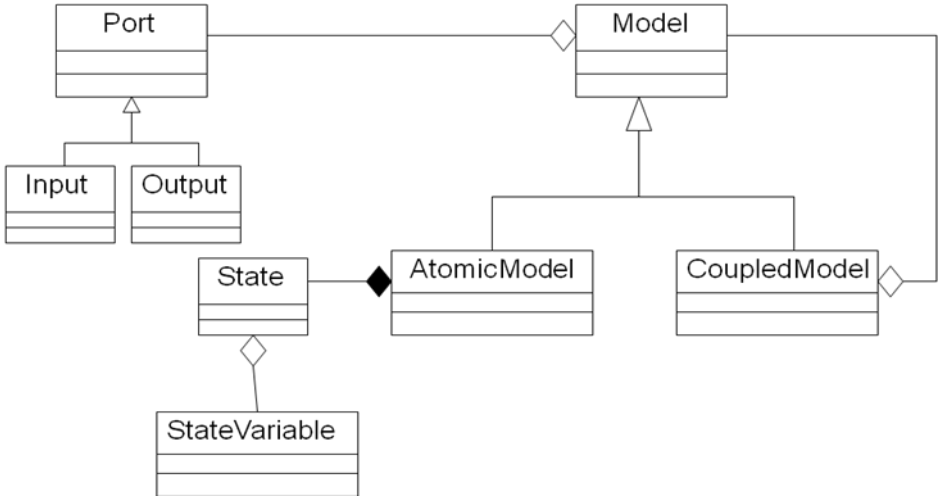
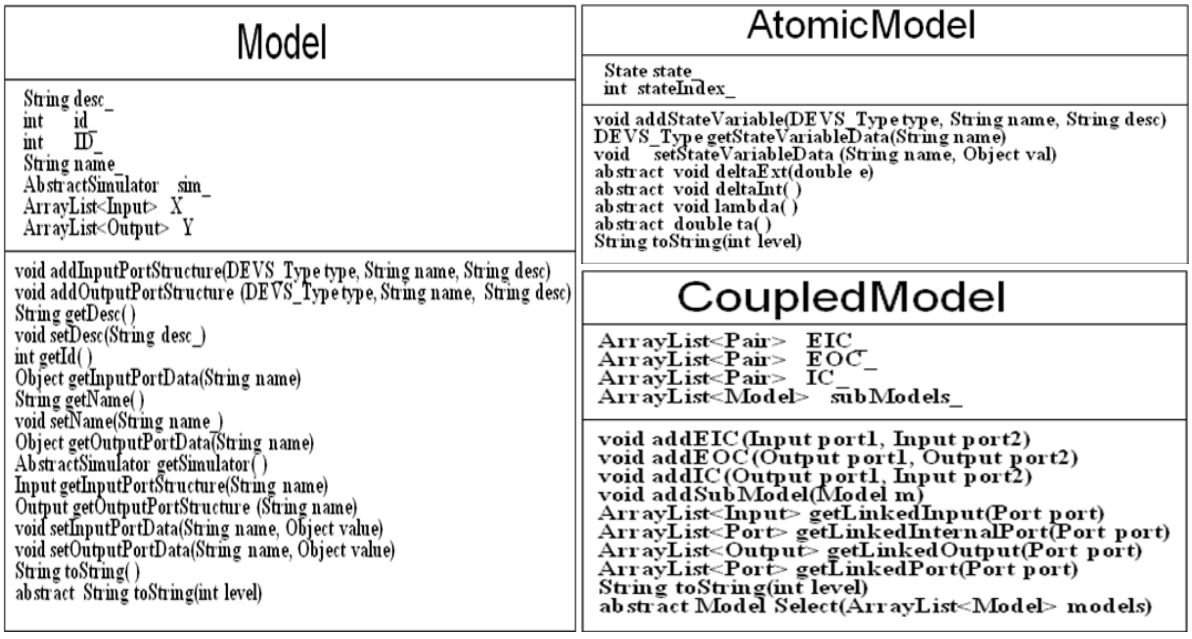
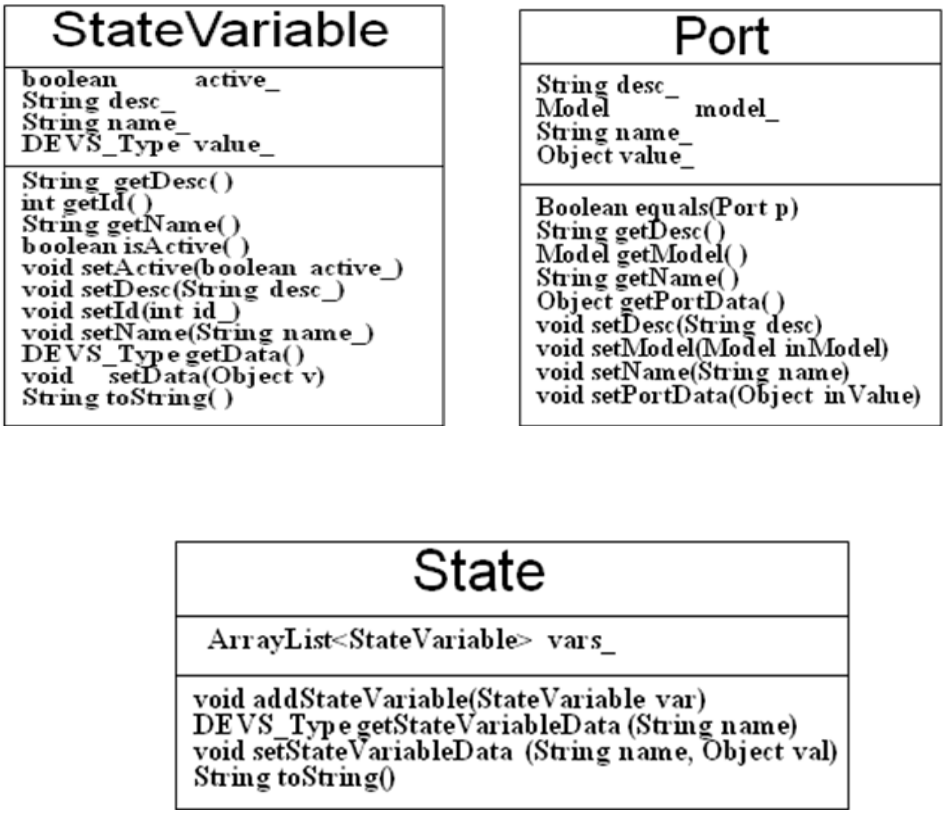


Fig 3.1: The Class Diagram for CDEVS Model





Remarks

Model:

- The ArrayLists X_ and Y_ store the list of Inputs and Outputs ports structure of the model respectively. And the AbstractSimulator sim_ is the simulator that simulates the model
- The two methods addInputPortStructure and addOutputPortStructure respectively add an input and output structures to the model, and there corresponding getter methods return the structures.
- To add data on a input port and output port the methods addInputPortData and addOutputPortData are used respectively and their corresponding getter methods are used to get the data.

AtomicModel:

- State_ and stateIndex_ keep track of the states of the atomic model. The abstract methods deltaInt, deltaExt, lamda, and ta are for the Modeler to implement in such a way that it suits his/her model.

CoupledModel:

- EIC_ stores the external input coupling, EOC_ stores the external output coupling, IC_ stores the internal coupling, and subModels_ is the list of sub

models in the coupled model. And their corresponding add methods add a port to the coupling list and a add model in case of subModels_.

- The getter methods take a port and return the list of ports linked (input, or output, or internal or the combination of the three) with the port.
- The select method is to be implemented by the modeler.

Port:

- A port can either be an input or output port, and it has a name, a value, description and a model. In addition to the setter and getter methods for the attributes listed above it also has a Boolean method "equals" that takes a port and return a Boolean value.

State:

- It has a list of state variables, a getter and setter methods for state variables, and an add method to add a state variable to the list.

StateVariable:

- It has the name, value, status, and description of a state variable and their corresponding setter and getter methods.

