ABSTRACT

Discrete Event System Specification (DEVS) formalism is a Modeling and Simulation (M&S) framework that provides a means of specifying an object called a system. This work presents implementations of this formalism using the Parallel DEVS simulation algorithm and developed as a new simulation engine (simulator). The presented techniques use an approach that exploits the Object Oriented Programming (OOP) paradigm concepts through Java programming language. Our goals include the implementations of this algorithm, benchmarking and analysis as well as using OOP approach to implement the CDEVS algorithm. Also, the implementations we propose are: one for sequential computers and the other for parallel computers. This is for comparison reasons regarding the real gain in performance (due to the exploitation of parallelism).

ACKNOWLEDGEMENTS

To God be the glory for the great things He has done.

I will like to express my gratitude to Professor Mamadou Kaba Traore for his support, guidance and encouragement throughout this work. Working with him has been a challenging yet rewarding experience.

Also, my appreciation goes to the African University of Science and Technology (AUST), Abuja community for providing support and a wonderful research environment.

To my family, friends and FWC Care group, thank you for your timely support, prayers and encouragement.

May God bless you all.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Besides the evolution in humans learning about nature, the twentieth century witnessed the creation of many artificial applications such as traffic control systems, automated factories, computer architectures, or biomedical devices that made it difficult to study especially when the complexity or the required level of detail is high. However, Modeling and Simulation (M&S) provide a well-developed, well-proven approach to solving such problem that advances steadily as more computing power becomes available at less cost.

There is a need to exploit the computing power of nowadays technologies by distributing simulation on multiple processors to be able to model and predict the behavior of complex systems (Road Transport Network, Weather forecast, Fire spread and many others). This is to reduce execution time, perform real time execution, and integrate simulators. The development of computers has offered alternative methods; models can be executed using computer simulation, allowing users to experiment different conditions under risk-free environments [1]. The advantages with this approach include model reuse, user interactions with the simulator, verification and validation of models and economic benefits. The M&S process begins with a problem that needs to be solved or understood. According to Zeigler [2], the source system is the environment under analysis while the model is an actual representation of such system. A simulator executes the model's instructions thus generating the model's behavior.

Discrete-event modeling is based on the notion of event [3], which is defined as a change in the state of the model. An event occurs at a given instant (called the event time) and causes the model to activate in order to produce a state change (e.g., at least one attribute in the model will change). Finally, a model’s state is the set of values of all the attributes of the model at a given instant. The model’s attributes are usually stored in variables; state variables are those that will influence the evolution of the model’s behavior.

1.2 OVERVIEW OF THE TOPIC

Nowadays, in Modeling and Simulation (M&S) exist several formalisms that are used to model and simulate different types of systems. In this work, we focus on the DEVS (Discrete Events Systems Specification) formalism which was originally defined in the 1970’s as a discrete-event M&S mechanism. DEVS [2] is a sound formal framework based

on generic dynamic systems concepts that supports provably correct, efficient, event-based simulation. The framework enables the construction of models in a hierarchical, modular fashion, allowing component reuse and reducing development and testing time.

1.3 OBJECTIVES

Our main objective is to build a DEVS virtual machine (DEVS VM), i.e. a middleware that embeds various implementations of the DEVS algorithms and provide a platform-independent programming environment that abstracts away details of the underlying hardware or operating system, and allows a simulation model to be executed in the same way on any platform using the DEVS VM. This provides a high-level abstraction where the modeler specifies models directly in its terms. Such an approach reduces the cost of development and the amount of error when driving a simulation project, by preventing repeated implementations of the algorithms. This advantage is provided through the Java Virtual Machine (JVM).

1.4 THESIS ORGANIZATION

This work is organized as follows: Chapter 2 introduces the Classic DEVS (CDEVS) and Parallel DEVS (PDEVS) formalisms, as well as some general concepts. A survey on existing DEVS-based simulation tools is presented. Chapter 3 presents how the abstract simulators were implemented. In Chapter 4, we present the methodology used and the technical manual of the DEVS Virtual Machine (VM) and in Chapter 5 a performance analysis is carried out between the implemented simulators. Finally, Chapter 6 provides our conclusions and future work.

CHAPTER 2

DISCRETE EVENT MODELING AND SIMULATION (M&S) TECHNIQUES

2.1 DISCRETE EVENT SYSTEM SPECIFICATION (DEVS)

In Discrete Event System simulation (DESs), the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant time and marks a change of state in the system [5]. There are three common types of 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. In addition to the representation of system state variables and the logic of what happens when system events occur, discrete event simulations include the Clock, Events List, Random-Number Generators, Statistics and Ending Condition.

Discrete Event System Specification (DEVS) was first introduced by Ziegler [6] in 1976 as formalism for modeling and analysis of Discrete Event Systems. It provides a means of specifying a system whose states change either upon the reception of an input event or due to the expiration of a time delay. Basically, a system has a time base, inputs, states, and outputs, and functions for determining next states and outputs given current states and inputs [7]. Based on this information, it is possible to design new simulation engines that can take in models, simulate them and produce the results.

In 1984[8], Zeigler proposed a hierarchical simulation algorithm for DEVS model simulation. Since then, many extended formalisms from DEVS have been introduced with their own purposes: DESS/DEVS for combined continuous and discrete event systems, P-DEVS for parallel DESs, G-DEVS for piecewise linear state trajectory modeling of DESs Due to the modular and hierarchical modeling views as well as its simulation-based analysis capability, DEVS formalism and its variations have been used in many application of engineering (such as hardware design, hardware/software co-design, communications systems, manufacturing systems) and science (such as biology, and sociology)[9].

2.2 CLASSIC DEVS (CDEVS) AND PARALLEL DEVS (PDEVS)

DEVS [6] defines system behavior as well as system structure. System behavior in DEVS formalism is described using input and output events as well as states while system structure as the composition of atomic or coupled components. Some of the ways in which these

models can be implemented are by using the Classic DEVS (CDEVS) simulator which was first developed or the Parallel DEVS (PDEVS) simulator. These simulators use algorithms to execute the DEVS models. As such this execution can be on sequential single processor systems in the case of CDEVS or on multiple processors architectures concurrently as in PDEVS.

2.2.1 ATOMIC CLASSIC DEVS MODEL

An atomic CDEVS 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: $S \rightarrow R_0^+ U \infty$ is the time advance function,
- $\delta ext : Q \times X \rightarrow S$ 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
- $\lambda: S \rightarrow Y$ is the output function

At any given moment, a DEVS model is in a state $s \in S$. In the absence of external events, it remains in that state for a lifetime defined by ta(s). When ta(s) expires, the model outputs the value $\lambda(s)$ through a port y, and it then changes to a new state given by $\delta int(s)$. A transition that occurs due to the consumption of time indicated by ta(s) is called an internal transition. On the other hand, an external transition occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by $\delta ext(s, e, x)$, where s is the current state, e is the time elapsed since the last transition, and $x \in X$ is the external event that has been received. The time advance function can take any real value between 0 and ∞. A state for which ta(s) = 0 is called a transient state i.e imminent (which will trigger an instantaneous internal transition). In contrast, if ta(s) = ∞, then s is said to be a passive state, in which the system will remain perpetually unless an external event is received (can be used as a termination condition).

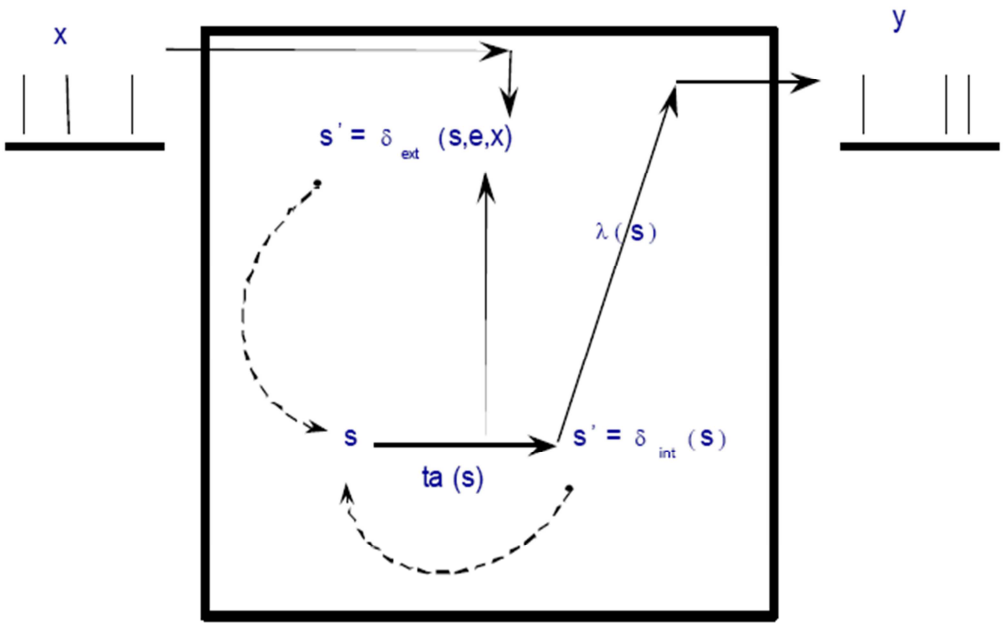


Figure 2.1: DEVS in action

The pseudo code algorithm of the simulator that would execute, and generate the behavior of the semantics of the atomic CDEVS model described above is given below:

- 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, text (no earlier than the current clock and no later than tN), the clock is advanced to text.
    - If text == tN the output is generated.
    - Then the input is processed by external event transition function.

2.2.2 COUPLED CLASSIC DEVS MODEL

A DEVS coupled model is composed of several atomic or coupled sub-models. It is formally defined by

CM = (X, Y, D, {M_d | d Є 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 Є D, M_d is a subcomponent. 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 : 2D – {} → D, the tie-breaking function

In coupled CDEVS multiple subcomponents can be scheduled for an internal transition at the same time, ambiguity could arise. For example if a subcomponent executes its output/internal transition first, producing an output that maps into an external event for another subcomponent (which is also scheduled for an internal transition at the same time), then it is not clear which transition this second component should execute first. There are two alternatives for this:

- to execute the external transition first and then the internal transition, with e = ta(s); or
- to execute the internal transition first, followed by the external transition, with e = 0.

The select function provides a simple way to solve this ambiguity. The function defines an ordering (tie-breaking) over all the components of the coupled model so that only the first model to execute in the case of simultaneous internal events can be chosen.

This strategy for tie-breaking is rigid and, in addition, it introduces serialization in the execution of components. The serialization introduced by this approach becomes visible when the select function has to be used to determine the priority in which the components have to be executed. For example, the select function is used to determine which atomic component has priority over the rest to execute its internal transition function when many interconnected atomic models are imminent.

For a coupled model with atomic model components, a coordinator is assigned to it and 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 as shown below:

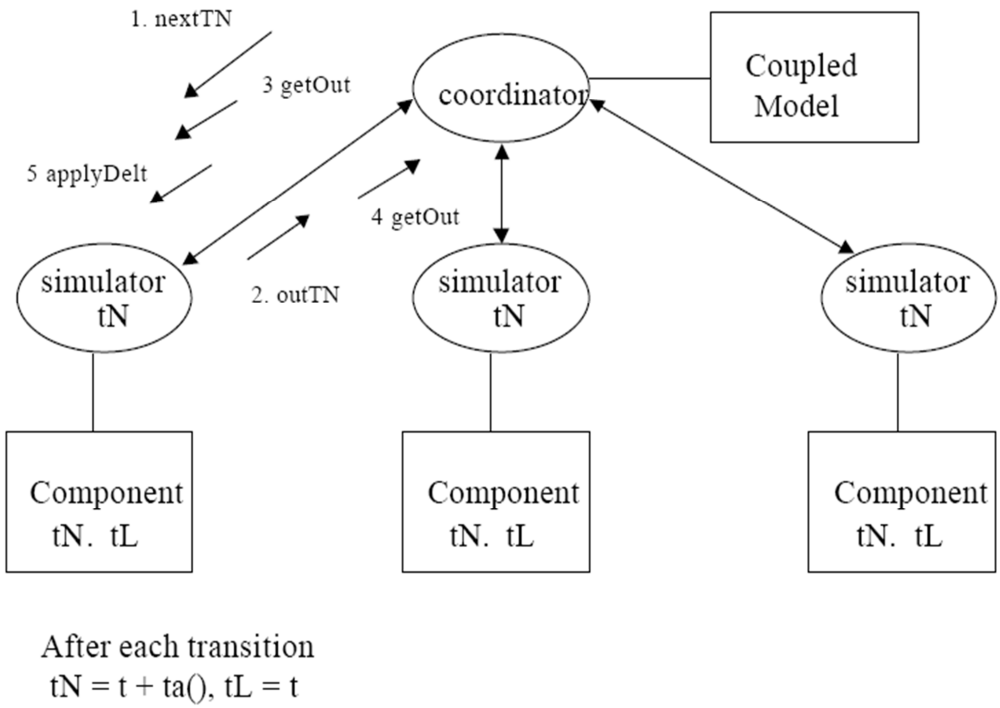


Figure 2.2: How the Coordinator works

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

- Coordinator sends nextTN to request tN from each of the simulators.
- All the simulators reply with their tNs in the outTN message to the coordinator.
- Coordinator sends to each simulator a getOut message containing the global tN (the minimum of the tNs).
- 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.
- Coordinator uses the coupling specification to distribute the outputs as accumulated messages back to the simulators in an applyDelt message to the simulators.
- 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

