FORMAL AND OPERATIONAL STUDY OF P-DEVS

A THESIS

SUBMITTED TO THE AFRICAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

ABUJA-NIGERIA

IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR

MASTER DEGREE IN COMPUTER SCIENCE & ENGINEERING

By

ELI – AKE GRACE EYITAYO KEHINDE

Supervisor

PROFESSOR MAMADOU KABA TRAORE

African University of Science and Technology

Abuja

Knowledge is Freedom

December 2011.

FORMAL AND OPERATIONAL STUDY OF P-DEVS

By

ELI – AKE GRACE EYITAYO KEHINDE

ABSTRACT

Discrete Event System Specification (DEVS) is a sound formal modeling and simulation (M&S) structure based on generic dynamic system concepts. PDEVS (Parallel Discrete Event System Specification) is a well-known formalism for the specification of complex concurrent systems organized as an interconnection of atomic and coupled interacting components. The abstract simulator of a PDEVS model is normally founded on the assumption of maximal parallelism: multiple components are allowed to undertake at the same time an independent state transition. Our work is to study PDEVS formalism, its operational semantics through various implementation strategies, the cleaning of the thread-less and the threaded implementations proposed in the PDEVS simulation engine, benchmarking of the two implementations and formal analysis of the simulation protocol.

Keywords: Discrete Event System Specification, Modelling and Simulation, Formal Analysis

DEDICATION

To the almighty God.

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

To my parents and Twin.

ACKNOWLEDGEMENT

To God be the glory for the wonderful things he has done in my life.

I will like to express my sincere gratitude to my supervisor, Prof. M.K. Traoré, for his guidance, support and encouragement.

I am also grateful to Prof. A. B. Abdallah, Prof. Soboyejo, Dr. Ekpe Okorafor, and Dr. Guy Degla for their encouragement.

I am thankful to my dearest Husband for all his support. Many thanks to my family the ELI – AKE's: Daddy, Mummy, Taiye, Idowu, Ayorinde and friends: Bright, Leke, Ireti, Doyin for their love, care and support.

Finally I would like to appreciate my course mates, all members of the AUST Community, staff and students.

God bless you all.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Introduction to Modeling and Simulation

A **computer simulation** is a computer program, or network of computers, that attempts to generate the behaviour of an abstract model of a particular system. Computer simulations have become a useful part of mathematical modeling of many natural systems in computational physics, astrophysics, chemistry and biology, human systems in economics, psychology, social science, and engineering. Simulations can be used to explore and gain new insights into new technology, and to estimate the performance of systems too complex for analytical solutions.

Computer simulations vary from computer programs that run a few minutes, to network-based groups of computers running for hours, to ongoing simulations that run for days. The scale of events being simulated by computer simulations has far exceeded anything possible using the traditional paper-and-pencil mathematical modeling.

Modeling and simulation (M&S) is the use of models, including emulators, prototypes, simulators, and stimulators, either statically or over time, to develop data as a basis for making managerial or technical decisions. The use of modeling and simulation (M&S) within engineering is well recognized. Simulation technology belongs to the tool set of engineers of all application domains and has been included into the body of knowledge of engineering management. M&S has already helped to reduce costs and increase the quality of products and systems.

Modeling and Simulation is a discipline for developing a level of understanding of the interaction of the parts of a system, and of the system as a whole. The level of understanding which may be developed via this discipline is seldom achievable via any other discipline.

A **system** is understood to be an entity which maintains its existence through the interaction of its parts. **A model** is a simplified representation of the actual system intended to promote understanding. Whether a model is a good model or not depends on the extent to which it promotes understanding. Since all models are simplifications of reality there is always a trade-off as to what level of detail is included in the model. If too little detail is included in the model one runs the risk of missing relevant interactions and the resultant model does not promote understanding. If too much detail is included in the model, the model may become overly complicated and actually preclude the development of understanding. One simply cannot develop all models in the context of the entire universe.

A simulation generally refers to a computerized version of the model which is run over time to study the implications of the defined interactions. Simulations are generally iterative in their development. One develops a model, simulates it, learns from the simulation, revises the model, and continues the iterations until an adequate level of understanding is obtained.