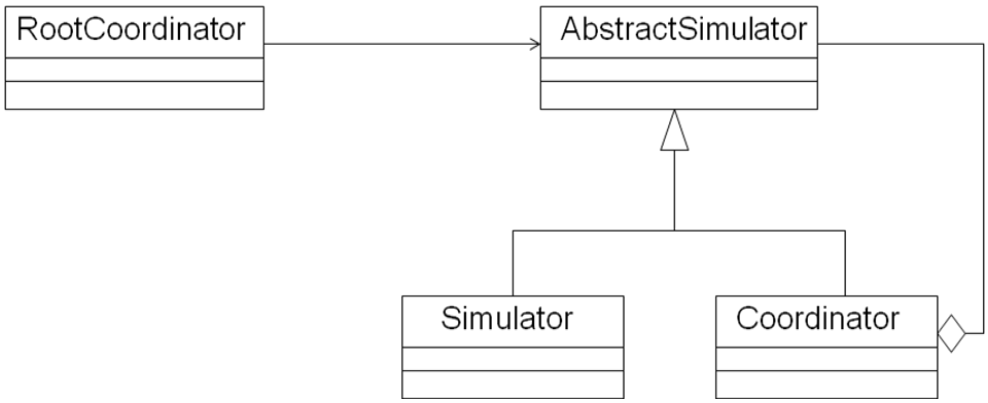
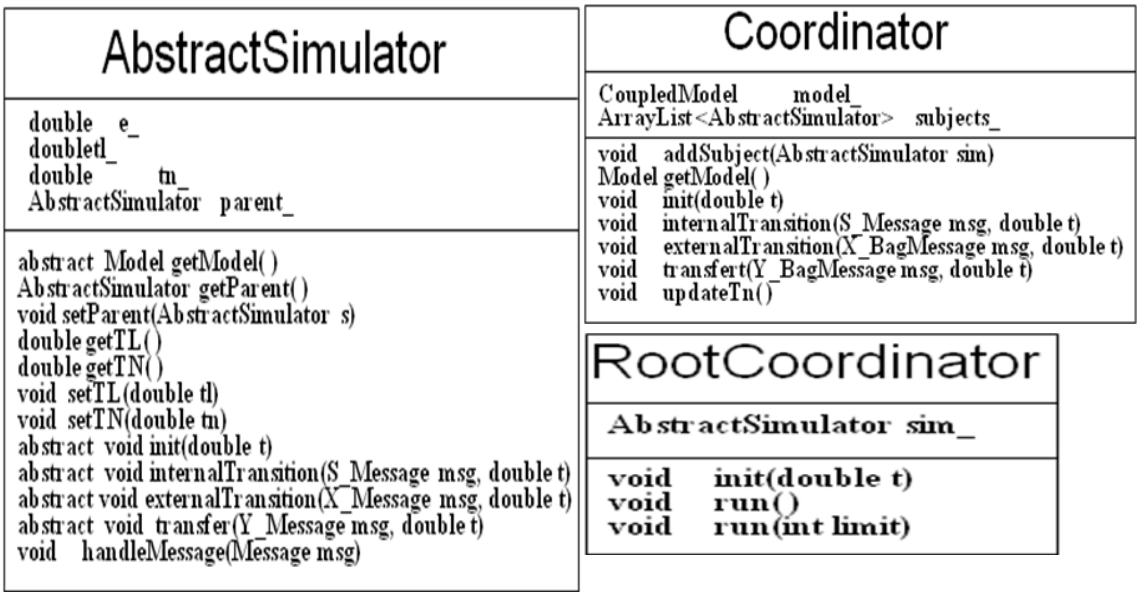
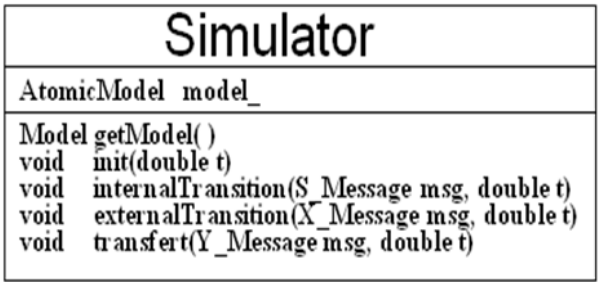


Fig 3.2: The Class Diagram for CDEVS Simulator





Remarks

AbstractSimulator:

- The variable tl_ stores the time of last event, tn_ stores the time of next event, and e_ stores the elapsed time. parent_ is the parent simulator to the abstract simulator.
- Each of the attributes listed above (tl_, tn_, and parent_) has a setter and getter methods.
- The method getModel returns the model the abstract simulator simulates. And handleMessage takes in a message (a message can be a *- message, i-message, x-message or y-message) and then call the appropriate method which can be an internalTransition, or externalTransition, or init, or transfer.

Coordinator:

- It has a coupled model that is simulated by the coordinator, and subjects_ is the list of children for the coordinator.
- The init method sends i-message to all children and updateTn update tn_ (time of next event).
- internalTransition, externalTransition and transfer send and receive messages based on the CDEVS simulator algorithm.

Simulator:

- The Simulator has model_ which is the atomic model it simulates.
- As in the case of the coordinator the internalTransition, externalTransition and transfer send and receive messages based on the CDEVS simulator algorithm.

RootCoordinator:

- The attribute sim_ is the abstract simulator that is to be managing by the root coordinator.
- The method init send i-message to the abstract simulator.
- There are two run methods to start the simulation one with limit and one without limit till simulation ends, this methods continue to send *-message to the abstract simulator.

3.2 The Dynamic View of the CDEVS Simulator

A sequence diagram is used to show the dynamic view of the CDEVS Simulator. See Fig 3.3.

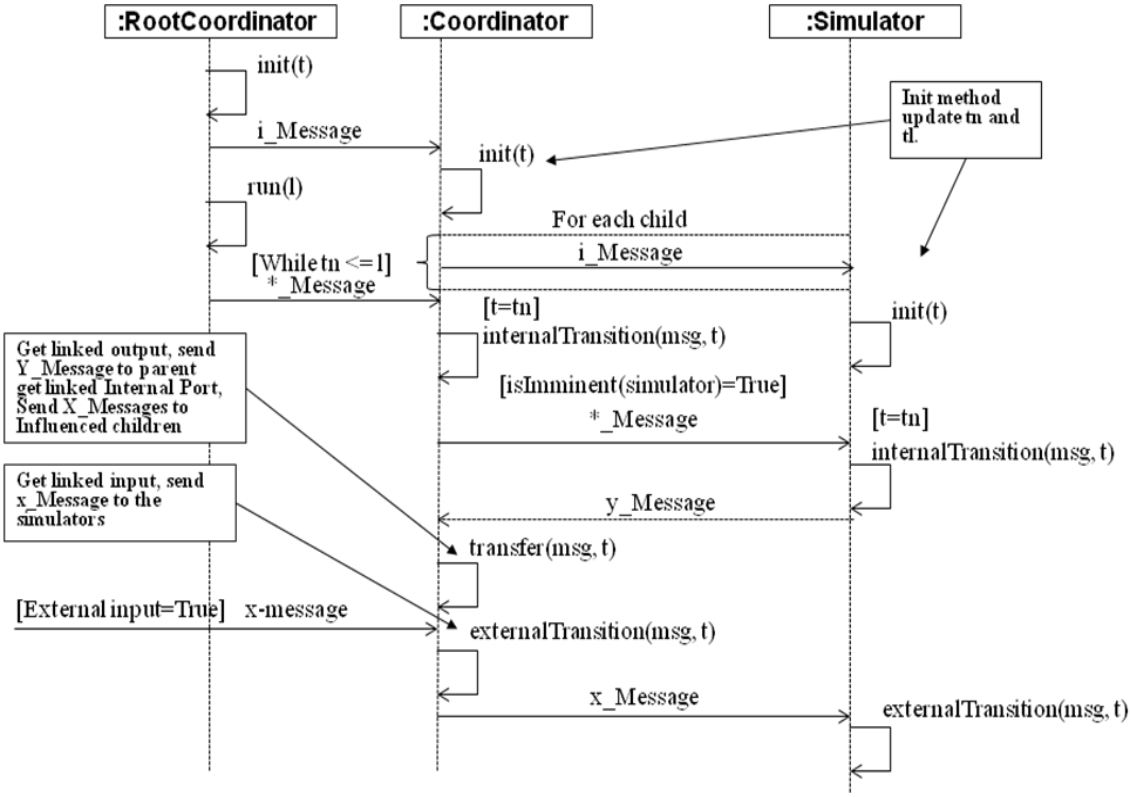


Fig 3.3: The Simulator Sequence Diagram for CDEVS

3.3 The Static View of the PDEVS Simulator

In the previous section we have seen the static and the dynamic view of the CDEVS Simulator. In this section we will now present the static view of the PDEVS Simulator.

As explained in the Introduction chapter, the PDEVS Simulator was built based on the existing CDEVS Simulator, therefore here we maintained the same static view shown in Fig 3.1 and Fig 3.2. But the existing methods and attributes were modified at the implementation stage, to meet the PDEVS Simulator algorithm and new ones are added. Below are some of the Methods added in the entities of the class diagram.

Model:
ArrayList<Input> getAllInputPorts ( ) : Returns the list of input ports of a model.
ArrayList<Output> getAllOutputPorts ( ) : Returns the list of output ports of a model.

Atomic Model:
abstract void deltaConf ( ): Allows the Modeler to implement confluent transition.

Coupled Model:
The method abstract Model Select (ArrayList<Model> models) is removed.

AbstractSimulator:
void addToInputBag (Port p) : Add a port to the input bag of the simulator.
ArrayList<Port> getInputBag ( ) : Returns the input bag of the simulator.
void handleBagMessage (BagMessage msg): Handle Bag Messages between simulators.

Coordinator:
void addToYparent (Port p): Add a port that is directed to the parent simulator to Yparent.
ArrayList<Port> getYparent ( ): Returns Yparent for the simulator.

3.4 The Dynamic View of the PDEVS Simulator

See Fig 3.4 for the sequence diagram of the PDEVS Simulator.