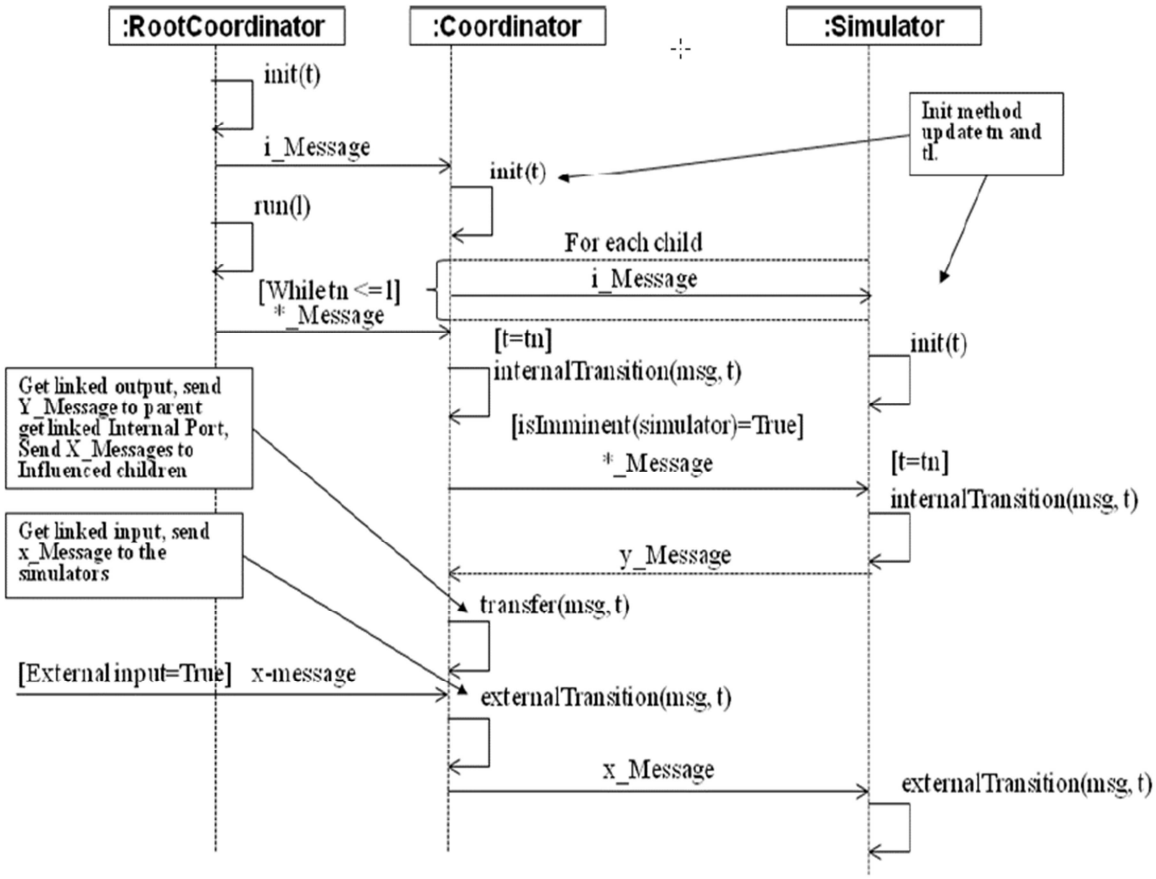


Figure 2.3: Sequence Diagram for the CDEVS Simulator

2.2.3 ATOMIC PARALLEL DEVS MODEL

The Parallel DEVS (PDEVS)[10] was introduced to properly handle collisions during tie-breaking i.e. the behavior when a model receives external events at the same time as its prescheduled internal transition. Previous solutions attempt to define the collision behavior implicitly either through the select function or by saying that an internal transition should occur before a colliding external transition. The P-DEVS formalism structure explicitly requires a modeler to define the collision behavior by using the confluent transition function, δconf. According to Chow [10] other desirable properties provided by PDEVS are degrees of parallelism which can be exploited in parallel and distribution environments and uniformity through the hierarchical construction of models.

P-DEVS permits increased degrees of parallelism that can be exploited in parallel and distributed environments. An atomic PDEVS model is defined as

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

Where

- X_M = {(p,v)| p ∈. IPorts, v ∈ Xp} is the set of input ports and values;
- Y_M = {(p,v)| p ∈ OPorts, v ∈ Yp} is the set of output ports and values;
- S is the set of sequential states;
- δext: Q x $X_M^b \rightarrow$ S is the external state transition function;
- δint: S → S is the internal state transition function;
- δconf: Q x $X_M^b \rightarrow$ S is the confluent transition function;
- λ : S → Y_Mb is the output function;
- ta : S → $R_0^+ \rightarrow \infty$ is the time advance function;

with Q = {(s, e) | s ∈ S , 0 ≤ e ≤ ta(s)} the set of total states.

There are two main differences between a CDEVS and a PDEVS atomic model. First, the external transition function uses a bag of events instead of a single event. This allows multiple events to be processed simultaneously. Since external events received by the component are added to a bag, X_Mb, external transition functions can combine the functionality of a number of external transitions into a single one. Second, the model specification includes a confluent transition function (δconf). When a collision between the internal and external functions occurs, the confluent function determines the new state of the model.

The semantics of PDEVS are similar to those of CDEVS. A basic model is in a state s at any given time. In the absence of external events, the model remains in that state for a lifetime period defined by ta(s). When that time expires, an internal transition takes place; the system outputs the value l(s) and then it changes to the state specified by δint(s). If one or more external events E occurs before ta(s) expires, the new state will be given by the model's external transition function, δext(s,e,E). PDEVS allows a better way to deal with collisions. External and internal transitions are in conflict when external events E are received when e = ta(s). In such cases, the new state of the model can be given by δext(δint(s),e,E) or δint(δext(s,e,E)). Hence, modelers have a flexible way of indicating the appropriate behavior for each model in the confluent function (δconf), which is triggered in case of collisions. This is described in Figure 2.4

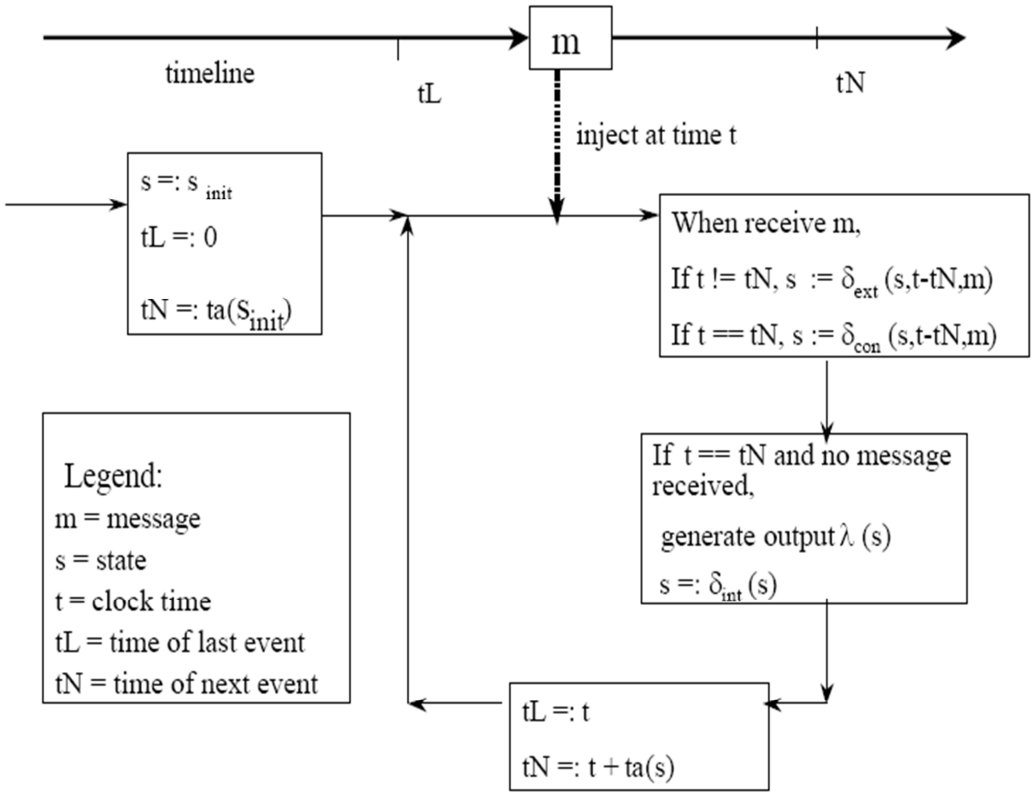


Figure 2.4: Semantics of an Atomic PDEVS Model

2.2.4 COUPLED PARALLEL DEVS MODEL

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

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

The definitions for the set of input and output events (X and Y), components (D and M_d), and couplings (EIC, EOC, and IC) follow the specifications of DEVS coupled models presented earlier in this chapter. If multiple components in a coupled model are imminent, all their outputs are first collected and mapped to their influencees. Then, the corresponding transition function is executed for every model. The diagram below shows the sequence of events in the PDEVS simulator

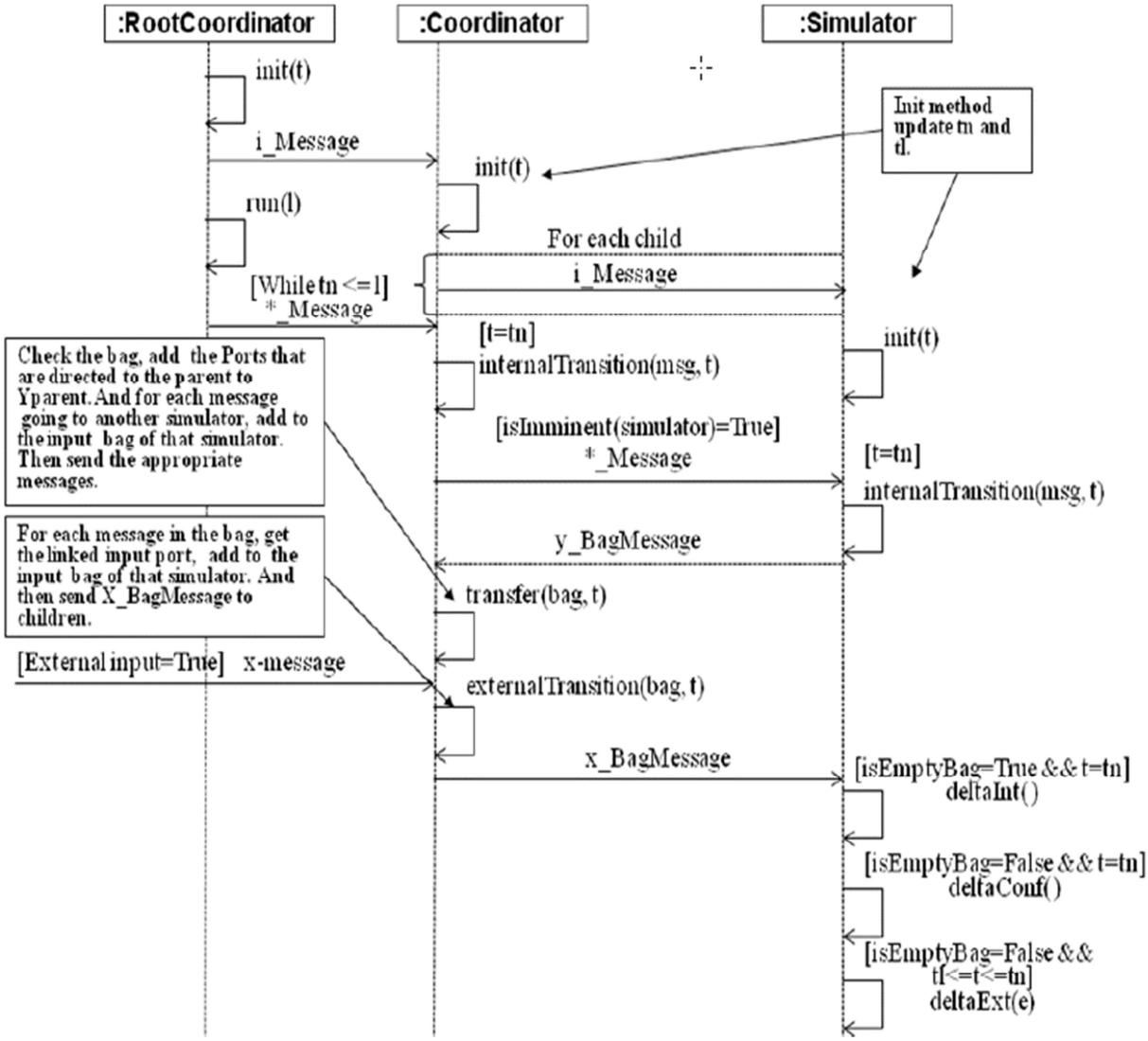


Figure 2.5: Sequence Diagram for the PDEVS Simulator

2.3 REVIEW OF DEVS BASED TOOLKITS FOR M &S

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

- CD++ Builder [11] is an Eclipse plugin that integrates varied applications and utility that aids in creating CD++ DEVS models, simulating and analyzing models.
- ADEVS [12] provides a C++ library based on DEVS, which developers can use to build their own models, and supports integration with other simulation environments.
- DEVS-C++ [13] is a DEVS-based M&S environment written in C++, which implements parallel execution and supports large-scale systems.
- DEVSJAVA [14] is a DEVS-based M&S environment written in Java. It

provides classes for the users to implement their own DEVS models.