1.1.1 Modeling and Simulation Concepts

Basic concepts

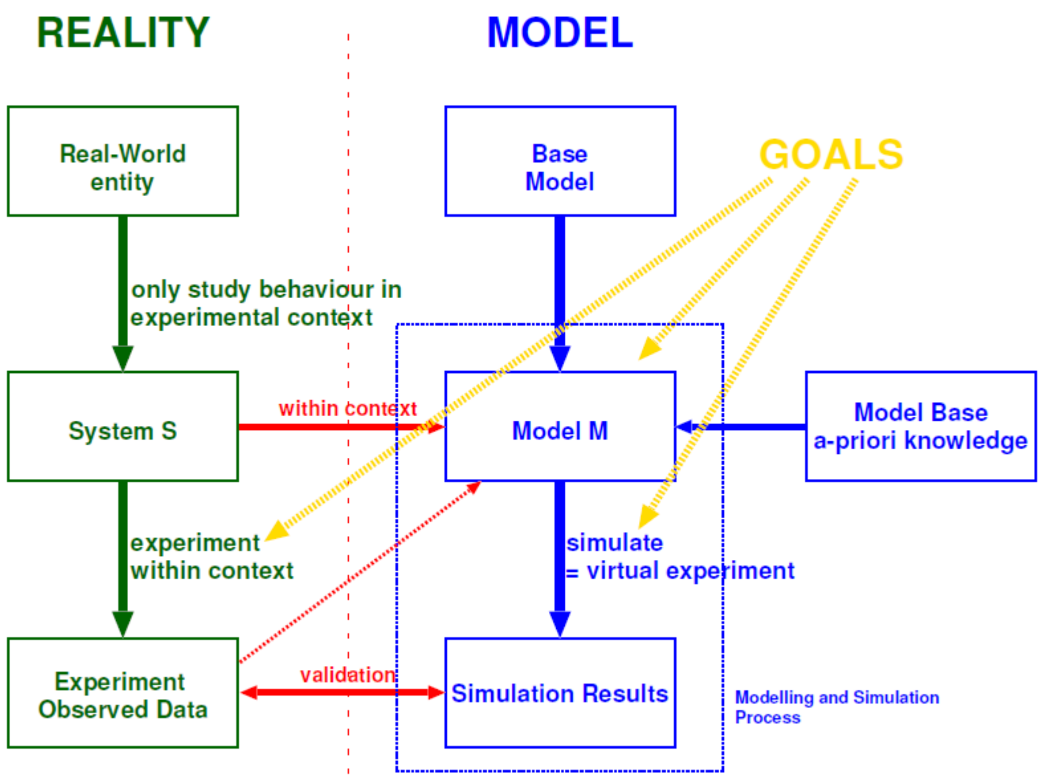

REALITY
MODEL
Real-World entity
Base Model
GOALS
only study behaviour in experimental context
System S
within context
Model M
Model Base a-priori knowledge
experiment within context
simulate = virtual experiment
Experiment Observed Data
validation
Simulation Results
Modelling and Simulation Process

Figure 1: Modeling and Simulation

The basic concepts of modeling and simulation are given in Figure1 above as introduced by Zeigler [Zei84, ZPK00]. Object is some entity in the Real World. Such an object can exhibit widely varying behaviour depending on the context in which it is studied, as well as the aspects of its behaviour which are under study. Base Model is a hypothetical, abstract representation of the object's properties, in particular, its behaviour, which is valid in all possible contexts, and describes all the object's facets. A base model is hypothetical as we will never —in practice— be able to construct/represent such a “total” model. The question whether a base model exists at all is a philosophical one.

System is a well defined object in the Real World under specific conditions, only considering

specific aspects of its structure and behaviour.
When one studies a system in the real world, the experimental frame (EF) describes experimental conditions (context), aspects, within which that system and corresponding models will be used. As such, the Experimental Frame reflects the objectives of the experimenter who performs experiments on a real system or, through simulation, on a model. Verification is concerned with the correctness of the transformation from some intermediate abstract representation (the conceptual model) to the program code (the simulation model) ensuring that the program code faithfully reflects the behaviour that is implicit in the specification of the conceptual model.

1.1.2 Modeling and Simulation Benefits

There are two major benefits to performing a simulation rather than actually building the design and testing it. The biggest of these is money. Designing, building, testing, redesigning, rebuilding, retesting of anything can be an expensive project. Simulations take the building/rebuilding phase out of the loop by using the model already created in the design phase. Most of the time, the simulation testing is cheaper and faster than performing the multiple tests of the design each time. The second benefit is that a simulation can give you result that are not experimentally measurable with our current level of technology. Simulation can be set to run for as many time as one desire and at any level of detail desired. Other Modeling and Simulation benefits includes:

- Exploring new design options without disrupting existing systems
- Testing new hardware, transportation systems, etc, without investing resources for their acquisition
- Time scaling can be compressed (for slow moving systems) or expanded (for fast moving systems)
- Internal variables can be made observable
- Sensitivity and interaction of variables can be studied to understand their impact on the system behavior
- Bottleneck analysis can be performed.