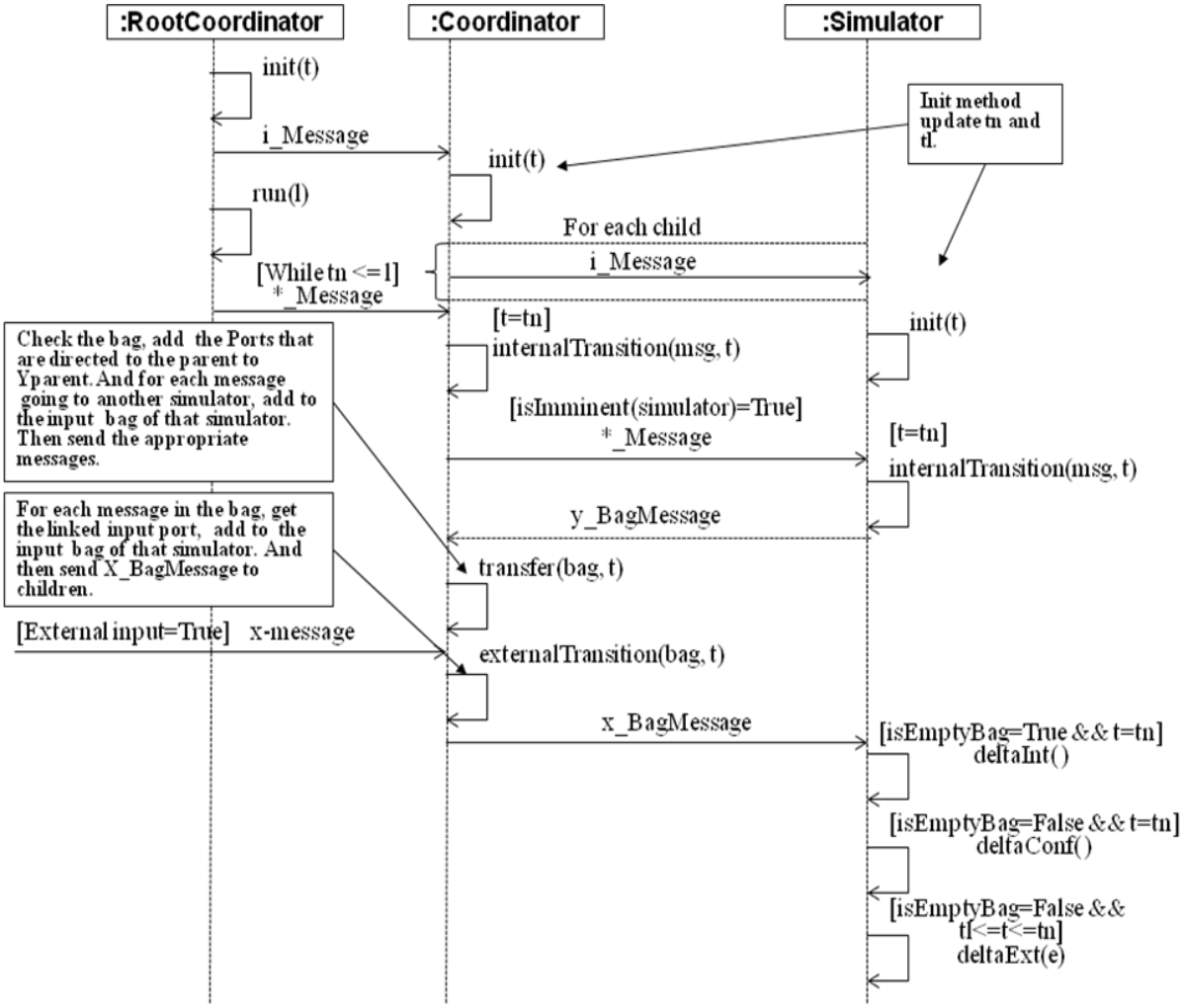


Fig 3.4: The Simulator Sequence Diagram for PDEVS
.

Chapter 4

Translation to Code

Both the CDEVS and the PDEVS Simulator Algorithms are implemented in Java. The CDEVS Simulator consists of 6 packages as shown below in Fig 4.1.

4.1 The CDEVS Simulation Algorithm Implementation

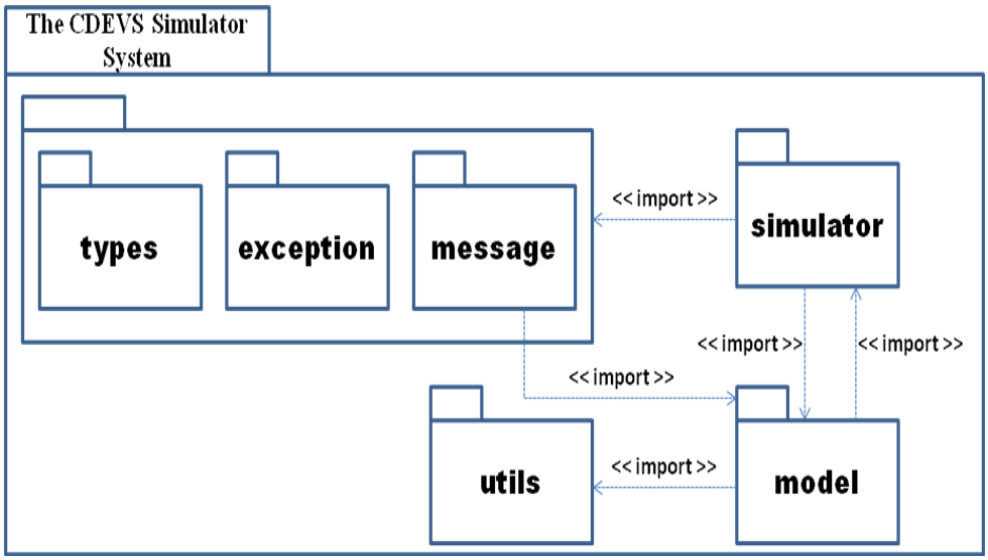


Fig 4.1: CDEVS UML Package Diagram

- **The exception package:**

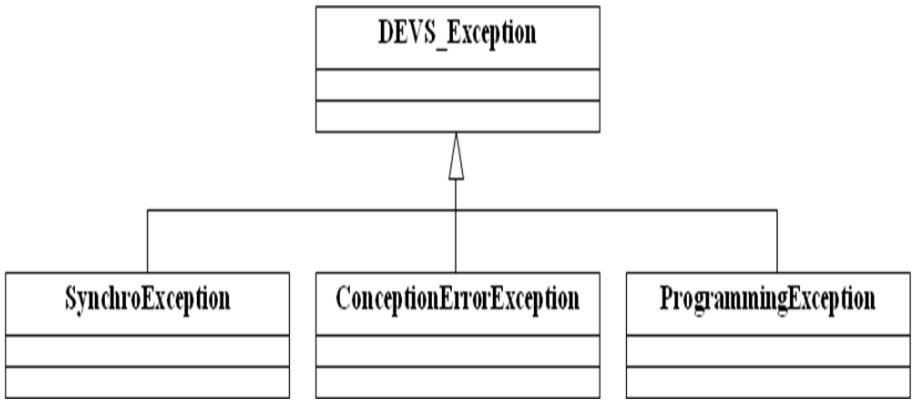


Fig 4.2: exception hierarchy

- **DEVS_Exception:** This class is a general exception. All the DEVS exception
  derive from this one. It can be used whenever the exception classes that have been defined cannot be used.

- **SynchroException:** This exception is used when there is synchronization problem in the communications.
- **ConceptionErrorException:** This class is used when model is not well constructed.
- **ProgrammingException:** This class should be used for debug purpose by the
  Developer who would continue the development of the library.

- **The message package:**

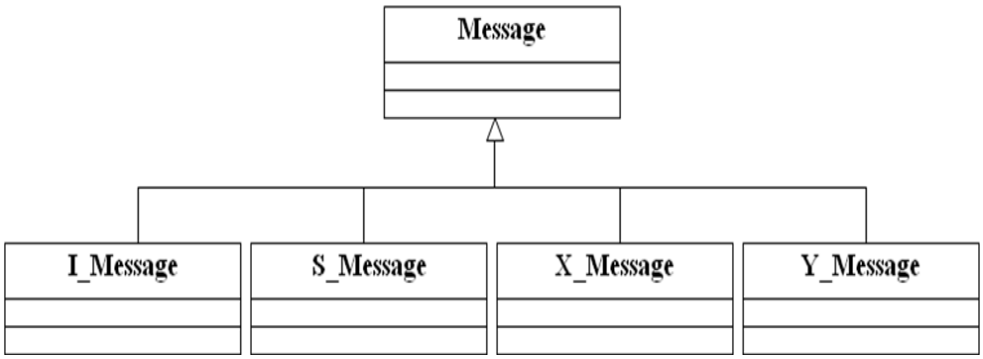


Fig 4.3: message hierarchy.

- **Message:** This is a standard class, used to define common specifications of the messages. All the other classes in the package just call the constructor of this class.

- **The model package:**
  For the structure of the package model see Fig 3.1 and the remarks below it, in the previous chapter.

- **The simulator package:**
  Similar to the model package, for the simulator package see Fig 3.2 and the remarks below it, in the previous chapter.

- **The types package:**

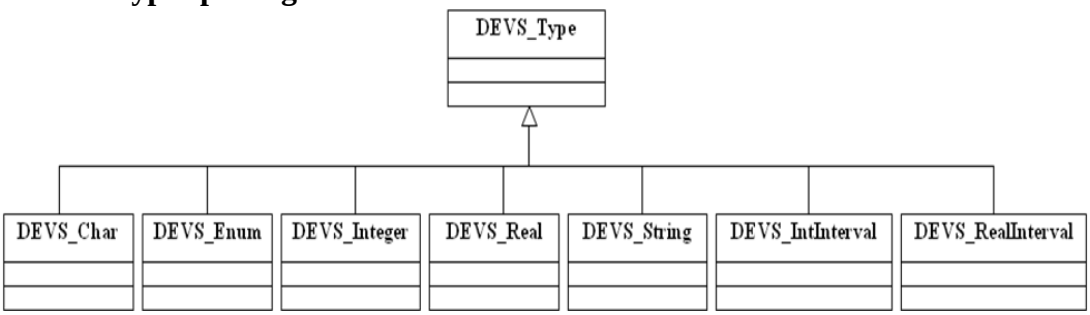


Fig 4.4: types hierarchy.