- SIMSTUDIO [15] is the virtual machine under development in the LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes).
- DEVSim++ [15] is an object-oriented software to simulate DEVS models, which was implemented in C++. The tool defines basic classes that can be extended by users to define their own atomic and coupled DEVS components.
- JAMES [16] implements DEVS theory to model and simulate agent systems. The toolkit supports software-in-the-loop simulation to test agents in virtual environments.
- JDEVS [17] is a DEVS M&S environment written in Java. It allows general purpose, component-based, object-oriented, visual simulation of models.
- PyDEVS [18] uses the ATOM3 tool to construct DEVS models and to create the code to be executed. Models are represented as a state graph used to generate Python code and then interpreted by PyDEVS.
- SimBeams [19] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, result output analysis and visualization using DEVS.

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

- CORBA (Common Object Request Broker Architecture) [19], an open standard promulgated by the Object Management Group (OMG),
- HLA (High Level Architecture) [20], a standard specifically designed for distributed simulations, and
- MPI [21], a message passing interface standard designed for high performance communication on parallel and distributed environments.

2.4 SYNCHRONIZATION ALGORITHMS

Parallel And Distributed Simulation (PADS) [1] deals with systems that can be executed over multiple processors. In this case, the resources provided by a single-processor machine often become insufficient to execute these systems. Parallel Discrete Event Simulation (PDES) studies the execution of discrete event models in parallel or distributed computers

and the execution of a system is subdivided in smaller, simpler parts that run on different processors or nodes. Each of these sub-parts is a sequential simulation, which is usually referred to as a logical process (LP). A logical process groups one or more simulation objects running in a node.

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

There are two main classes of algorithms for synchronization. Conservative algorithms offer a pessimistic approach. They avoid violating causality constraints at all times during the execution of a simulation by processing events in strict time-stamped order. This pessimistic approach arranges for the potential causality errors through the provision of look-ahead, in which each model provides a time in the immediate future up to which it promises not to send input events. Optimistic algorithms allow some violations to happen, but provide a mechanism to detect and recover from these situations. Optimistic algorithms have two main advantages over conservative approaches: (i) they enable greater degrees of parallelism, and (ii) they do not rely on application-specific data to determine events that are safe to process, which is usually the case in conservative approaches.

2.5 FLAT & HIERARCHICAL APPROACHES TO THE ALGORITHMS

A different way for improving simulation performance and reducing execution times deals with the structure of the simulator. Hierarchical simulation mechanisms incur in greater overheads due to an increased number of exchanged messages that travel up and down the entire structure. Flat simulation approaches have been able to reduce the overhead by simplifying the structure.

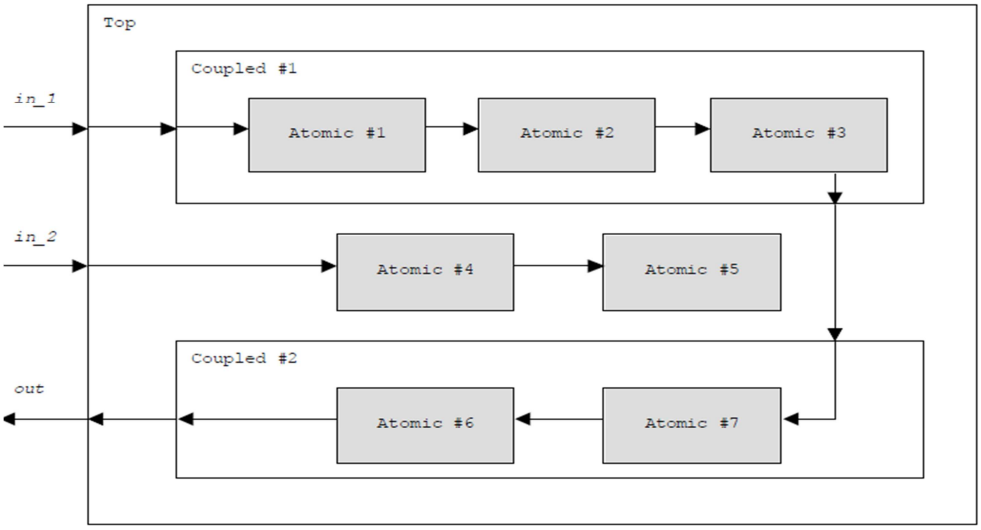
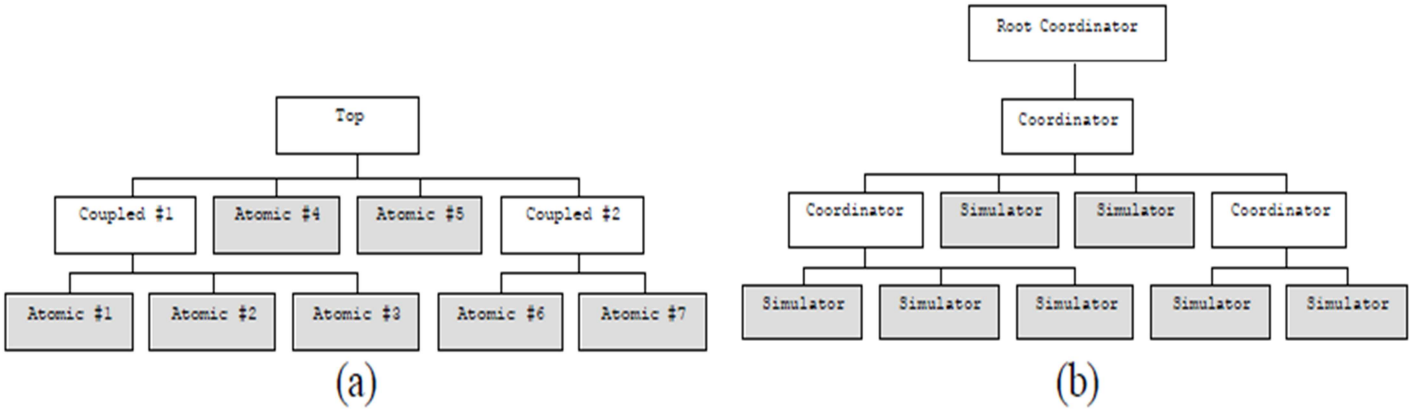


Figure 2.6: Layout of a sample DEVS Model



(a) Models (b) Components of the simulator that drives the models

Figure 2.7: Sample DEVS Model using the Hierarchical approach

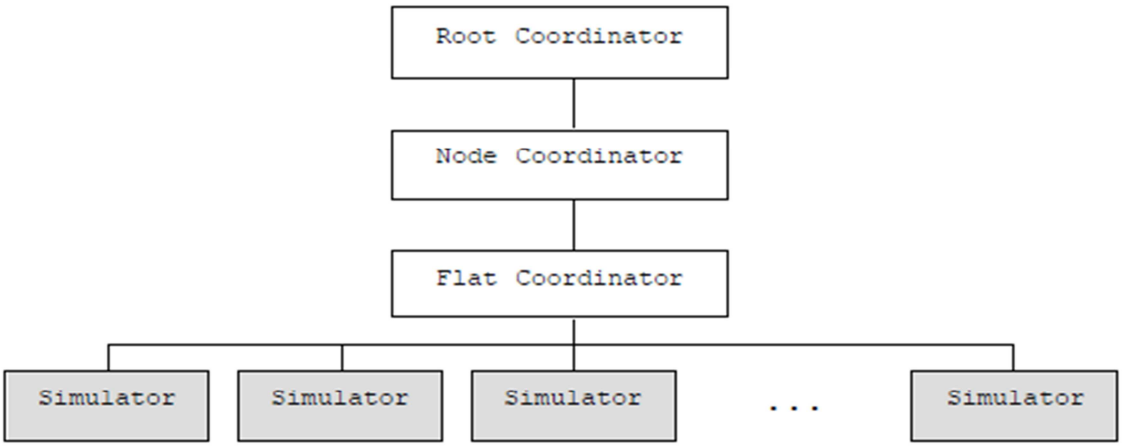


Figure 2.8: Sample DEVS Model using the Flat approach

In this chapter we discuss details about the simulation engine that drives the execution and hierarchical construction of these models. This is possible due to the separation of concerns in DEVS thus allowing the modeler to focus only on the models being created, avoiding the details about the simulation engine that drives them.

3.1 DEVS META-MODEL

Different implementations of the DEVS formalism share the same semantics due to the DEVS mathematical specification, but they differ in the underlying software design. In order to allow an abstraction for different implementations, we have defined a Model class which can be atomic and coupled as shown in Figure 3.1. A simulator usually directly invokes operations on the model.

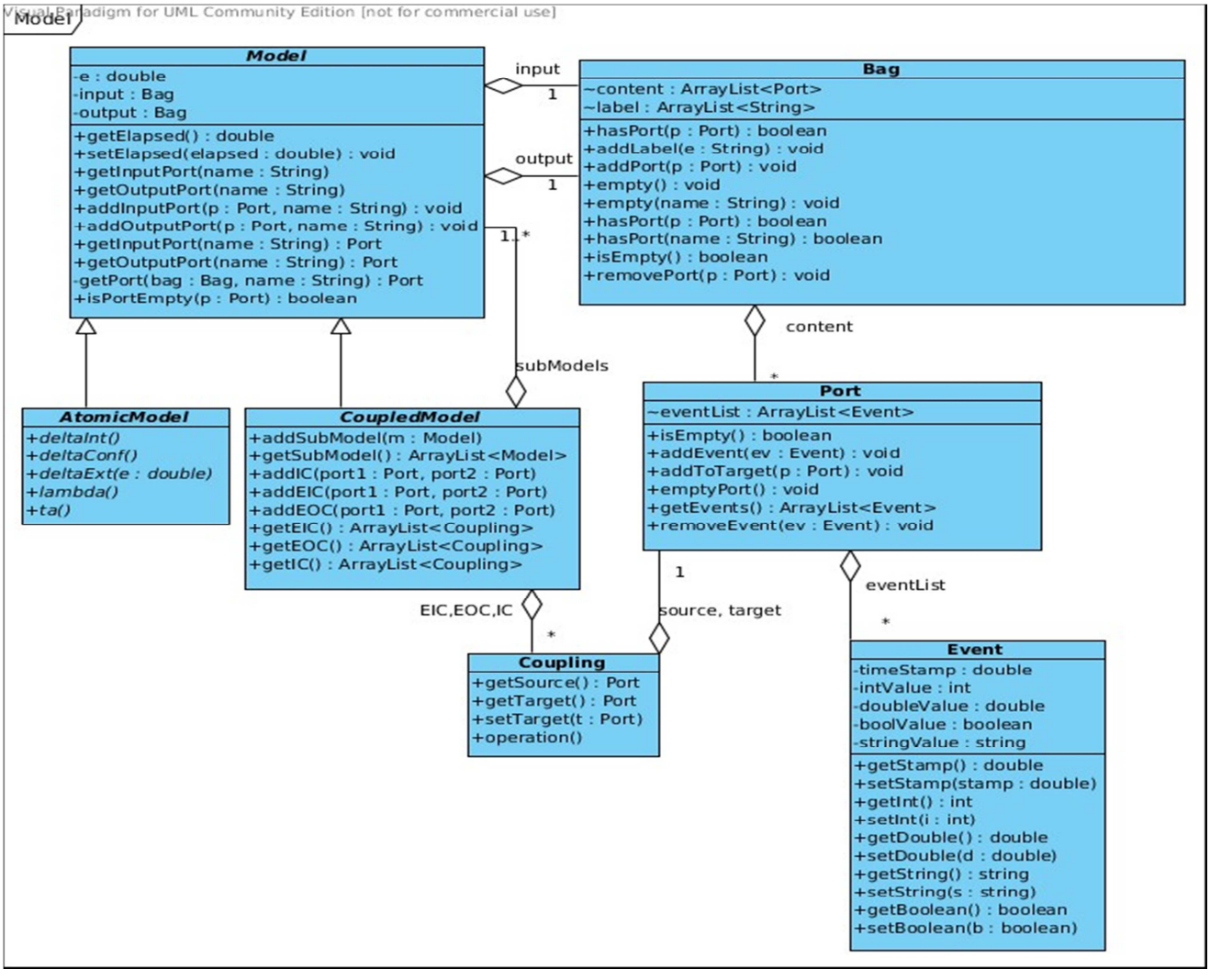


Figure 3.1: The Model Package

Ports are represented explicitly I.e. there can be any of input and output ports on which values can be received and sent. Each Model is associated with a bag. Instead of receiving a single input or sending a single output the models (atomic or coupled) can handle bags of ports which can be input or output. Each of these ports can take in events simultaneously.

The Model class is an abstract class that provides the basic functionality and data that are common to all DEVS models in the application. It defines the bag (input and output) and the elapsed time. Atomic and Coupled Models are subclasses of the superclass which the modeler must inherit. They contain methods required for the specified type of model. As part of DEVS specification a Coupled model can have sub-models which can be atomic or coupled. Also each coupled model contains port couplings which are External Input Coupling (EIC), External Output Coupling (EOC) and Internal Coupling (IC). According to the PDEVS specifications a model makes use of bags, in the Bag class are methods and data that define the structure of the bag in terms of ports and their labels. The Event class describes the data structure of events that are contained in a port.