1.1.3 Modeling and Simulation Importance

Nowadays, technology has enabled people to accomplish things that would have been impossible a few decades ago. Computer simulations have played an important part in shaping the world that people now live in. Computer simulations, also referred to as computational models or computer models, are basically an attempt to simulate a digital representation of a system using computers as well as computer programs. While computer modeling and simulations have many applications across various fields and industries, they play an important part in constructing mathematical models that can be used in engineering, social science, chemistry, astrophysics, physics, as well as economics. Computer modeling and simulation:

1. Allows experts to analyze systems before they are even built or applied. This allows them to come up with new innovations that can further improve the current levels of knowledge and technology. There are many different types of computer models and simulations depending on the software and hardware used for the process. Some models are made using a computer running basic modeling software while others use powerful and complex computer networks to run simulations that can last up to days or weeks. In the years before the advent of computing, people had to run simulations using pens and paper for mathematical modeling. However, this type of modeling was very limited and could only handle relatively simple models and simulations.
2. Computer modeling far exceeds the capabilities of pen and paper-based modeling, allowing experts to simulate very complex systems with relative ease. These days, it is possible to create computer models and simulations of complex battles involving thousands of units of infantry, armoured units as well as air- and water-based units. It is also now possible to create complex models of atoms, molecules and chemicals.
3. Computer modeling can be used for almost every single aspect of human life. They are especially important to the fields of business planning, medicine and science and technology.

1.1.4 Modeling and Simulation Challenges

The challenges of modeling and simulation includes:

1. Model building may require special training
2. Many people do not consider what they do engineering unless they can see, hear, feel, and taste the project.
3. Simulation packages may be expensive

4. Learning curve to use simulation packages may be longer than the time available
5. Closed form analysis may be possible
6. Results may be difficult to interpret

1.2 Introduction to DEVS formalism and its variants

The DEVS formalism provides a hierarchical and modular modeling mechanism, which tends itself to reuse and interoperability. The DEVS formalism allows the rigorous description of complex dynamic systems. Its main advantages are the definition of component-based models and the efficient simulation algorithms for these models.

DEVS formalism is a well known for modeling and simulation for discrete-event systems. Some of the advantages of the DEVS formalism are that it allows the hierarchical description of systems, that it provides natural ways for modular design and implementation of systems, and that there are efficient algorithms for their simulation. The basic DEVS formalism is also called Classic DEVS which has some limitations for parallel implementation. For example, the select function used in Classic DEVS coupled model for collision tie-breaking, is less controllable as the tie-breaking decision can only be made in the global level.
Parallel DEVS, as an extension to Classic DEVS, which eliminates the select function in coupled model and introduces the confluent function in atomic model, gives the modeler complete control over the collision behavior. Parallel DEVS also uses bags as the message structure. This allows that inputs of a component arrive in any order and that more than one input with the same identity may arrive from one or more sources. In this work, the DEVS formalism that we meta - modelled is the Parallel DEVS.

A DEVS model is either atomic or coupled. **An atomic model** describes a simple system. **A coupled model** is the composition of several submodels which can be atomic or coupled. Submodels have ports, which are connected by channels. Ports have a type: they are either input or output ports. Ports and channels allow a model to receive and send signals from and to other models respectively. A channel must go from an output port of some model to an input port of a different model, from an input port in a coupled model to an input port of one of its submodels, or from an output port of a submodel to an output port of its parent model. An atomic model has, in addition to ports, a set of states, one of which is the initial state, and two types of transitions between states: internal and external. Associated with each state is a

time-advance and an output. An atomic model allows to specify the behavior of a basic element of a given system. Connections between different atomic models can be performed by a Coupled Model (CM) (Zeigler 1976, Zeigler 1984):
A coupled model, tells how to couple (connect) several component models together to form a new model. This latter model can itself be employed as a component in a larger coupled model, thus giving rise to hierarchical construction. A simulator is associated with the DEVS formalism in order to exercise coupled model's instructions to actually generate its behavior. The architecture of a DEVS simulation system is derived from the abstract simulator concepts (Zeigler, 1990) associated with the hierarchical and modular DEVS formalism. We talk more on DEVS formalism in the next chapter.

1.3 The need for formal analysis of DEVS simulation protocol

Formal methods use mathematics to prove that software design models meet their requirements that can greatly increase confidence in the safety and correctness of software. Recent advances in formal analysis tools have made it practical to formally verify important properties of these models to ensure that design defects are identified and corrected early in the lifecycle. This paper describes how formal method tool can be inserted into a software to prove its property of correctness.
In computer science and software engineering,formal methods are mathematically-based techniques for the specification, development
and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design. However, the high cost of using formal methods' means that they are usually only used in the development of high-integrity systems, where safety or security is of utmost importance.