- **The utils package:**

  The package utils consists of two classes, Pair and Debug.
  Pair Class: This class defines a data structure pair, which takes in two objects and forms a pair of objects.
  Debug Class: This class is used for debug purpose.

4.2 The PDEVS Simulation Algorithm Implementation

The packages used in the CDEVS Simulator are maintained for the PDEVS Simulator, with the addition of 3 classes in the package message. The PDEVS structure of the message package is shown in Fig 4.5.

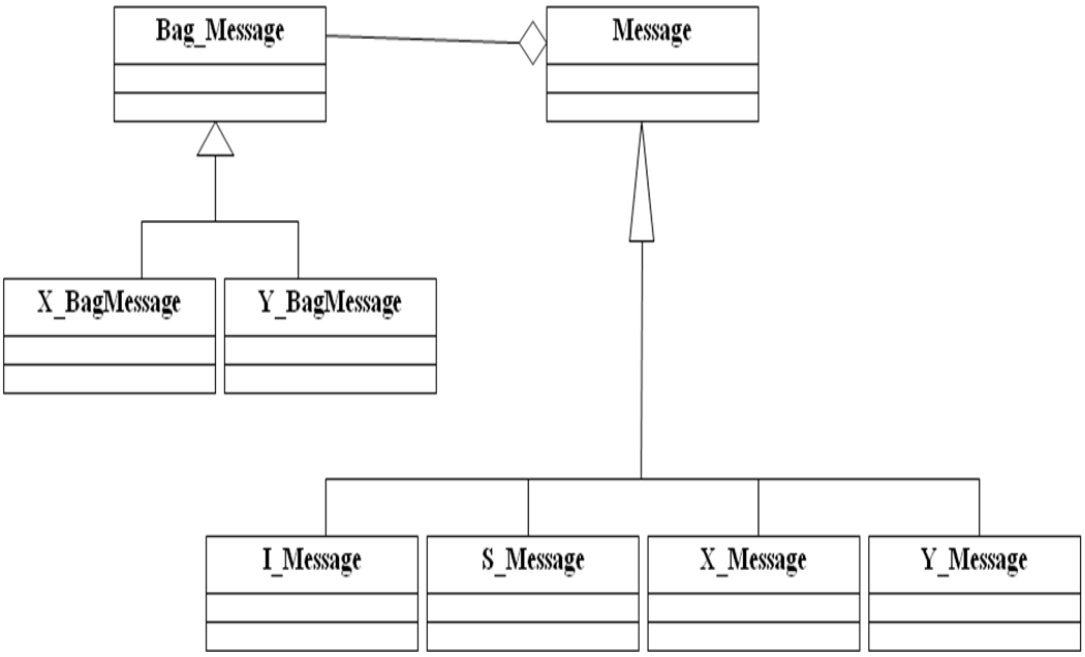


Figure 4.5: PDEVS message hierarchy.

- **Bag_Message:** This is similar to the class message except that instead of port it receives a bag which is an arraylist of ports. The two classes X_BagMessage and Y_BagMessage extend this class.
- **X_BagMessage:** This class just calls the constructor of the Bag_Message class.
- **Y_BagMessage:** Similar to the X_BagMessage class, this class also just calls the constructor of the Bag_Message class.

Chapter 5

Case Study and Comparison

5.1 Case Study: The Microwave Oven Example

We modelled a Microwave Oven using the DEVS Formalism and simulate the model using the CDEVS Simulator and then the PDEVS Simulator.

Consider the microwave structure specified by DEVS below:

Fig 5.1: Microwave Oven Picture.

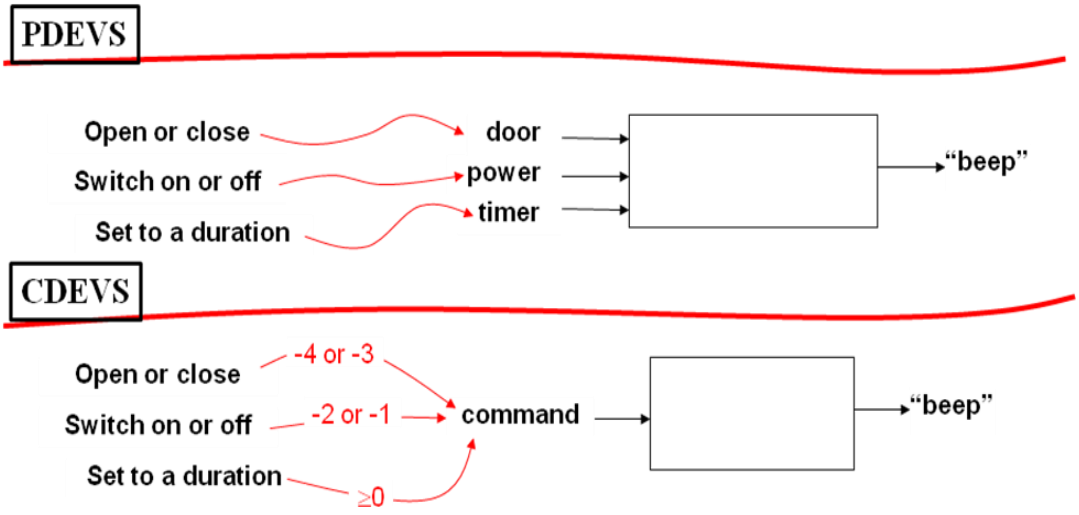
$S \perp \{(v, S_v), v \in State\}, dom(v) = S_v$

$\delta_{int} : \otimes S_v \rightarrow \otimes S_v$ internal transition function

$\delta_{ext} : \otimes S_v \times \Re^+ \times \otimes X_p \rightarrow \otimes S_v$ external transition function

$\lambda : \otimes S_v \rightarrow \otimes Y_q$ output function

$ta : \otimes S_v \rightarrow$ time advance function



5.2 CDEVS Microwave Oven Example

- **Specification**

$M_{Microwave} = \langle X, Y, T, S, \delta_{ext}, \delta_{int}, \lambda, t_a \rangle$

$X \perp \{(command, \{-4, -3, -2, -1\} \cup IR^+)\}$

$Y = \{beep\}$

$S \perp \{(power, \{on, off\}), (door, \{opened, closed\}), (timer, IR^+)\}$

$\delta_{ext}((u, v, w), e, -1) = (off, v, w), \forall (u, v, w) \in S, (u, v) \neq (on, closed), \forall e \in IR^+$

$\delta_{ext}((on, closed, w), e, -1) = (off, closed, w\text{-}e), \forall w \in IR^+, \forall e \in IR^+$

$\delta_{ext}((u, v, w), e, -2)) = (on, v, w), \forall (u, v, w) \in S, (u, v) \neq (on, closed), \forall e \in IR^+$

$\delta_{ext}((on, closed, w), e, -2) = (on, closed, w\text{-}e), \forall w \in IR^+, \forall e \in IR^+$

$\delta_{ext}((u, v, w), e, -4) = (u, opened, w), \forall (u, v, w) \in S, (u, v) \neq (on, closed), \forall e \in IR^+$

$\delta_{ext}((on, closed, w), e, -4) = (on, opened, w\text{-}e), \forall w \in IR^+, \forall e \in IR^+$

$\delta_{ext}((u, v, w), e, -3)) = (u, closed, w), \forall (u, v, w) \in S, (u, v) \neq (on, closed), \forall e \in IR^+$

$\delta_{ext}((on, closed, w), e, -3) = (on, closed, w\text{-}e), \forall w \in IR^+, \forall e \in IR^+$

$\delta_{ext}((u, v, w), e, command) = (u, v, x), \forall (u, v, w) \in S, \forall command \in IR+, \forall e \in IR+$