3.2 SIMULATION HIERARCHY

We have different types of implementations; one based on the threading mechanism, the other on the object-oriented organization and more details on them is given in Chapter 4. The Abstract Simulator as shown in Figure 3.2 is split into two simulation engines the simulator which drives the atomic models and the coordinator which drives the coupled models. The Root Coordinator drives the global aspects of the simulation; it maintains the global time, starts/finishes the simulation (when a termination condition is detected), and is related to the Coordinator of the top-level coupled model (collecting the outputs from it and feeding it with external input events).

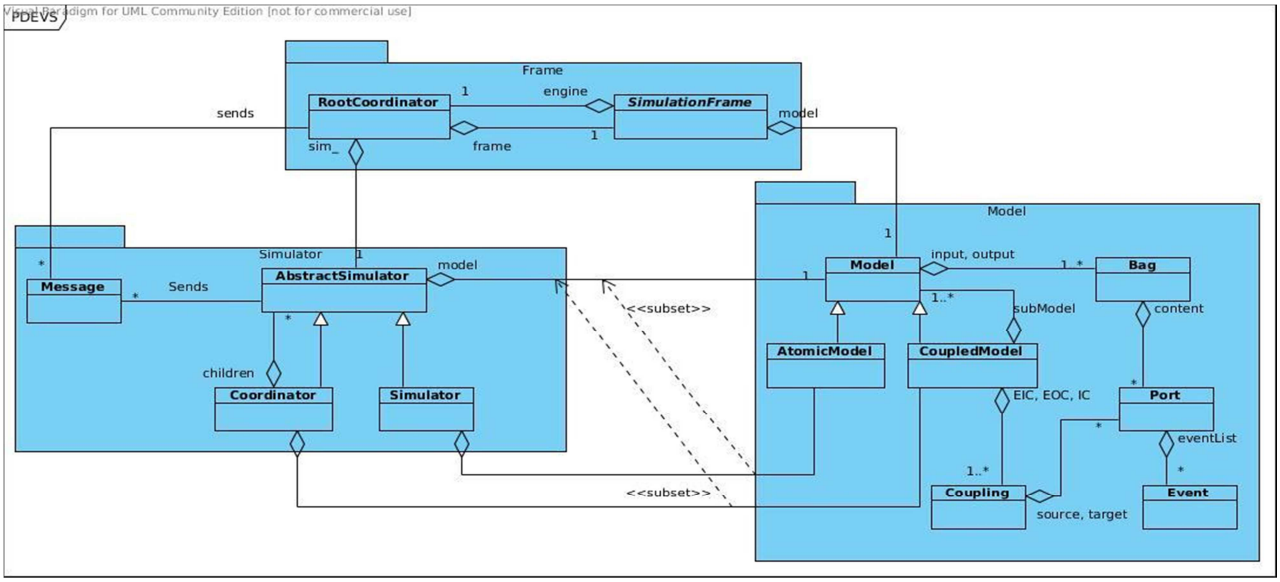


Figure 3.2: Package Diagram of the PDEVS (non-Threaded) Simulator

In the implementation of PDEVS with thread, the Abstract Simulator includes the multi-threading facility to depict the concurrent processing of events within the simulator. This is shown in Figure 3.3 showing the structure of the Simulator.

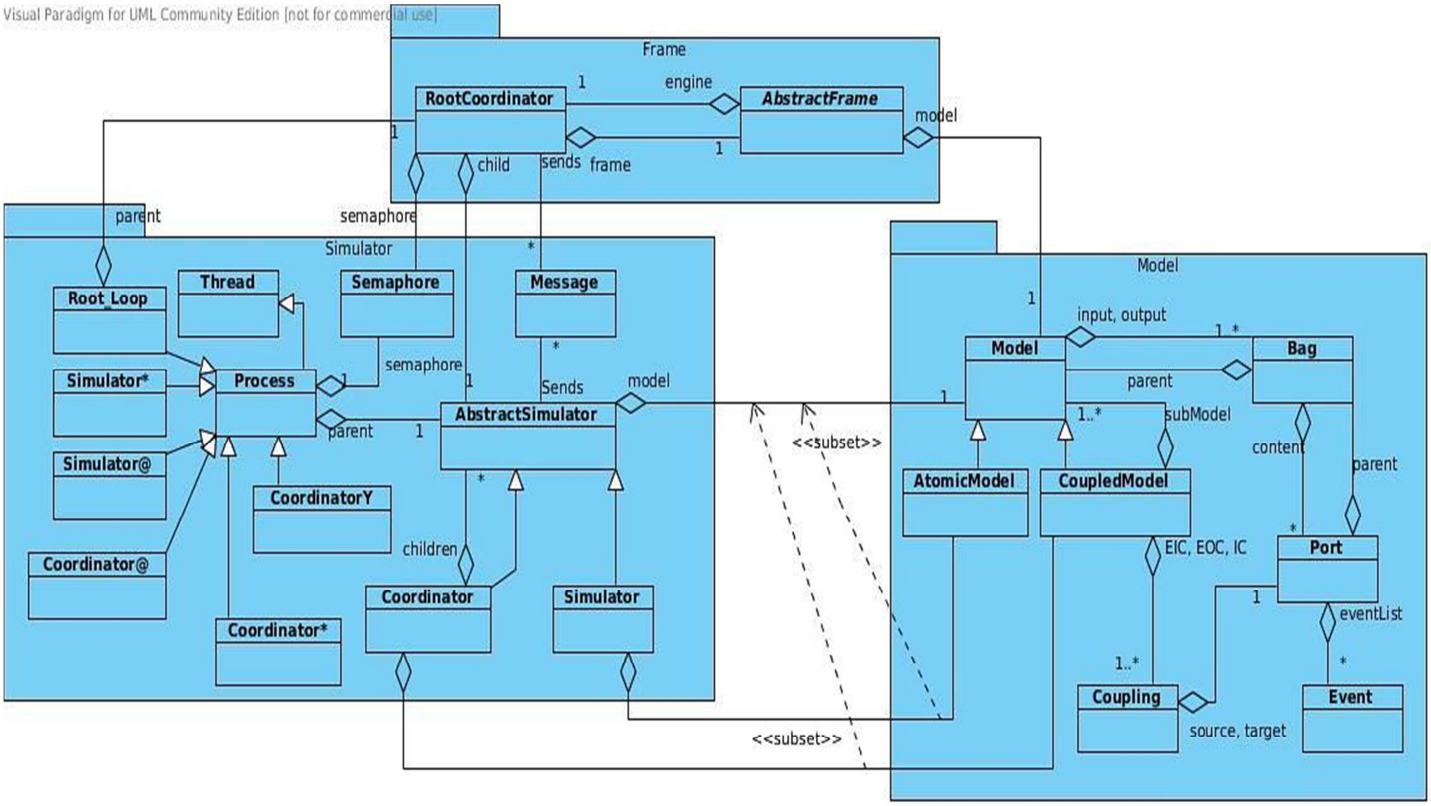


Figure 3.3: Class Diagram of the PDEVS (Threaded) Simulator

3.3 MESSAGE PASSING

Simulation is driven by passing messages among the Processors; each represents an event to process. The messages include information about the event origin/destination, the time of the event the message represents, and its content. Three kinds of messages are used:

- I Messages signal the initialization of the simulator.
- * Messages signal the occurrence of internal events.
- @ Messages carry information about external input events.

Figure 3.4 describes the sequence of events in the simulator and how the messages are passed when the simulation is started. Also Figure 3.5a and b show how these actions take place in the PDEVS (threaded) implementation. The messages that are passed are the same with the three mentioned above also with the inclusion of messages that transport the output content in the simulator;

- y messages: causes the outputs to be produced

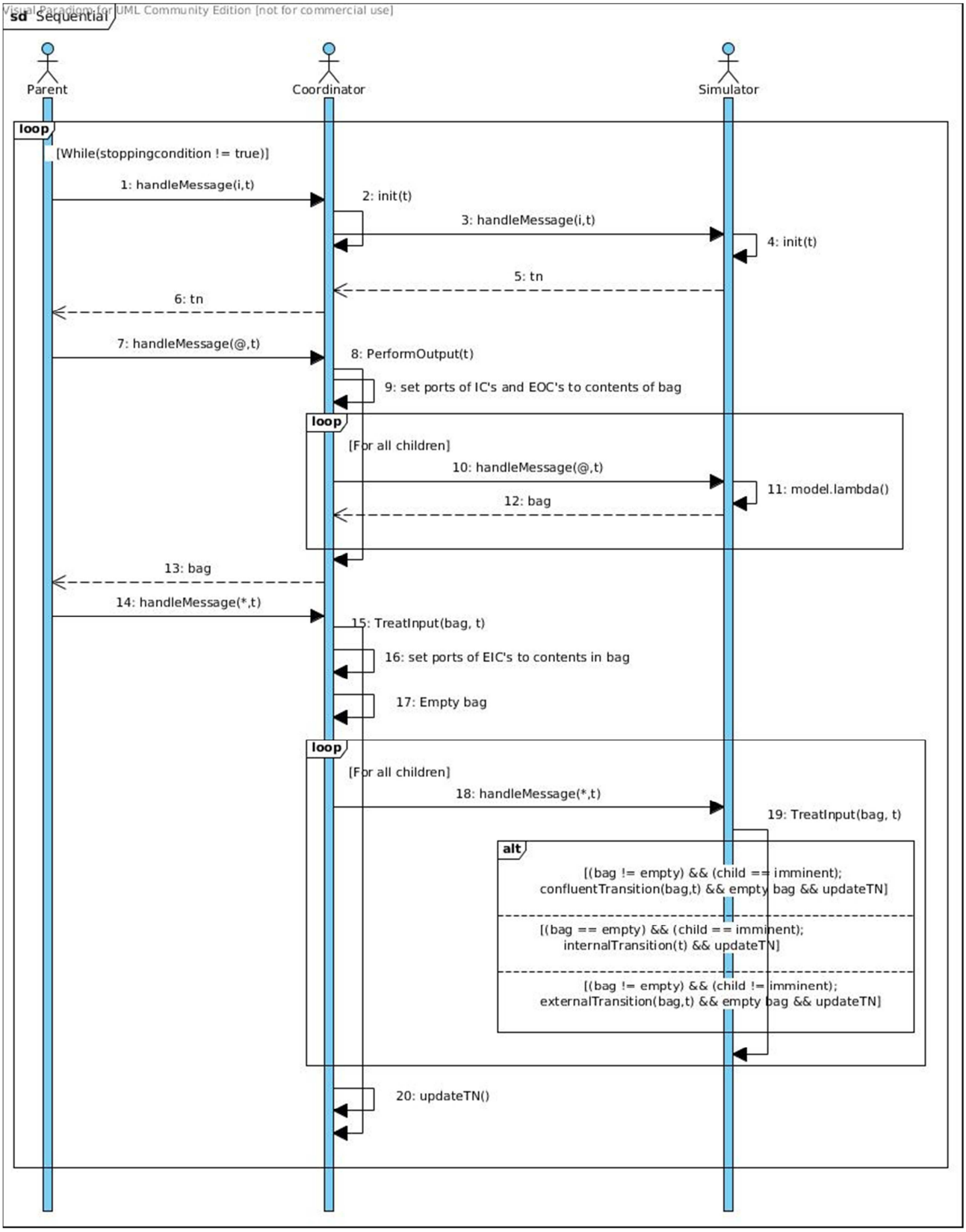


Figure 3.4: Sequence Diagram showing events in the PDEVS (non-Threaded) simulator.