1.4 Structure of thesis report

The Chapter 1 is the introduction to modeling and simulation. Chapter 2 describes DEVS. Chapter 3 presents the literature review on PDEVS implementation. In Chapter 4, we present formal methods, in Chapter 5 we present the Simstudio and formal methods and finally in Chapter 6 (conclusion), we present the summary of the work, challenges and future work.

Chapter 2: Discrete Event System Specification (DEVS)

2.1 Discrete Event System Specification (DEVS)

Discrete event simulation utilizes a mathematical/logical model of a physical system that represents state changes at precise points in simulated time. Both the nature of the state change and the time at which the change occurs mandates precise description. Customers waiting for service, the management of parts inventory or military combat are typical domains of discrete event simulation.

Some components of a Discrete-Event Simulation

In addition to the representation of system state variables and the logic of what happens when system events occur, discrete event simulations include the following:

- **Clock:** The simulation must keep track of the current simulation time, in whatever measurement units are suitable for the system being modeled, and because events are instantaneous the clock skips to the next event start time as the simulation proceeds.

- **Events List:** The simulation maintains at least one list of simulation events. This is sometimes called the pending event set because it lists events that are pending as a result of a previously simulated event but are yet to be simulated themselves. 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. It is common for the event code to be parameterized, in which case, the event description also contains parameters to the event code.

- **Random-Number Generators:** The simulation needs to generate random variables of various kinds, depending on the system model.

- **Statistics:** The simulation typically keeps track of the system's statistics, which quantify the aspects of interest.

- **Ending Condition:** Because events are bootstrapped, 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”.

Discrete Event System Specification (DEVS) 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.
DEVS is a timed event system, a formalism for modeling and analysis of discrete event systems (DESs). The DEVS formalism was invented by Dr. Bernard P. Zeigler, who is a professor at the University of Arizona. DEVS was introduced to the public in Zeigler's first book, *Theory of Modeling and Simulation*, in 1976, while Zeigler was an associate professor at University of Michigan. DEVS can be seen as an extension of the Moore machine formalism, 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 a lifespan with each state [Zeigler76],
2. providing a hierarchical concept with an operation, called *coupling* [Zeigler84].

Since the lifespan of each state is a real number (more precisely, non-negative real) or infinity, it is distinguished from discrete time systems, sequential machines, and Moore machines, in which time is determined by a tick time multiplied by non-negative integers. Moreover, the lifespan can be a random variable; for example the lifespan of a given state can be distributed exponentially or uniformly. The state transition and output functions of DEVS can also be stochastic.

Zeigler proposed a hierarchical algorithm for DEVS model simulation in 1984 [Zeigler84] which was published in *Simulation* journal in 1987. Since then, many extended formalism 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 continuous 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. In addition to its extensions, there are some subclasses such as SP-DEVS and FD-DEVS have been researched for achieving decidability of system properties.

Due to the modular and hierarchical modeling views, as well as its simulation-based analysis capability, the 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 .

DEVS defines system behavior as well as system structure. System behavior in DEVS formalism is described using input and output events as well as states (Wikipedia 2011).

Parallel DEVS is the formalism chosen as the foundation for this work. P-DEVS models are described very much like CDEVS models, on the basis of an atomic model which describes a simple system and a coupled model that is composed of several sub-models which can be atomic or coupled. P-DEVS (Chow & Zeigler, 2010) is a formalism that has been defined to exploit the inherent parallelism of the DEVS formalism (Zeigler, 1976). While the simulation algorithms are well defined, their implementation is a challenge due to both correctness issue and efficiency issue.

2.2 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. Classic DEVS was the first version to be developed and after some fifteen years, it successor was introduced as Parallel DEVS. As we will explain later, Parallel DEVS removes constraints that originated with the sequential operation of early computers and hindered the exploitation of parallelism, a critical element in more modern computing. In the classic DEVS formalism, Atomic DEVS captures the system behavior, while Coupled DEVS describes the structure of system.

Classic DEVS (CDEVS) Atomic Model

Classic DEVS System Specification

A Discrete Event System Specification (DEVS) is a structure

M = <X_M , Y_M , S, □ext , □int, □con, □, ta>

Where:

- X is the set of inputs
- S is a set of states
- Y is the set of outputs
- □ext: Q x X_M^b□□S is the external state transition function;
- □int: S □□S is the internal state transition function;
- □con: Q x X_M^b□□S is the confluent transition function;
- □□: S □□Y_Mb is the output function;
- ta : S □□R_0^+□□□is the time advance function; with Q = {(s, e) | s □□S , 0 □□e □□ta(s)} the set of total states.