$\delta_{int}(on, closed, w) = (off, closed, +\infty)$

$\lambda(on, closed, w) = beep, \forall w \in IR^+$

$t_a(u, opened, w) = +\infty, \forall (u, w) \in \{on, off\} \times IN$

$t_a(off, v, w) = +\infty, \forall (v, w) \in \{opened, closed\} \times IN$

$t_a(on, closed, w) = w, \forall w \in IN$

5.2.1 Coupled Model

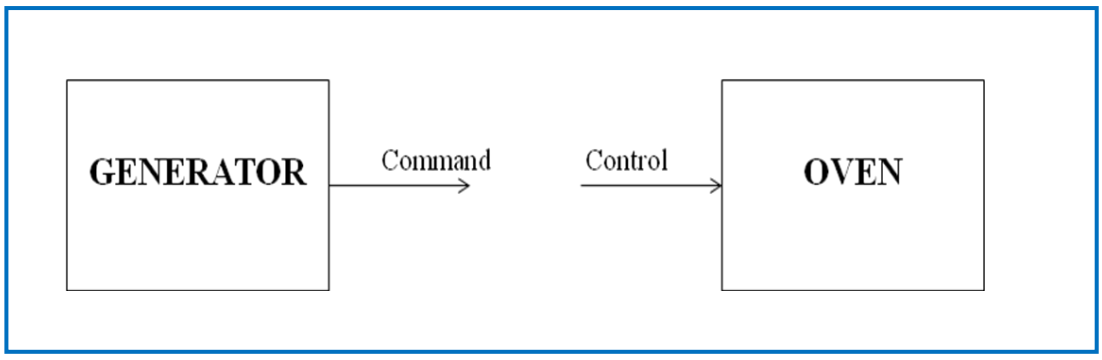


Fig 5.2: The coupled Model for the CDEVS Microwave Oven example

5.2.2 Atomic Models

- **Generator:**
  The Generator generates randomly a real number between -4 and 49 inclusive, the Java code for the Generator is shown below:

```
package microwave;

import model.*;
import types.*;
import exception.*;
import java.util.*;

public class Generator extends AtomicModel{

Random rand;

        public Generator( ) {
                super("Generator","Generates input between -4 and 49");
                addOutputPortStructure(new DEVS_Integer( ),"command", "generated
value");
                rand = new Random( );
        }

        public void deltaInt( ) {}

        public void deltaExt(double e) { }

        public void lambda( ) throws DEVS_Exception {
                int ctrl = -1*rand.nextInt(5);
                if(ctrl==0){
                        ctrl=rand.nextInt(50);
                }
                setOutputPortData("command", new DEVS_Integer(ctrl));
        }

        public double ta( ) {
                return 80*rand.nextDouble( );
        }

}
```

- **Oven:**
  The Oven receives a command from the Generator and do an external transition which may modify the state of the model and leads to an internal transition and also the displays to the user the current status of power, door, timer and the command received from the Generator. Whenever there is an internal transition, an output is displayed to the user showing “beep” meaning that the time set while the machine is on

and closed has elapsed (Food is ready). Below are the Java codes for the external transition, time advance and internal transition.

External Transition:

```
public void deltaExt(double e) throws DEVS_Exception {
        int control = ((DEVS_Integer)getInputPortData("control")).getInteger();
        ctrl=control;
        System.out.println("power:"+image(power));
        System.out.println("door:"+image(door));
        System.out.println("timer:"+timer);
        System.out.println("duration:"+duration);
        System.out.println("command:"+image(control));
        System.out.println("---------------------------------------");

        String out = Integer.toString(-power)+Integer.toString(-door)+Integer.toString(-
        ctrl)+Double.toString(timer);
        scr.setContent(out, getSimulator().getTL( ));

        switch(control){
        case -1:
                power = -1;
                if (state==0)
                        state=2;
                else if (state==1)
                        state=3;
                break;
        case -2:
                power = -2;
                if (state==2)
                        state=0;
                else if (state==3)
                        state=1;
                break;
        case -3:
                door = -3;
                if (state==1)
                        state=0;
                else if (state==3)
                        state=2;
                break;
        case -4:
                door = -4;
                if (state==0)
                        state=1;
                else if (state==2)
                        state=3;
                break;
        default:
                timer=control;
                break;
        }
}
```

Time Advance:

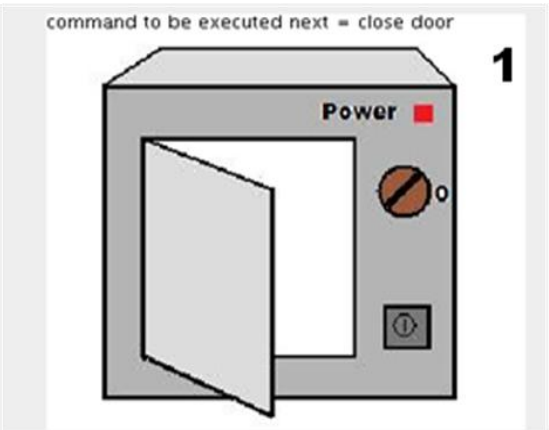
```
public double ta( ) {
        switch(state){
        case 0:
                duration = timer;
                break;
        default:
                duration = DEVS_Real.POSITIVE_INFINITY;
        }
        return duration;
}
```

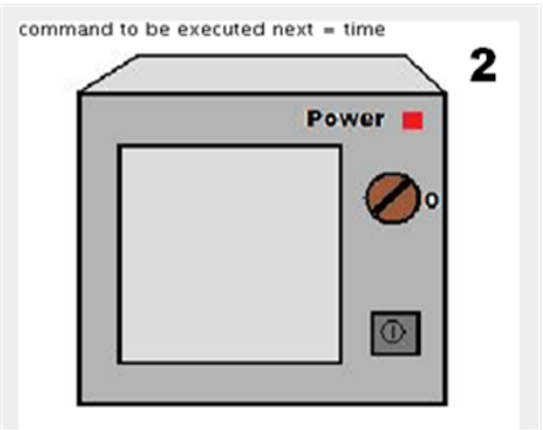
Internal Transition:

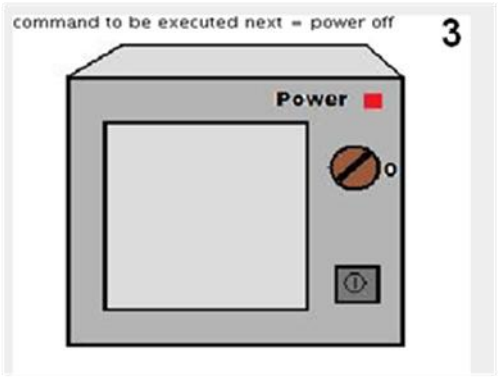
```
public void deltaInt ( ) {
                scr.setContent("000000", getSimulator( ).getTL( ));
                power=-1;
                timer=0.0;
                state=2;
                //duration = DEVS_Real.POSITIVE_INFINITY;
                System.out.println("");
                System.out.println("***** beep *****");
                System.out.println("");

        }
```

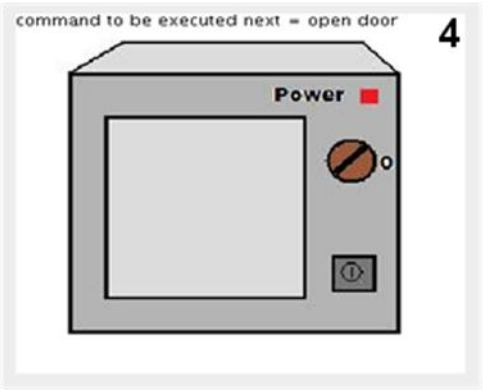
5.2.3 Simulation Result Snapshot

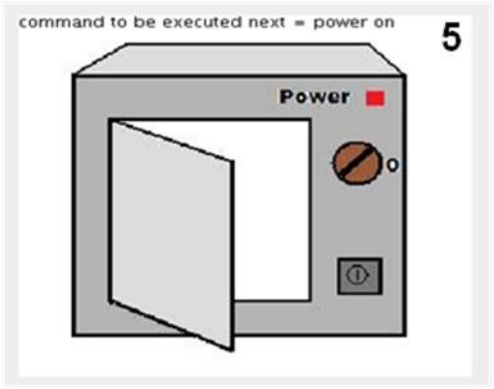
The microwave model specified and implemented above is simulated and below are some snapshots of the simulation result. Each snapshot shows the current status of the oven and the command that is to be executed at the time of next event.

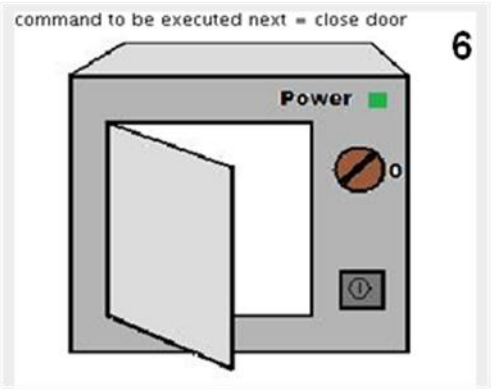


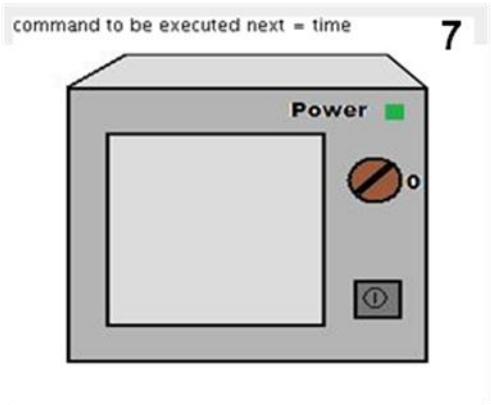



command to be executed next = power off
3
Power

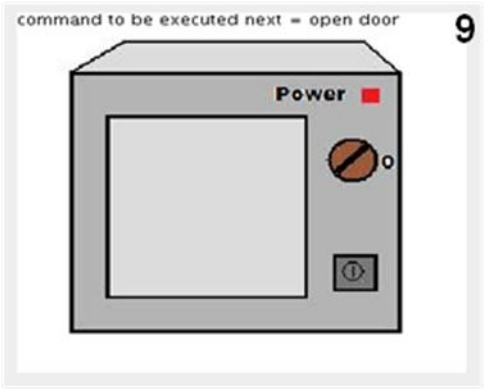

command to be executed next = open door
4
Power

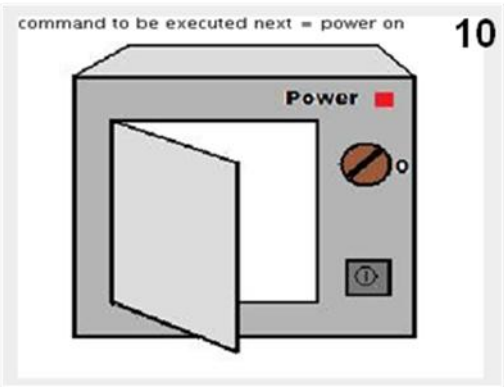

command to be executed next = power on
5
Power


command to be executed next = close door
6
Power


command to be executed next = time
7
Power


command to be executed next = time
8


command to be executed next = open door
9
Power


command to be executed next = power on
10
Power

5.2.4 Simulation Time

We run the Model 1000 times, at each time the time limit is set to 1000 unit time, we kept track of the duration of the simulation at each time. The average of the duration of the simulation was found to be **3.801802** unit time, and we plotted a graph shown below in Fig 5.3.