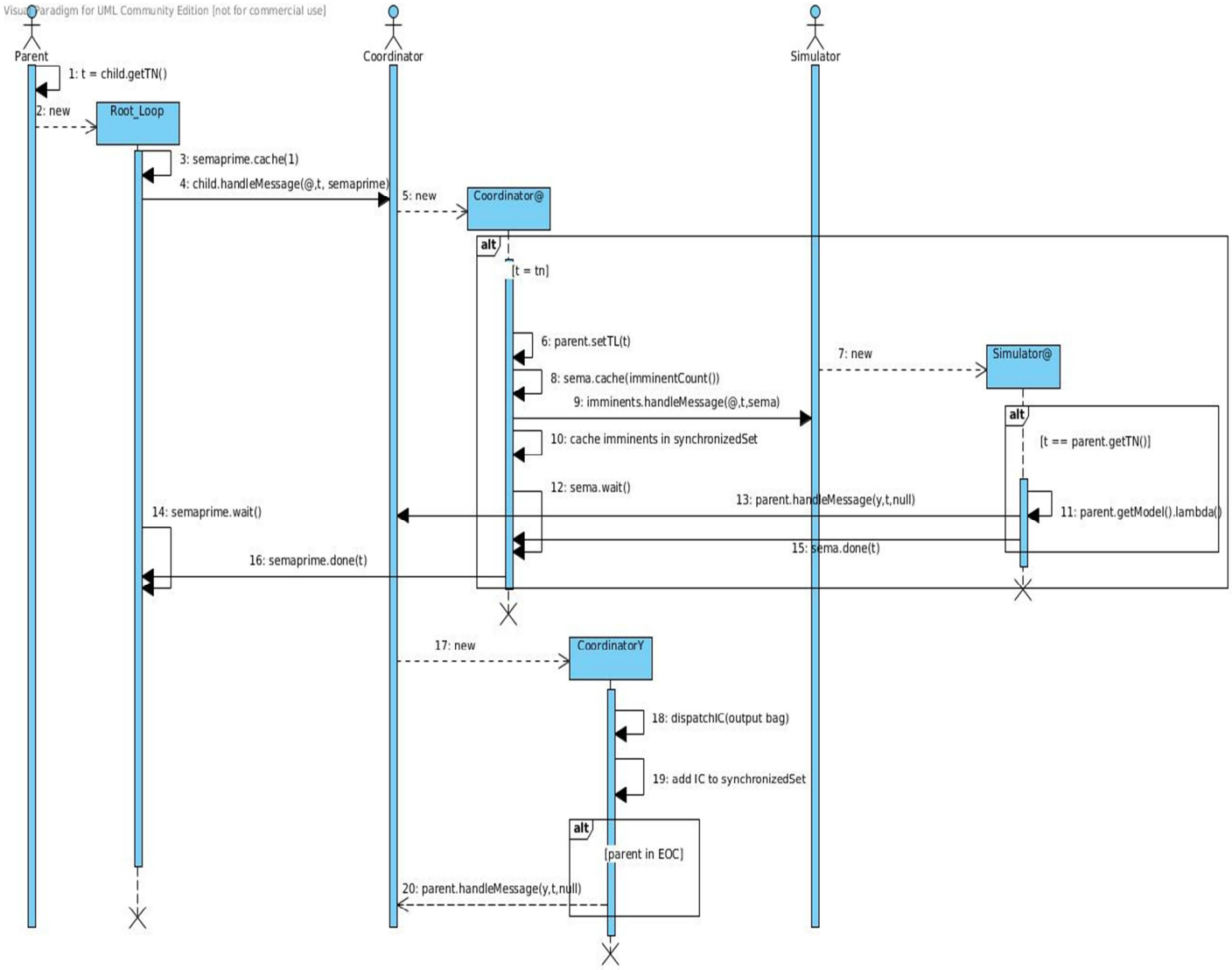


Figure 3.5a: Sequence Diagram: @ and Y message passing in the PDEVS (Threaded) simulator.

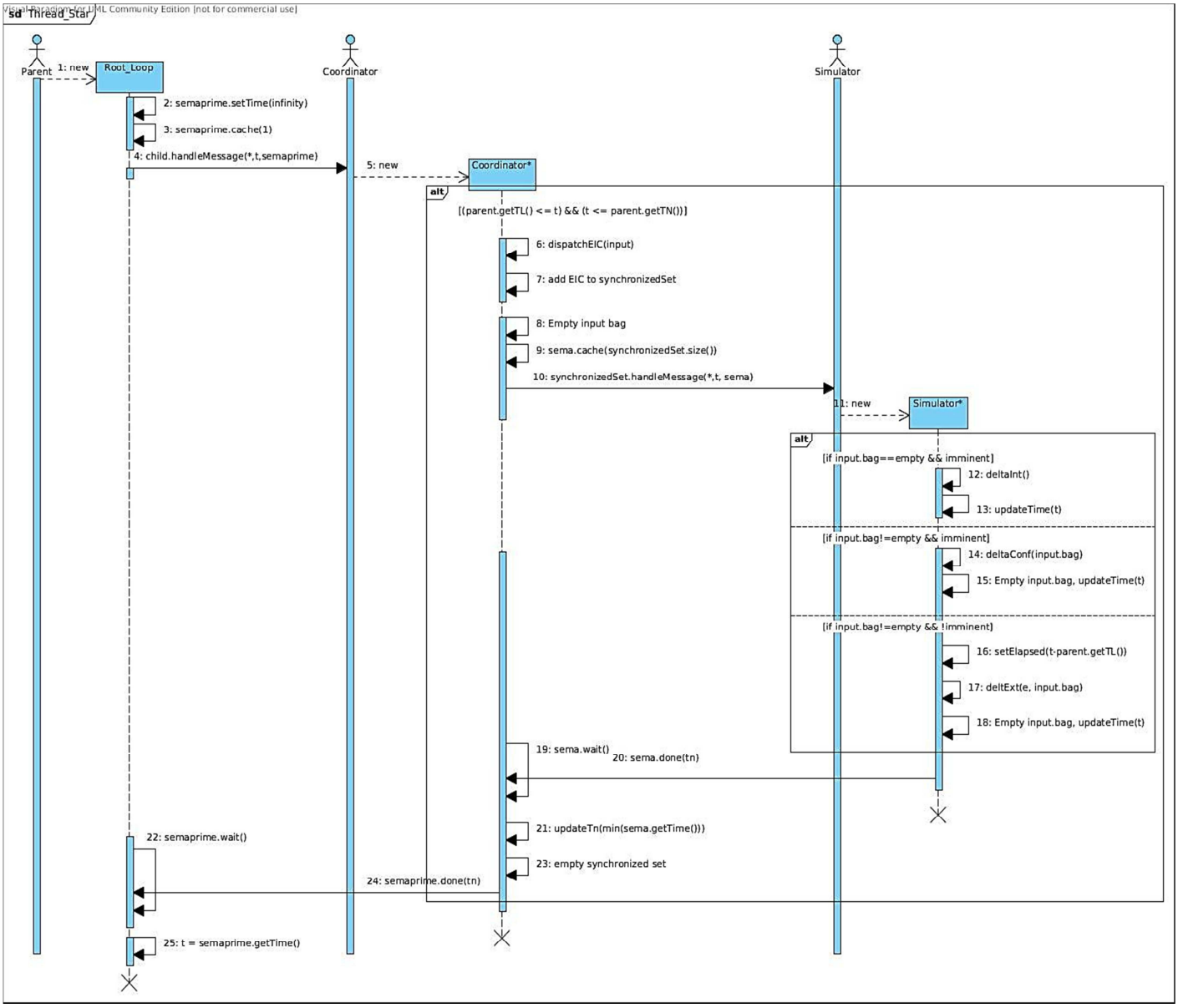


Figure 3.5b: Sequence Diagram: * message passing in the PDEVS (Threaded) simulator.

CHAPTER 4

MODELLING AND IMPLEMENTING THE SIMULATION SYSTEM

This section presents modeling of the implementations of the simulation system using Unified Modeling Language (UML) [22].

4.1 IMPLEMENTATIONS OF THE SIMULATION SYSTEM

The different types of implementations we have are:

- **I:** An implementation of the PDEVS simulation algorithm in a sequential manner.
- **II:** An implementation of the PDEVS simulation algorithm that takes advantage of parallelism provided for by Java threads, so that we can evaluate the overhead of computation due to multi-threading; therefore we can estimate the real gain or loss of performance whether parallelism is fully exploited or pseudo-parallelism is used.
- **III:** A third implementation, that realizes the CDEVS simulation; this one will provide the pure sequential version of DEVS (sequential in nature, sequential in execution), as opposed to the thread-based PDEVS implementation (parallel in nature, parallel in execution) and the non-thread-based PDEVS implementation (parallel in nature, sequential in execution).

4.2 PACKAGE AND CLASS VIEW OF THE SIMULATOR

4.2.1 COMMON ENTITIES

There are some packages containing classes that are common to the threaded and non-threaded implementations and they are:

- **Frame Package:** is used to start the simulation process. It contains RootCoordinator Class that manages the global clock and controls the execution of the simulators/coordinator hierarchy while the AbstractFrame class manages the number of times the simulation should run. The package also imports or makes use of functions or attributes provided by other packages.

These packages were implemented by defining AbstractFrame as an Abstract class which the modeler must inherit from. This class sets the simulator into motion by calling the RootCoordinator's run() method in its runExperiment(). If the condition defined by the modeler in the abstract method endingCondition() is true the simulation ends.

InitializeFrame() is used to specify the model to start the simulation with while the init() initializes the whole system.

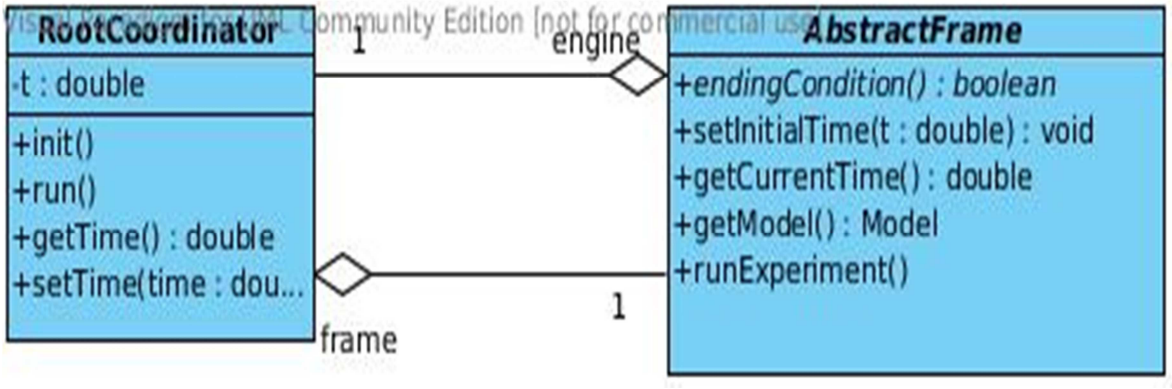


Figure 4.1 Frame Package

- **Model Package:** contains description required by a model. The implementation of this package has been described in chapter 3
  - Model: This is an abstract class containing attributes and methods common to both the AtomicModel and CoupledModel Classes.
  - AtomicModel: This contains information required from the modeler about the structure of the atomic model.
  - CoupledModel: This contains information required from the modeler about the structure of the couple model
  - Bag: It defines the structure of the bag of ports.
  - Port: Defines the input and output ports of a model.
  - Event: Defines the data structure of event a port should receive.
  - Coupling: Describes the coupling information required by the CoupledModel.

A diagram of the Model Package is shown in Figure 3.1

4.2.2 IMPLEMENTATION I

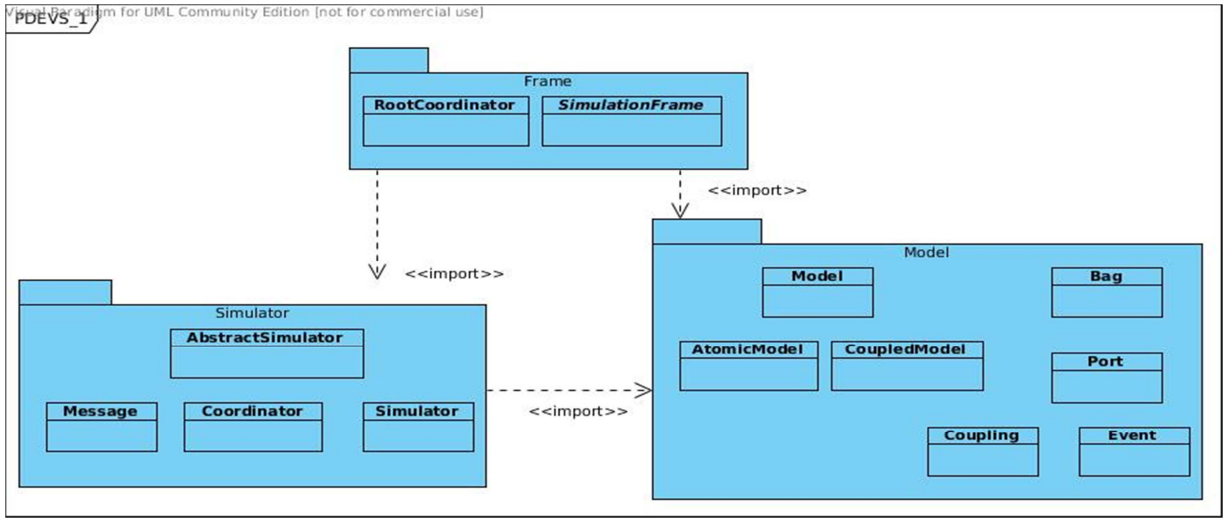


Figure 4.2: Package and Class View of Implementation I

In the simulator exist 3 packages namely Frame, Model both of which have been described in the previous section as well gave Simulator described below.

- **Simulator Package:** This package provides the classes that run the entire simulator.
  - AbstractSimulator: This is an abstract class containing attributes and methods common to both the Simulator and Coordinator Classes which are its subclasses.
  - Simulator: It controls the atomic models
  - Coordinator: It controls the coupled models and is an AbstractSimulator
  - Message: It contains the structure of the message to be sent and received in the package

We defined an abstract class in the simulator package, the AbstractSimulator. It defines the method handleMessage() which executes a method depending on the type of message it sends or receives. setTN(), getTN(), setTL(), getTL() are used for time management in the simulator. They are used to report time of last change in events and time of next change. The treatInput(), performOutput() are used to define the actions the simulator to take on the receipt of a message. Since a Simulator has access to the definition of its associated atomic model it is possible for it to execute the internal, external and confluent transition functions. Access to associated coupled model would enable the Coordinator execute functions that are based on the defined couplings in the model. In this implementation the "q and y messages" which transports the outputs have been implicitly defined when dispatching to the couplings.