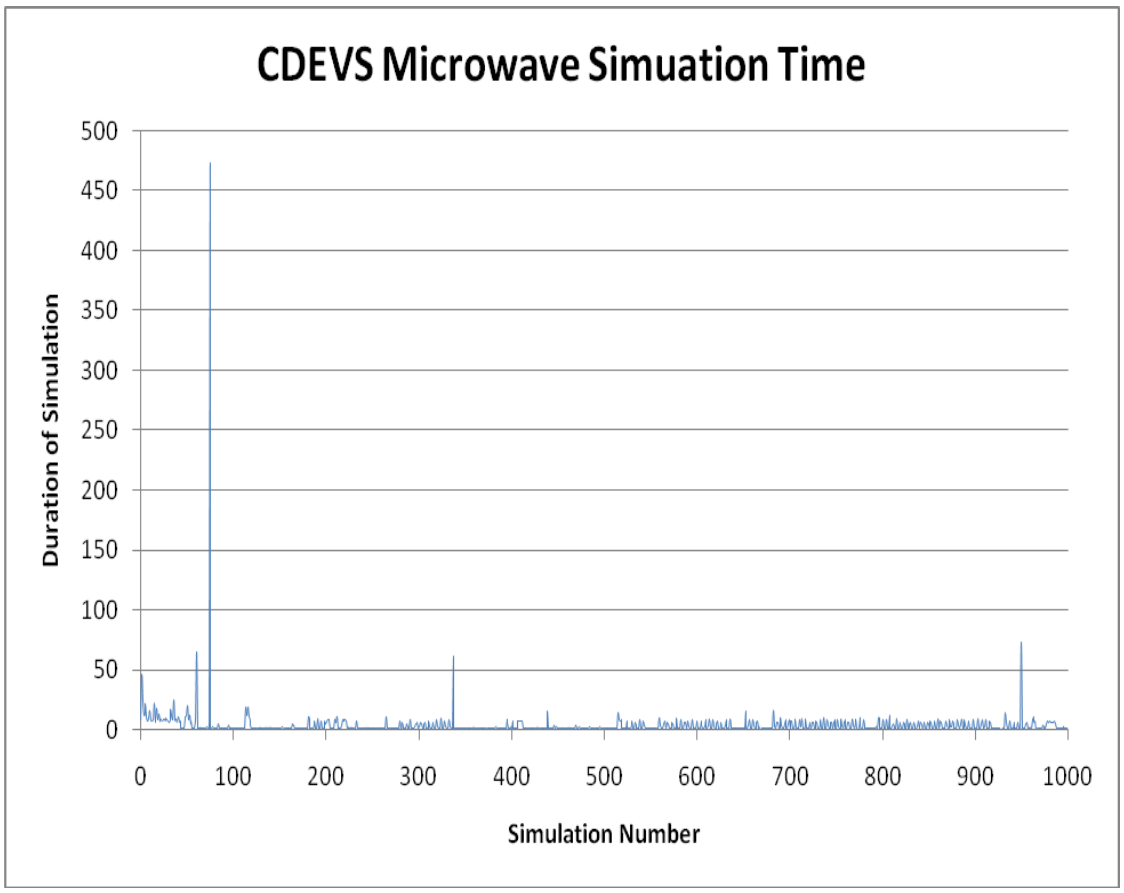


Fig 5.3: CDEVS Microwave Example Simulation Time

5.3 PDEVS Microwave Oven Example

5.3.1 Coupled Model

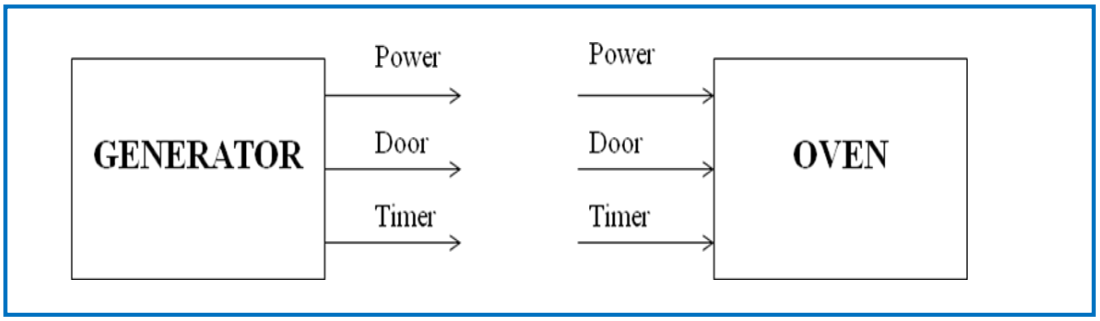


Fig 5.4: The coupled Model for the PDEVS Microwave Oven example

5.3.2 Atomic Models

- **Generator:**
  The Generator generates randomly a real numbers for the power, door, and the timer. The Java code for the Generator is shown below:

```
package microwave;
import model.*;
import types.*;
import exception.*;
import java.util.*;

public class Generator extends AtomicModel{
Random rand1,rand2,rand3;
/**
 * power command {0:null, 1:on, 2:off}
 * door command {0:null, 1:open, 2:close}
 * timer command {0-49: timer setting}
 */
        public Generator() {
                super("Generator","Generates input for the power, door and the timer");
                addOutputPortStructure(new DEVS_Integer(),"powerCommand",
                "generated value");
                addOutputPortStructure(new DEVS_Integer(),"doorCommand",
                "generated value");
                addOutputPortStructure(new DEVS_Integer(),"timerCommand",
                "generated value");
                rand1 = new Random();
                rand2 = new Random();
                rand3 = new Random();
        }
        public void deltaInt() {}

        public void deltaExt(double e) {}

        public void lambda() throws DEVS_Exception {
                int powerCtrl = rand1.nextInt(3);
                int doorCtrl = rand2.nextInt(3);
                int timerCtrl = rand3.nextInt(50);

                setOutputPortData("powerCommand", new DEVS_Integer(powerCtrl));
                setOutputPortData("doorCommand", new DEVS_Integer(doorCtrl));
                setOutputPortData("timerCommand", new DEVS_Integer(timerCtrl));
        }
        public double ta() {
                return 80*rand1.nextDouble();
        }
        public void deltaConf() throws DEVS_Exception {
                deltaInt();
                deltaExt(0);
        }
}
```

- **Oven:**
  Just the same way as the CDEVS implementation of the Microwave Oven, the Oven receives a command from the Generator and do an external transition which may modify the state of the model and leads to an internal transition and also the displays to the user the current status of power, door, timer and the command received from the Generator. Whenever there is an internal transition, an output is displayed to the user showing "beep" meaning that the time set while the machine is on and closed has elapsed (Food is ready). Below are the Java codes for the external transition, time advance and internal transition.

  External Transition:

```
public void deltaExt(double e) throws DEVS_Exception {
		int powerCtrl = 
((DEVS_Integer)getInputPortData("powerControl")).getInteger();
		int doorCtrl = 
((DEVS_Integer)getInputPortData("doorControl")).getInteger();
		int timerCtrl = 
((DEVS_Integer)getInputPortData("timerControl")).getInteger();

		String t="";
		if(timerCtrl==0)
			t="null";
		else
			t = Integer.toString(timerCtrl);

		if ((powerCtrl==0) && (doorCtrl==0) && (timerCtrl==0)) {
			ctrl = 0;
		}else if ((powerCtrl==0) && (doorCtrl==0) && (timerCtrl!=0)) {
			ctrl = 2;
		}else if ((powerCtrl==0) && (doorCtrl!=0) && (timerCtrl==0)){
			if (doorCtrl==1)
				ctrl = 3;
			else
				ctrl = 4;
		} else if ((powerCtrl==0) && (doorCtrl!=0) && (timerCtrl!=0)) {
			if (doorCtrl==1)
				ctrl = 5;
			else
				ctrl = 6;


		}else if ((powerCtrl!=0) && (doorCtrl==0) && (timerCtrl==0)) {
			if (powerCtrl==1)
				ctrl = 7;
			else
				ctrl = 8;
		}else if ((powerCtrl!=0) && (doorCtrl==0) && (timerCtrl!=0)) {
			if (powerCtrl==1)
				ctrl = 9;
			else
				ctrl = 10;
```

```
}else if ((powerCtrl!=0) && (doorCtrl!=0) && (timerCtrl==0)) {
        if (powerCtrl==1){
                if (doorCtrl==1)
                        ctrl = 11;
                else
                        ctrl =12;
        } else {
                if (doorCtrl==1)
                        ctrl = 13;
                else
                        ctrl =14;
        }
} else if ((powerCtrl!=0) && (doorCtrl!=0) && (timerCtrl!=0)){
        if (powerCtrl==1){
                if (doorCtrl==1)
                        ctrl = 15;
                else
                        ctrl =16;
        } else {
                if (doorCtrl==1)
                        ctrl = 17;
                else
                        ctrl =18;
        }
}
System.out.println("power:"+powerImage(power));
System.out.println("door:"+doorImage(door));
System.out.println("timer:"+timer);
System.out.println("duration:"+duration);
System.out.println("command:"+image(ctrl)+t+"}");
System.out.println("---------------------------------------");

String out =
Integer.toString(power)+Integer.toString(door)+Integer.toString
(ctrl)+Double.toString(timer);
scr.setContent(out, getSimulator( ).getTL( ));

switch(ctrl){

case 0:
        break;
case 2:
        timerCommand(timerCtrl, e);
        break;
case 3:
        openCommand(doorCtrl, e);
        break;
case 4:
        closeCommand(doorCtrl, e);
        break;
```

```
case 5:
        timerCommand(timerCtrl, e);
        openCommand(doorCtrl, e);
        break;
case 6:
        timerCommand(timerCtrl, e);
        closeCommand(doorCtrl, e);
        break;
case 7:
        onCommand(powerCtrl, e);
        break;
case 8:
        offCommand(powerCtrl, e);
        break;
case 9:
        timerCommand(timerCtrl, e);
        onCommand(powerCtrl, e);
        break;
case 10:
        timerCommand(timerCtrl, e);
        offCommand(powerCtrl, e);
        break;
case 11:
        onCommand(powerCtrl, e);
        openCommand(doorCtrl, e);
        break;
case 12:
        onCommand(powerCtrl, e);
        closeCommand(doorCtrl, e);
        break;
case 13:
        offCommand(powerCtrl, e);
        openCommand(doorCtrl, e);
        break;
case 14:
        closeCommand(doorCtrl, e);
        offCommand(powerCtrl, e);
        break;
case 15:
        timerCommand(timerCtrl, e);
        onCommand(powerCtrl, e);
        openCommand(doorCtrl, e);
        break;

case 16:
        timerCommand(timerCtrl, e);
        onCommand(powerCtrl, e);
        closeCommand(doorCtrl, e);
        break;
case 17:
        timerCommand(timerCtrl, e);
        offCommand(powerCtrl, e);
        openCommand(doorCtrl, e);
        break;
```

```
case 18:
        timerCommand(timerCtrl, e);
        closeCommand(doorCtrl, e);
        offCommand(powerCtrl, e);
        break;
}
}
```

Time Advance:

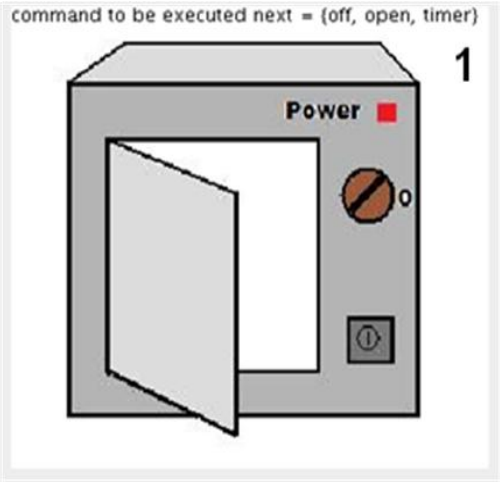
```
public double ta( ) {

        return duration;

}
```

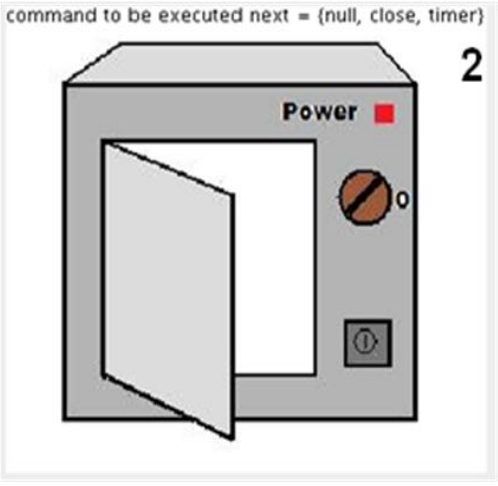
Internal Transition:

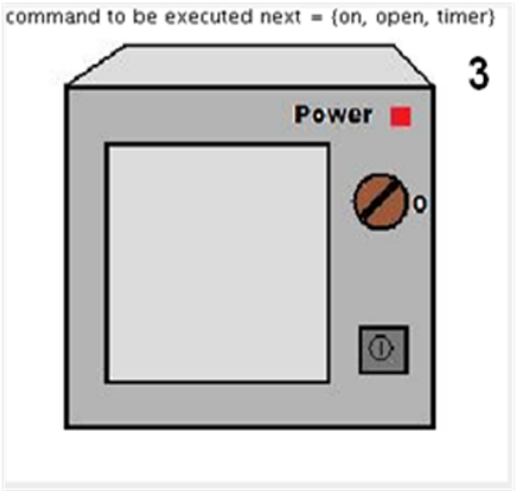
```
public void deltaInt( ) {
                scr.setContent("000000", getSimulator( ).getTL( ));
                power=2;
                timer=0.0;
                duration = DEVS_Real.POSITIVE_INFINITY;

}
```

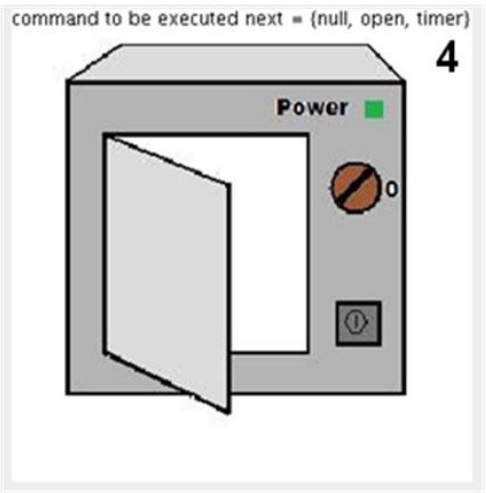
5.3.3 Simulation Result Snapshot

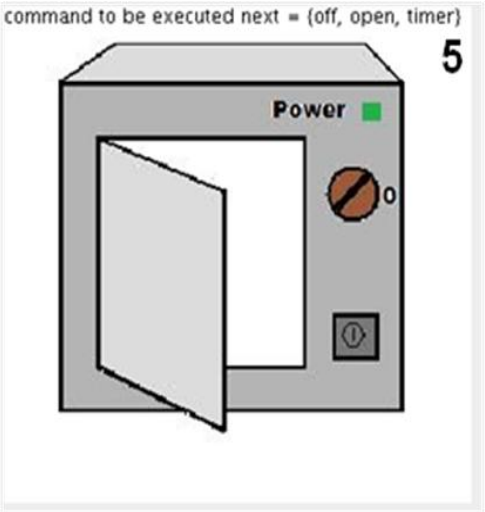
The microwave model specified and implemented above is simulated and below are some snapshots of the simulation result. Each snapshot shows the current status of the oven and the command that is to be executed at the time of next event.

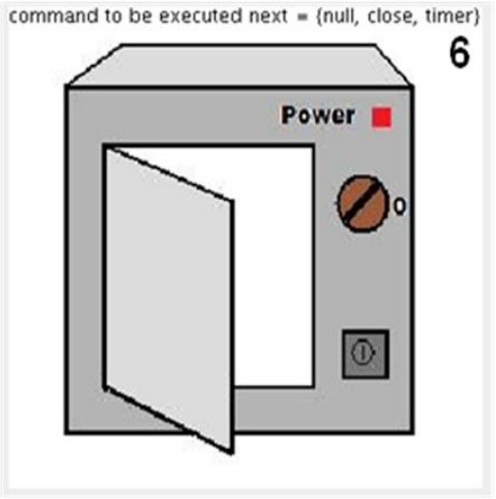


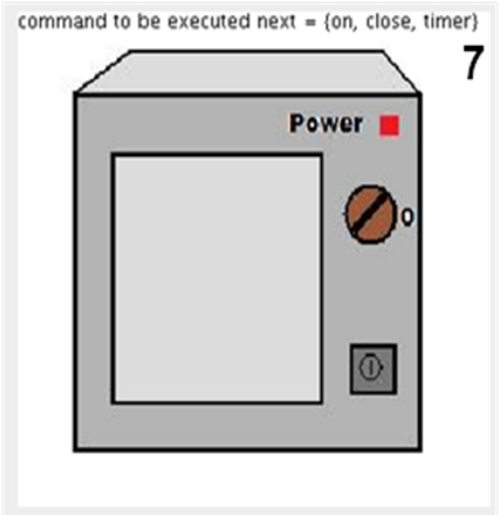



command to be executed next = {on, open, timer}
3
Power


command to be executed next = {null, open, timer}
4
Power


command to be executed next = {off, open, timer}
5
Power


command to be executed next = {null, close, timer}
6
Power


command to be executed next = {on, close, timer}
7
Power


command to be executed next = {null, null, null}
8

5.3.4 Simulation Time

We run the Model 1000 times, at each time the time limit is set to 1000 unit time, we kept track of the duration of the simulation at each time. The average of the duration of the simulation was found to be **3.749** unit time, and we plotted a graph shown below in Fig 5.5.