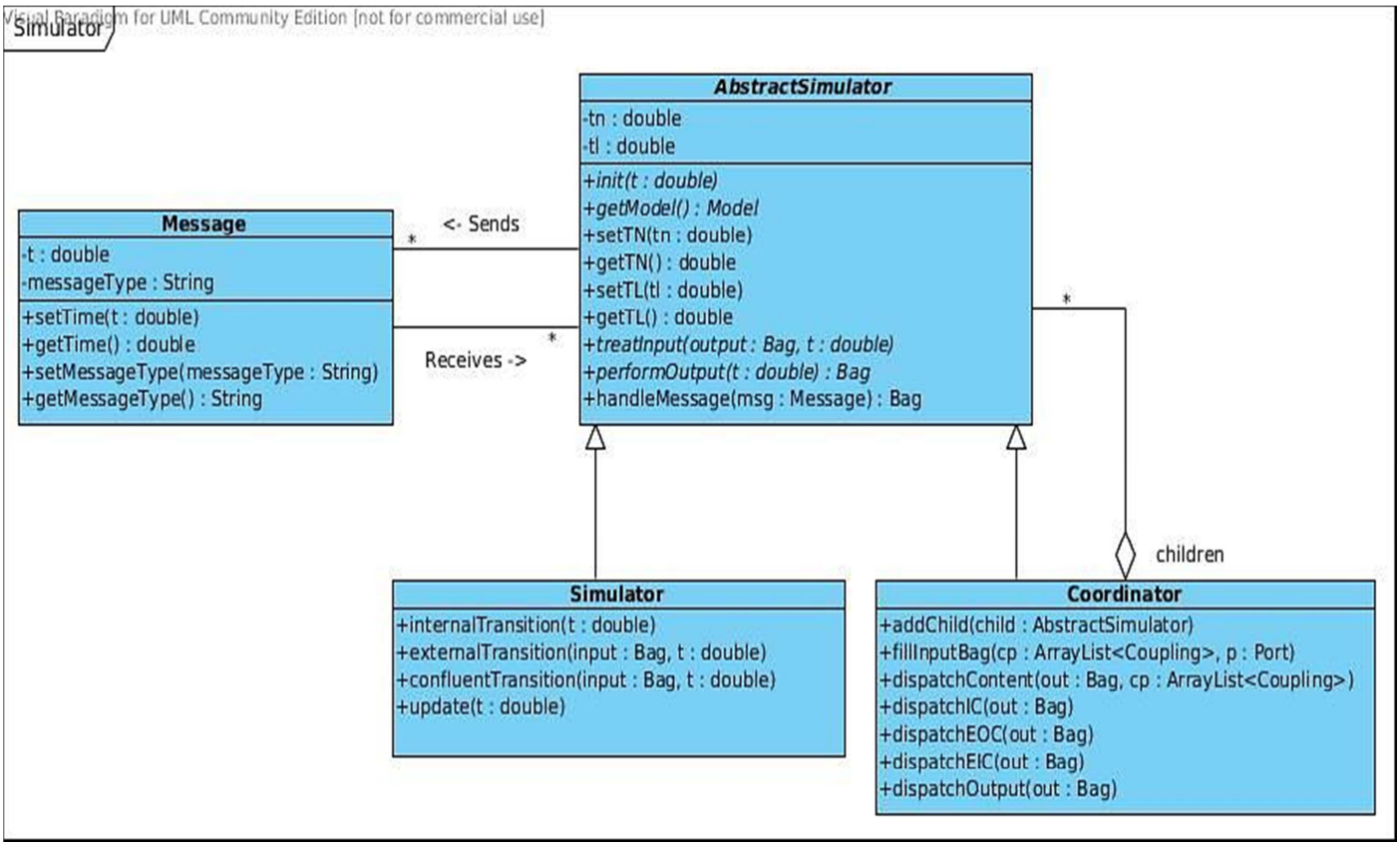


Figure 4.3: Simulator Package

4.2.3 IMPLEMENTATION II

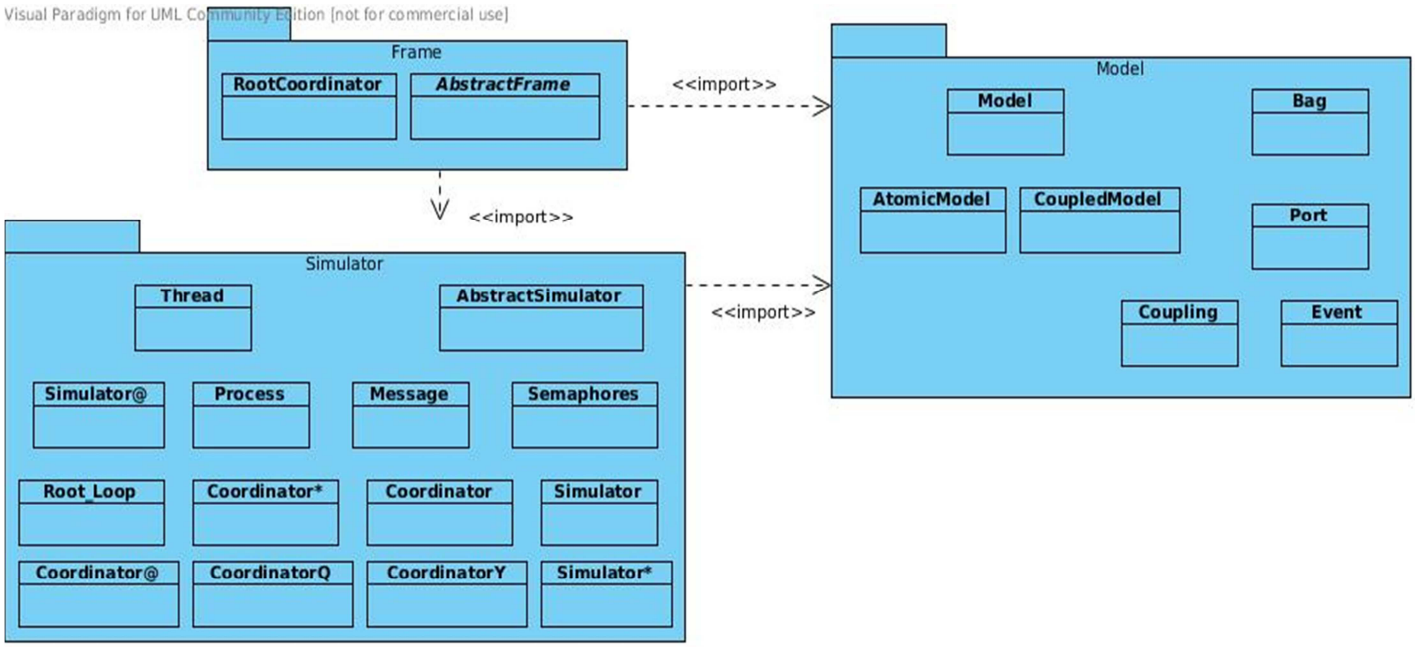


Figure 4.4: Package and Class View of Implementation II

This differs from first implementation because it makes use of multithreading. This can be seen in the new classes introduced in the Simulator Package which are:

- **Thread:** This super class provides the multi-threading facility for the TProcess class.
- **TProcess:** This is an abstract class containing attributes and methods common to its subclasses. The subclasses are used to treat the messages received in the simulator. They include
  - SimulatorQ, Simulator@, Simulator*
  - CoordinatorQ, CoordinatorY, Coordinator@, Coordinator*.
  - Root_Loop: Used by the RootCoordinator to process the starting and ending of the simulation process.
- **Semaphore:** Is used to synchronize the number of children messages are to be sent to and received from.

These subclasses treat these messages concurrently during the simulation. A new process is started and is contained in these classes.

4.2.4 IMPLEMENTATION III

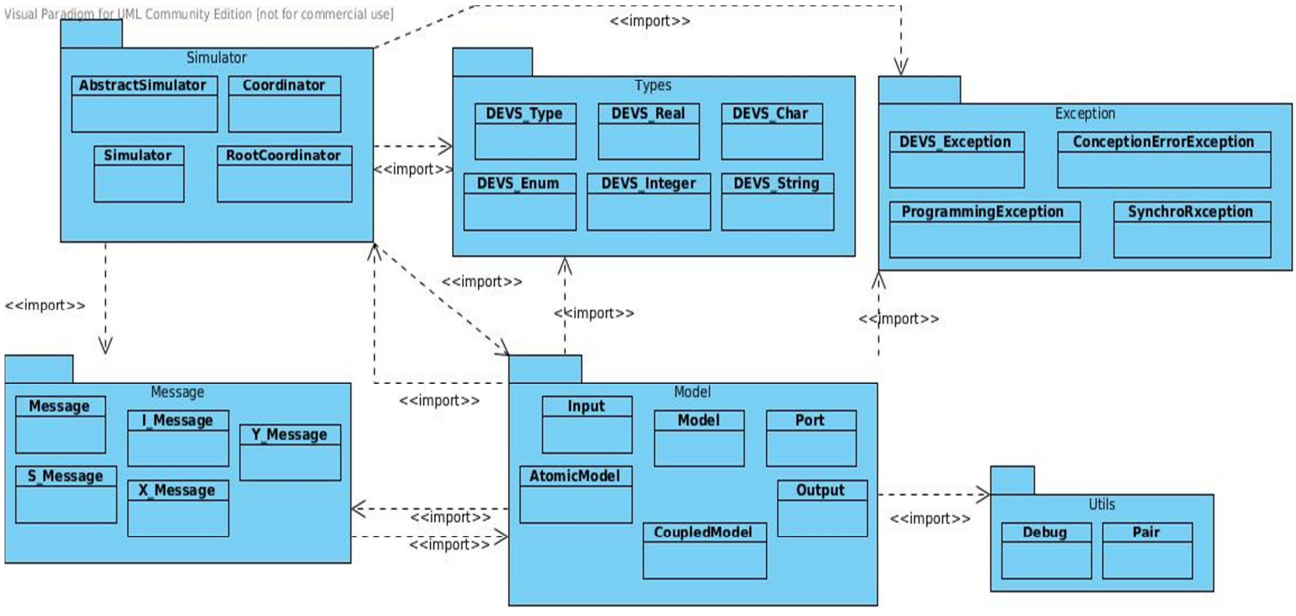


Figure 4.5: Original SIMSTUDIO1_1_1 (CDEVS) implementation

This implementation was prepared as a Java package (SIMSTUDIO_1_1) by Aminu [23] in his work. We redesigned it and made it more efficient by removing irrelevant classes and

linked components in our work. Also we made it an object oriented implementation of the CDEVS algorithm only.

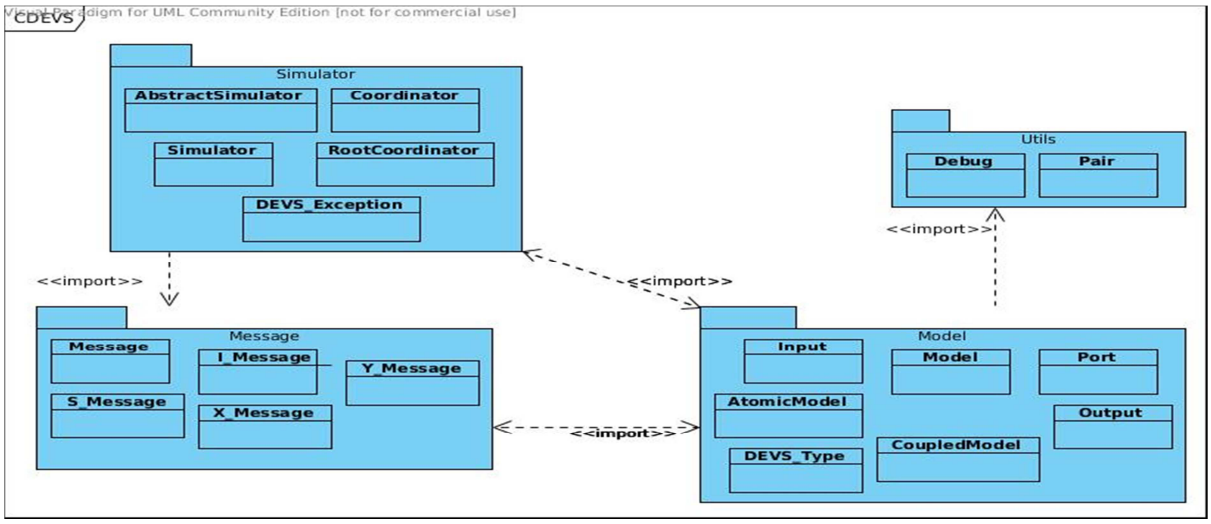


Figure 4.6: Package and Class View of Implementation III

This simulator package consists of 4 packages namely Simulator, Model, Message and Utils:

- **Simulator Package:** This package contains the components that run the simulators and coordinators in the system.
  - AbstractSimulator: This is an abstract class containing attributes and methods common to both the Simulator and Coordinator Classes.
  - Simulator: It controls the atomic models and is an AbstractSimulator
  - Coordinator: It controls the coupled models and is an AbstractSimulator
  - RootCoordinator: It manages the global clock and controls the execution of the simulators/coordinator hierarchy.
  - DEVS_Exception: This is an exception class. All the exceptions thrown in the Simulator package use this class.
- **Model Package:** contains descriptions required by a model and the model can either be coupled or atomic model.
  - Model: This is an abstract class containing attributes and methods common to both the AtomicModel and CoupledModel Classes.
  - AtomicModel: This contains information required from the modeler about the structure of the atomic model.
  - CoupledModel: This contains information required from the modeler about the structure of the couple model

- Input: It defines the structure of the input ports.
- Port: Defines the input and output ports of a model.
- Output: Defines the structure of the output ports.
- DEVS_Type: Defines the data structure of event a port should receive.