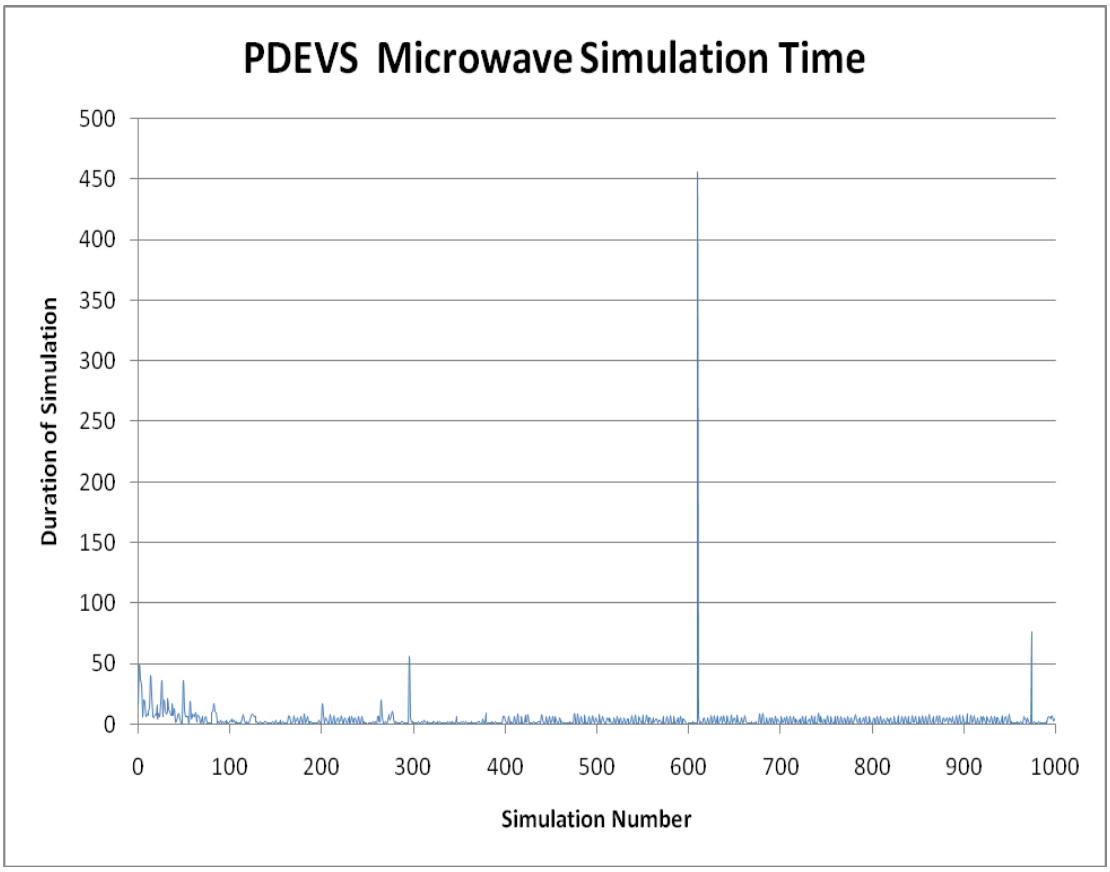


Fig 5.5: PDEVS Microwave Example Simulation Time

5.4 Comparison between CDEVS and PDEVS Simulation Time

We plot the data (the duration of simulation) obtained in the section 5.2 and section 5.3 on the same graph in Fig 5.6, so as to compare between the CDEVS and the PDEVS simulation time. The average time of CDEVS was found to be 3.80 unit time and for the PDEVS is 3.75 unit time. We expect to get less simulation time for the PDEVS due to exploitation of parallelism, which is actually the case here, but the difference is not much due to probably the nature of the model chosen as a case study and the random number generator used.
Considering Fig 5.6, the two simulators shows almost the same behaviour in terms of the duration of simulation at each run.

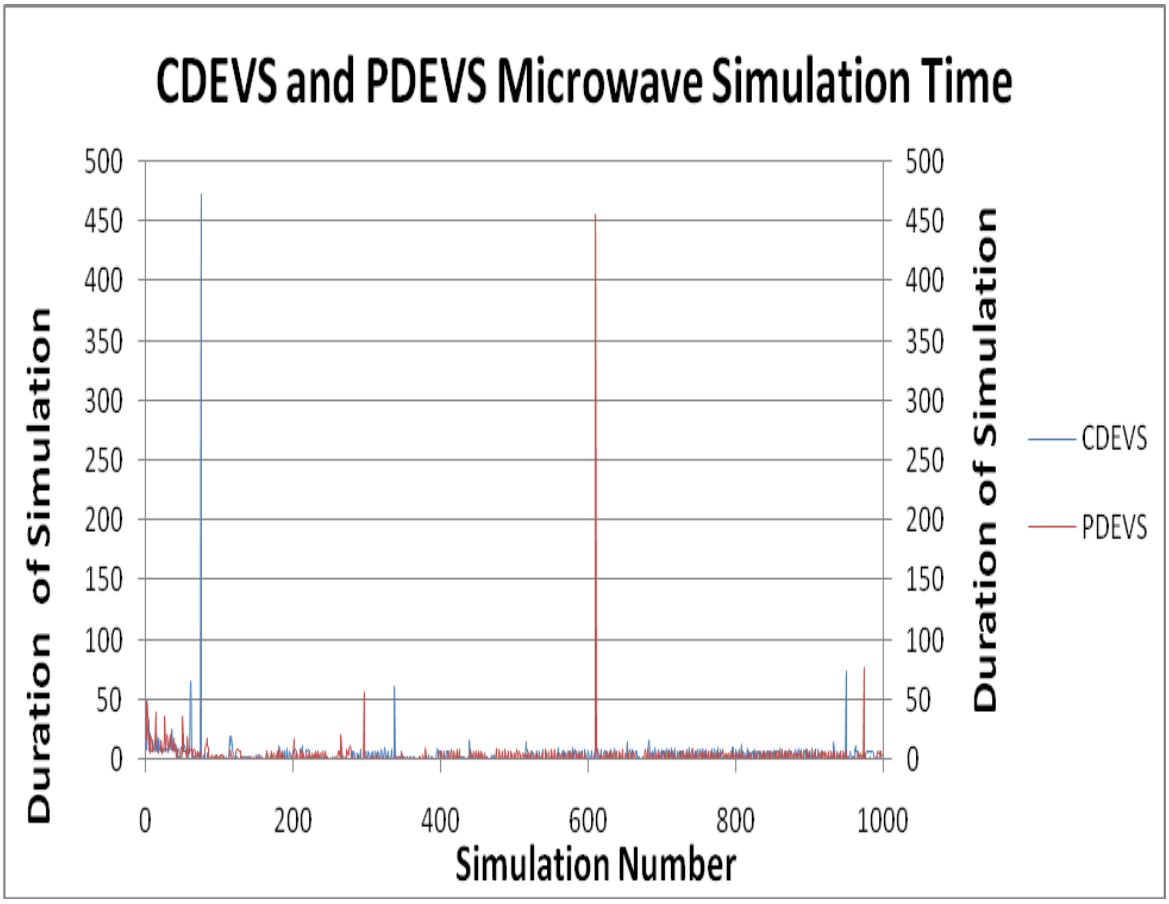


Fig 5.6: CDEVS and PDEVS Microwave Example Simulation Time

Chapter 6

Conclusion

We were able to present and provide two packages, one for the CDEVS object oriented implementation and the other for the PDEVS object oriented implantation. These can be used as our modeling level in the undergoing approach to a simulation virtual machine. We are now left with the distribution of the simulation onto any hardware host like a LAN, a Grid, a Cluster, the Internet and so on. The other thing that we will add is a user friendly method of capturing models and transforming them to either CDEVS or PDEVS modeling formalism.

After completion of the above two pot holes, we will come up with a simulation virtual machine that is can be reused for general purpose environment (as opposed to domain-specific), and at the same time exploiting the computing power of nowadays technologies by allowing distribution of the simulation to gain: execution time reduction, real-time execution, virtual environments geographically distributed, and potential for fault tolerance.

References

Douglas, W. Jones 1986. Implementations of Time, Proceedings of the 18th Winter Simulation Conference

Fujimoto, Richard M. 2000. Parallel and Distributed Simulation Systems, Wiley Series on Parallel and Distributed Computing.

Paul A. Fishwick and Yi-Bing Lin 1996. Asynchronous Discrete Event Simulation, IEEE Transactions on Systems, Man and Cybernetics.

Wikipedia 2007. http://en.wikipedia.org/wiki/DEVS .

Zeigler, Bernard P. 1976. Theory of Modeling and Simulation (First Edition). Wiley Interscience, New York.

Zeigler, Bernard P. 1984. Multifacetted Modeling and Discrete Event Simulation. Academic Press, London; Orlando.

Zeigler, Bernard P. , Herbert Praehofer, and Tag Gon Kim 2000. Theory of Modeling and Simulation (Second Edition), Academic Press