- **Message Package:** Contains information required to synchronize activity in the simulator.

  - I_Message: This class is used to initialize the simulator.
  - Message: Defines common specifications of the messages.
  - S_Message: Causes internal transition in the simulator.
  - X_Message: Causes external transition in the simulator.
  - Y_Message: Produces output.

- **Utils Package:** Contains classes that are used debugging the simulator (Debug) and coupling the ports in coupled models (Pair)

CHAPTER 5

CASE STUDY AND PERFORMANCE ANALYSIS

5.1 CASE STUDY: CLOUD GENERATION

Clouds are ever-changing feature of the outdoors. They are an integral factor in Earth's weather systems, and a strong indicator of weather patterns and changes which makes their effects and indications important to flight, weather forecasting among others. However the complexity of cloud formation and several other factors make real cloud simulation very difficult and expensive in real time.

Simulation is usually the tool of choice used to realize and analyze real world situations, in our case we used the DEVS formalism to develop a model of our case study and simulated it using the implementations discussed in previous chapters. Then we went ahead to do a performance analysis based on the execution time.

5.2 CLOUD FORMATION

Our simulation space was divided into grid corresponding to cells used in Cellular automaton. At each grid point, four state variables hum, cld, act, and ext are assigned and they represent the states; vapor, clouds, phase transition from vapor to clouds and clouds extinction respectively. The state of each variable is either 0 or 1. These values are generated randomly.

Their grid status (i, j, k) at time t+1 are calculated by the status at time t using the following transition rules [24].

Act (i, j, k, t+1) = .NOT.act (i, j, k, t).AND.hum(i, j, k, t).AND. fact(•)

Hum (i, j, k, t+1) = hum (i, j, k, t).AND. .NOT. Act (i, j, k, t)

Ext(i,j,k,t+1) = .NOT.ext(I ,j , k ,t). AND. Cld (i, j, k, t).AND. fext (•)

Cld (i, j, k, t+1) = Cld (i, j, k, t) .OR. Act (i, j, k, t)

Where, *fact* (•) is a Boolean function and its value is calculated by the status of Act around the grid.

fact (•)= Act(i+1, j, k, t).OR. Act (i-1, j, k, t).OR. Act (i, j+1, k, t).OR. Act (i, j-1, k, t).OR. Act (i, j , k+1,t).OR. Act (i, j, k-1, t)).OR. Act (i+2, j, k, t).OR. Act (i-2, j, k, t) .OR. Act (i, j+2, k, t).OR. Act (i, j-2, k, t).OR. Act (i, j, k-2, t)

And *fext*(•) is a Boolean function similar to *fact* and its value is calculated by the state of ext around the grid.

fext (•) = Ext(i+1, j, k, t).OR. Ext (i-1, j, k, t).OR. Ext (i, j+1, k, t).OR. Ext (i, j-1, k, t).OR. Ext (i, j, k+1,t).OR. Ext (i, j, k-1, t)).OR. Ext (i+2, j, k, t). OR. Ext (i-2, j, k, t) .OR. Ext (i, j+2, k, t).OR. Ext (i, j-2, k, t).OR. Ext (i, j, k-2, t)

5.3 MODEL STRUCTURE

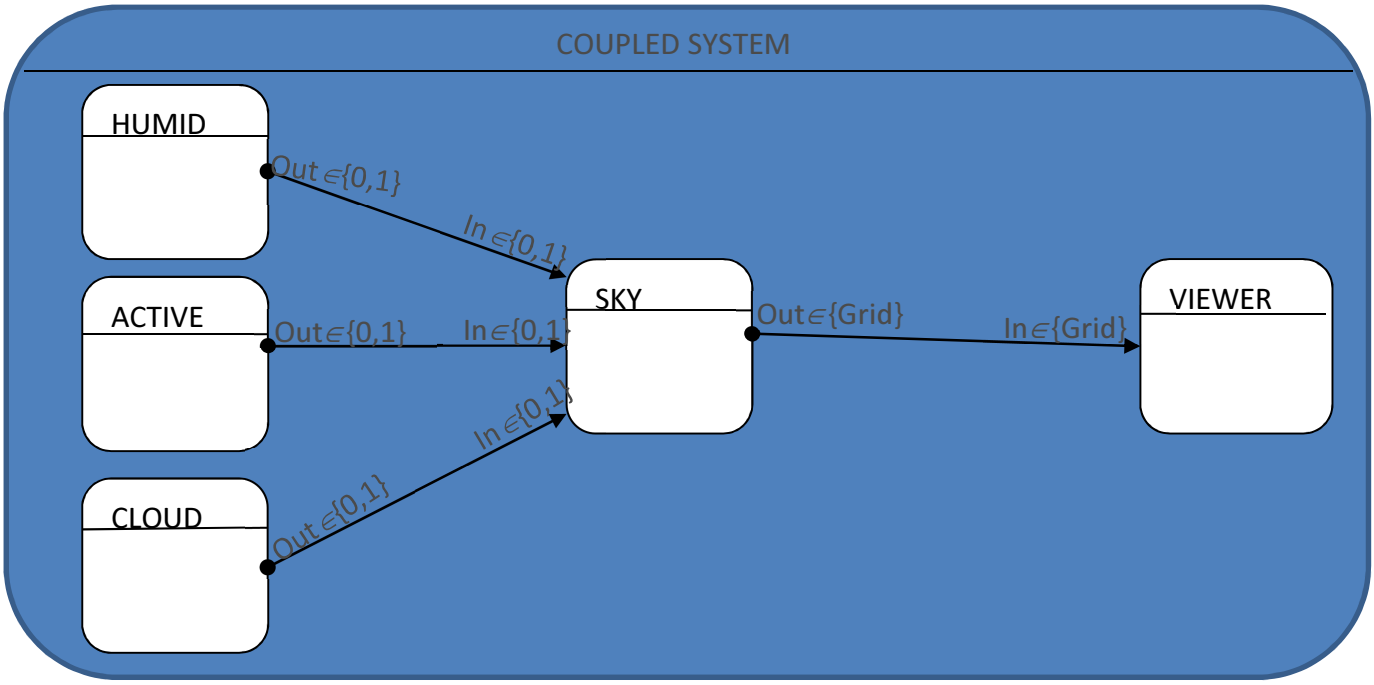


Figure 5.1: Cloud Generation System

The entire structure consists of 5 atomic models namely Humid, Active, Cloud, Sky, Viewer and a coupled model CoupledSystem. The atomic models are sub models of the coupled. Humid, Active, Cloud generate the humid, active and cloudy states respectively while in the Sky model the transition rules are applied and the results are sent to the Viewer to be displayed.

5.3.1 HIERARCHICAL STRUCTURE OF THE SYSTEM

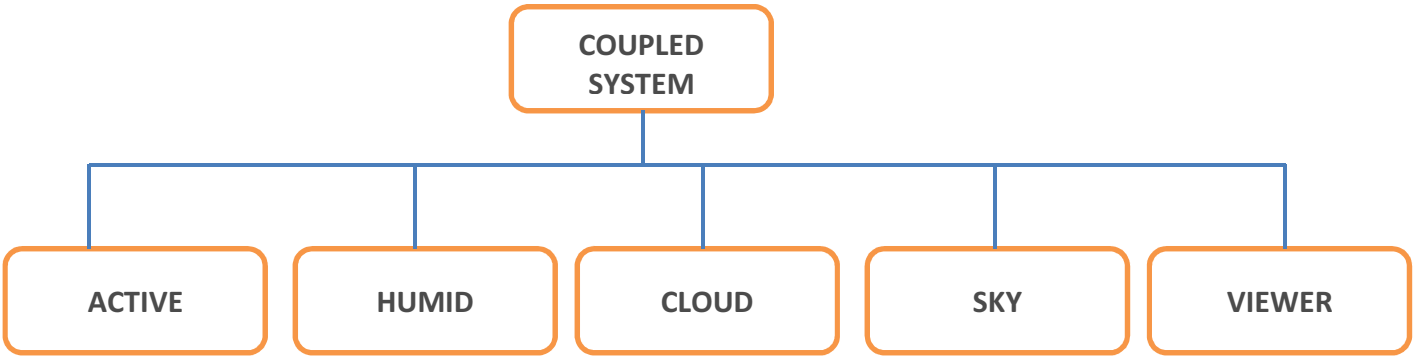


Figure 5.2: Model Hierarchy

5.3.2 ATOMIC MODELS

According to the DEVS atomic model we made the following specifications;

$$M = < X, Y, S, ta, \delta ext, \delta int, \delta conf, \lambda >$$

	X	*Y*	*S*	*Ta*	*δext*	*δint*	*δconf*	*λ*
ACTIVE	-	act	{0,1}	1	-	-	-	Act ∈ {0,1}
HUMID	-	hum	{0,1}	1	-	-	-	Hum∈{0,1}
CLOUD	-	cld	{0,1}	1	-	-	-	Cld ∈ {0,1}
SKY	Act, hum, cld	sky	Active, Humid, Cloudy, Cloudless	1	Active, Humid, Cloudy, Cloudless	Active, Humid, Cloudy, Cloudless	δint	Grid
VIEWER	sky	-	Display grid	Infinity	Display grid	-	δext	-

5.3.3 COUPLED MODEL

The DEVS model specification for the CoupledSystem is given as

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

Where

X and ***Y*** are empty since the model has no input and output port

D is the model itself which is the CoupledSystem

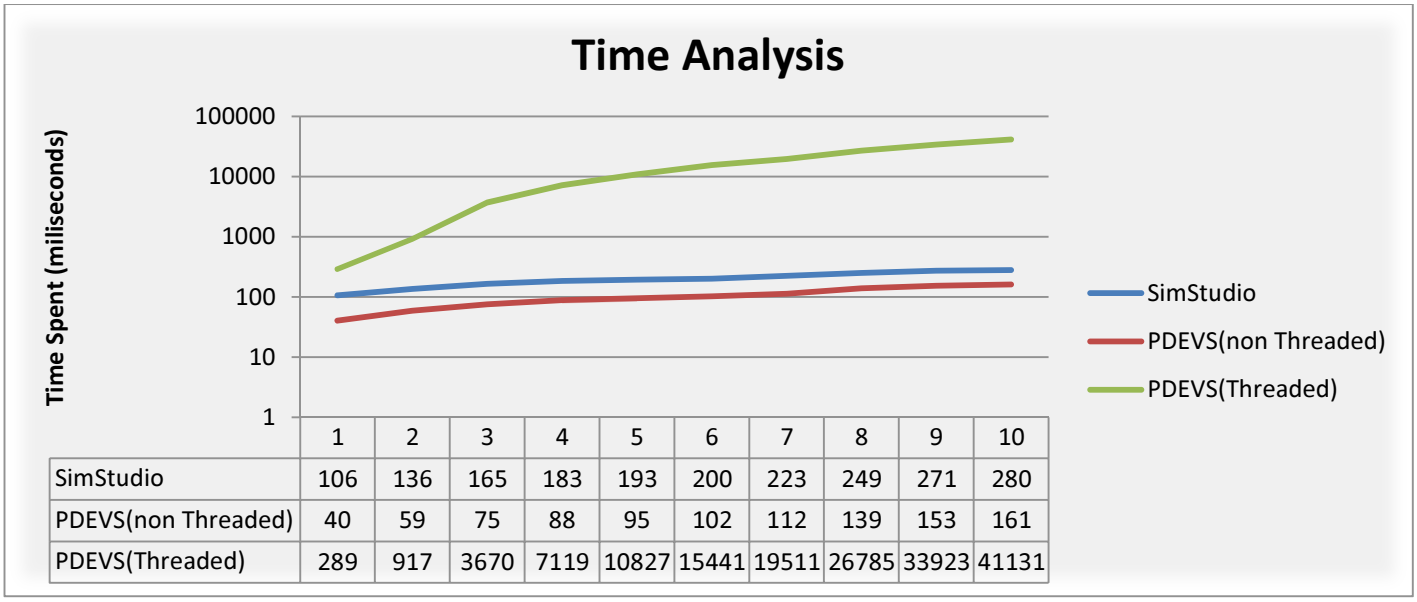
$d \in D$ are the sub components of the coupled model and they are the Active, Humid, Sky, Cloud and Viewer models.

EIC and ***EOC*** are empty since there are no ports from the input of the coupled model to its sub-models or from the output of any sub-model to it.

IC is {Active.out, Sky.in}, {Humid.out, Sky.in}, {Cloud.out, Sky.in} and {Sky.out, Viewer.in}

5.4 PERFORMANCE ANALYSIS

The comparison was done between the CDEVS and PDEVS simulators: SimStudio1_1_1, threaded and non-threaded PDEVS simulators. However during the time analysis, we repeated the simulation 10 times and at each time the limit was set to 10 unit time and as well we kept track of the simulation at each time. To analyze each of the simulators we calculated the elapsed time which was measured in milliseconds and plotted the graph. This was done using a uniprocessor.



	1	2	3	4	5	6	7	8	9	10
SimStudio	106	136	165	183	193	200	223	249	271	280
PDEVS(non Threaded)	40	59	75	88	95	102	112	139	153	161
PDEVS(Threaded)	289	917	3670	7119	10827	15441	19511	26785	33923	41131

Figure 5.3: Time Analysis Graph

However, in this analysis it was observed that the average elapsed time spent in the SimStudio1_1_1, threaded and non-threaded PDEVS simulators were 0.1, 0.2 and 15 seconds respectively. There was an improvement in the time spent in the non-threaded PDEVS simulator when compared to the CDEVS simulator thus making it more efficient.

Exploiting parallelism was made possible in the parallel implementation, PDEVS (threaded) formalism. During the analysis it was observed that each thread consumed more memory and a large amount of additional computation time as the number of simulation runs increased. Though we expected a reduction in time for this simulator due to threads but the result of the analysis may have been affected as well by the choice of model used for testing and the amount of messages that were exchanged in the simulator during simulation.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

This work discussed different implementation techniques for the Classic and Parallel DEVS in an effort to address the need for hierarchical construction of models as well as the need for efficient and fast execution of these models using parallel simulation.

Using a model to test each implementation, we have shown that; the PDEVS (non-threaded) implementation out-performs the CDEVS and the PDEVS (threaded) implementation slowed down execution time and consumed more computer memory than the other implementations presented in this work.

6.2 FUTURE WORK

There are several areas of interest for future research which include:

- Proper management of threads in the PDEVS (threaded) implementation to improve efficiency and performance.
- There is a need to look into having flat simulation mechanism for PDEVS as opposed to hierarchical in the simulator: This is for comparison reasons in terms of the number of messages exchanged and the amount of communication overheads.
- Further analysis of the simulator performance on a multiprocessor architecture. The performance analysis presented in this work was carried out on a single processor machine.
- The development of a Graphics User Interface (GUI) for the simulator: The creation of models using the DEVS formalism is not domain specific; therefore GUI would make it easier for the modeler to specify his models.
- A DEVS testing tool: This can be dedicated to analyzing and comparing the performance of various DEVS simulators and can also include the verification and validation of the created models.

REFERENCES

[1] Glinsky, E. *New Techniques for Parallel Simulation of Devs and Cell-Devs Models in Cd++*, Master's Thesis, Carleton University Ottawa, Canada. 2004

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

[3] Wainer, G. A. *Discrete-Event Modeling and Simulation: A practitioner's Approach.* New York: CRC Press 2009.

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

[5] Wikipedia. *Discrete Event Simulation.* http://en.wikipedia.org/wiki/Discrete_event_simulation.

[6] Zeigler, B. P. *Theory of modeling and simulation*. New York: Wiley-Interscience. 1976.

[7] Zeigler, B. P.; Sarjoughian, H. S. *Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-Based Simulation Models (draft version 3).* Arizona, USA. January 2005.

[8] Zeigler, B. P. *Multifaceted Modeling and Discrete Event Simulation*. New York: Academic Press, 1984.

[9] Wikipedia. *DEVS.* http://en.wikipedia.org/wiki/DEVS.

[10] 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.

[11] Bonaventura, M.; Wainer, G.; Castro, R. *Advanced IDE for Modeling and Simulation of Discrete Event Systems.* Proceedings of the Symposium on Theory of Modeling and Simulation. 245-252 Florida, USA. 2010.

[12] Nutaro, J. *ADEVS.* http://www.ornl.gov/~1qn/adevs/index.html

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

[14] Sarjoughian, H.S.; Zeigler, B.P. *DEVSJAVA: Basis for a DEVS-based collaborative M&S environment.* Proceedings of the International Conference on Web-Based Modeling and Simulation. vol. 5, pp. 2936. San Diego, CA. USA. 1998.

[15] Traoré, M.K. *SimStudio. A Next Generation Modeling and Simulation Framework.* Proceedings of SIMUTools. Marseille, France. 2008.

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

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

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

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

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

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

[21] Message Passing Interface Forum. *MPI: A Message-Passing Interface standard (version 1.1). Technical report.* http://www.mpi-forum.org.

[22] Unified Modeling Language (UML). http://www.uml.org/.

[23] Aminu, M. *Approach to a Simulation Virtual Machine: Object Oriented Implementation of DEVS and PDEVS.* Master's Thesis, African University of Science and Technology, Nigeria, December 2009.

[24] Yoshinori, D.; Tomoyuki, N.; Tsuyoshi. O. *Animation Of Clouds Using Cellular Automaton.* Proceedings of the Computer Graphics and Imagery'99. 1999.

APPENDIX

USER MANUAL: HOW TO PLUG-IN MODEL SPECIFICATIONS INTO THE SIMULATOR

This section contains instructions on how to specify the models in Java programming language and connect them to the simulators.

PDEVS SIMULATORS

This is works for both threaded and non-threaded PDEVS simulators

Atomic Models can be defined as

```
//Import the Model Package in the simulator.
import model.*;

//Give the name of the atomic model to be simulated and extend the AtomicModel class
public class MySubModel extends AtomicModel {

    //Declare state variables
    Port portName;

    //Define the constructor:
    //      1. Call the constructor of the super class
    //      2. Define input and output ports
    //      3. Initialize state variables
    public MySubModel() {
        super();
        PortName = new Port();
        addInputPort(Port portName, String label);
        addOutputPort(Port portName, String label);
    }

    //Override the internal transition function of the super class
    //Define the function
    @Override
    public void deltaInt() {
        //
```

```
        }

        //Override the output function of the super class
        //Define the function
        //Set the output port(s) to receive an event
        @Override
        public void lambda() {
                portName.addEvent(Event type);
        }

        //Override the external transition function of the super class
        //Define the function and set it to accept an argument
        @Override
        public void deltaExt(double e) {
                //
        }

        //Override the time advance function of the super class
        //Define the function
        //Set it to return the time interval between two transitions
        @Override
        public double ta() {
                return sigma;
        }

        //Override the confluent transition function of the super class
        //Define the function
        @Override
        public double deltaconf() {
                //
        }
}
```

For **Coupled Models**

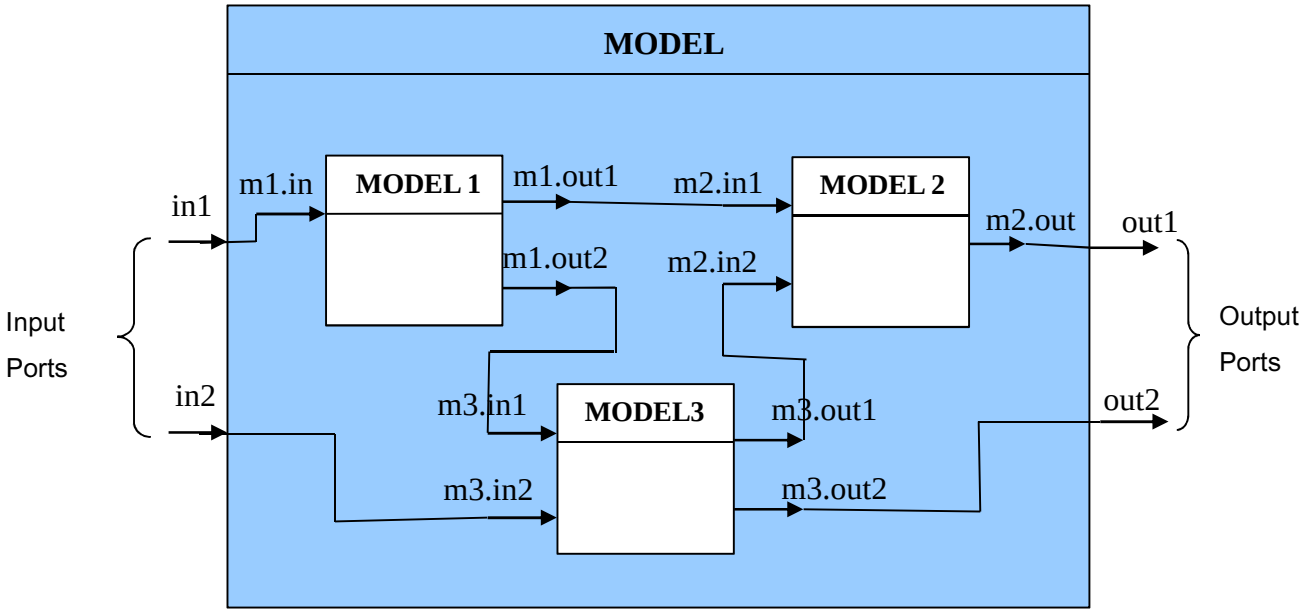


Figure I: Sample Model

The Figure above will be used to describe the Coupled Model specification

```
//Import the Model Package in the Simulator
import model.*;

//Give the name of the coupled model to be simulated and extend the CoupledModel
class
public class MyModel extends CoupledModel {

//Declare atomic models
        mySubModel model;

//Define the constructor:
        //      1. Call the constructor of the super class
        //      2. Initialize the atomic models
        //      3. Add each atomic model to the coupled model
        //      4. Define the couplings
        public MyModel() {
                super();
                model = new mySubModel();
                addSubModel(model);
```

```
            addIC(Model1.getOutputPort("m1.out1"),Model2.getInputPortStructure("
            m2.in1"));
            addEIC(Model.getInputPort("in1"),Model1.getInputPortStructure("m1.in1
            "));
            addEOC(Model2.getOutputPort("m2.out"),Model.getIOutputPortStructur
            e("out1"));
        }
}
```

START THE SIMULATOR

To start the simulator a new frame class should be created and the condition to end the simulation should be set.

```
//Import the Frame and Model Packages from the simulator
import frame.*;
import model.*;

//Create a new class and extend the AbstractFrame class in the package
public class MyFrame extends AbstractFrame{

//Define the constructor
//Initialize the constructor of the super class with an instance of the coupled model
    public MyFrame(Model m) {
        super(m);
    }

//Override the ending condition function of the super class
// Set the condition for which the simulation should end
    @Override
    public boolean endingCondition() {
        //
    }
}
```

```
//Create a new class
public class Simulation{

//Declare the instances of coupled model and abstractframe classes
        MyModel sys;
        MyFrame frame;

//Define the constructor
//Initialize instances of the classes declared
        public Simulation() {
                sys = new MyModel();
                frame = new MyFrame(sys);
        }

//Create a Main method.
//      1. Create and initialize an object of the Simulation class
//      2. Use it to set the initial time
//      3. Use it to run the program
        public static void main(String args[]) {
                Simulation sim = new Simulation();
                sim.frame.setInitialTime(0);
                sim.frame.runExperiment();
        }
}
```

CDEVS SIMULATOR: SimStudio

Atomic Models can be defined as

```
//Import the necessary packages in the SimStudio.
import model.*;
import types.*;
import exception.*;
```

```
//Give the name of the atomic model to be simulated and extend the AtomicModel
//class
public class MySubModel extends AtomicModel {

//Declare state variables
        Int sigma;

//Define the constructor:
//      1. Call the constructor of the super class
//      2. Define input and output ports
//      3. Initialize state variables
// Other types of data-structures can be used in place of DEVS_Type.

        public MySubModel(String name, String desc) {
                super(name, desc);
                sigma = 1;
                addInputPortStructure(DEVS_Type type, String name, String desc);
                addOutputPortStructure(DEVS_Type type, String nam, String desc);
        }

        //Override the internal transition function of the super class
        //Define the function
        @Override
        public void deltaInt() {
                //
        }

        //Override the output function of the super class
        //Define the function
        //Set the output port(s) to receive an event
        @Override
        public void lambda() {
                setOutputPortData(String name, DEVS_Type type);
        }
```

```
        //Override the external transition function of the super class
        //Define the function and set it to accept an argument
        @Override
        public void deltaExt(double e) throws DEVS_Exception {
                //
        }

        //Override the time advance function of the super class
        //Define the function
        //Set it to return the time interval between two transitions
        @Override
        public double ta() {
                return sigma;
        }
}
```

For **Coupled Models**

```
        //Import the necessary packages in the SimStudio.
        import model.*;

        //Give the name of the coupled model and extend the CoupledModel class
        public class MyModel extends CoupledModel {

        //Declare atomic models
                mySubModel model;

        //Define the constructor:
                //      1. Call the constructor of the super class
                //      2. Initialize the atomic models
                //      3. Add each atomic model to the coupled model
                //      4. Define the couplings
```

```
        public MyModel(String name, String desc) {
                super(name, desc);
                model = new mySubModel("ModelName", "description");
                addSubModel(model);
                addIC(Model1.getOutputPortStructure("m1.out1"),Model2.getInputPo
                rtStructure("m2.in1"));
                addEIC(Model.getInputPortStructure("in1"),Model1.getInputPortStru
                cture("m1.in1"));
                addEOC(Model2.getOutputPortStructure("m2.out"),Model.getIOutput
                PortStructure("out1"));
        }

        //Override the select function of the super class
        //Define the function and set it to accept an argument
        public Model select(ArrayList<Model> arg0) {
                //
        }
}
```

START THE SIMULATOR

To start the simulator a new class has to be created

```
//Import the necessary packages in the SimStudio.
import simulator.*;
import exception.*;

//Give the name of the new class
public class Simulation {

//Create a Main method with an exception
//      1. Create and initialize an object of the Coupled Model
//      2. Create and initialize an object of RootCoordinator class with the simulator
//      associated to the coupled model
//      3. Set the initial time
//      4. Set the number of times the simulation should run
```

```
        public static void main(String args[]) throws DEVS_Exception {
                CoupledModel coupledModel;
                RootCoordinator root = new RootCoordinator(coupledModel.getSimulator());
                root.init(startTime);
                sim.root.run(value);
        }
                // The startTime is the time in which the simulation should start from while
                //value represents the number of times the simulation should run.
}
```