The interpretation of these elements is illustrated in Figure 2.1. 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*). Notice that *ta*(*s*) could be a real number as one would expect. But it can also take on the values 0 and ∞. In the first case, the stay in state *S* is so short that no external events can intervene – we say that *S* is a *transitory* state. In the second case, the system will stay in *S* forever unless an external event interrupts its slumber - we say that *S* is *a passive* state in this case. When the resting time expires, i.e., when the elapsed time, $e = ta\ (s)$, the system outputs the value, $\lambda\ (s)$, and changes to state δint (*s*). Note output is only possible just before internal transitions. If an external event $x \in X$ occurs before this expiration time, i.e., when the system is in total state (*s*, *e*) with $e \leq ta(s)$, the system changes to state δext (*s, e, x*). Thus the internal transition function dictates the system's new state when no events have occurred since the last transition. While the external transition function dictates the system's new state when an external event occurs – this state is determined by the input, *x*, the current state, *S*, and how long the system has been in this state, *e*, when the external event occurred. In both cases, the system is then is some new state *S'* with some new resting time, *ta*(*s'*) and it continues the same way.

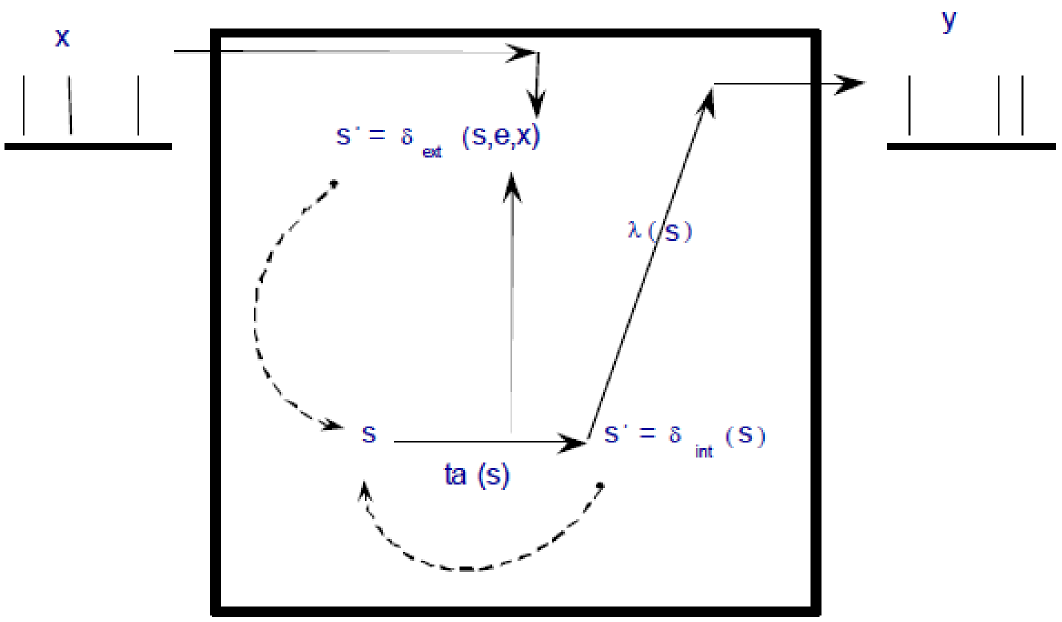


Figure 2: DEVS in Action

The algorithm of the simulator that would execute, and generate the behaviour of the semantics of the DEVS 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, sinit. The event times, tL and tN are set to 0 and ta (sinit), 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.

Classic DEVS (CDEVS) Coupled Model

Basically, a DEVS coupled model is composed of DEVS components i.e. atomic and coupled models, by defining a coupling relation between them. Here is the definition of DEVS coupled models: $CM = (X, Y, D,\{M_d|d\in D\},EIC,EOC,IC)$

Where :

- X is the set of input values,
- Y is the set of output values,
- D is the set of model references, that is to say a set of names associated to the model's components $\{Md / d \in D\}$ is the set of coupled model's components, with d being in D. These components are either atomic or coupled DEVS model, IC, EIC and EOC define the coupling structure in the coupled system.
- IC defines the internal coupling, transforming a component's output into another component's input within the coupled model.
- EIC is the set of external input coupling, which connects the inputs of a coupled model to components inputs.
- EOC is the set of external output coupling.

However, no direct feedback loops are allowed, i.e., no output port of a component may be connected to an input port of the same component.

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 apply Delt message to the
6. simulators – for those simulators not receiving any input, the messages sent are empty
7. 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.

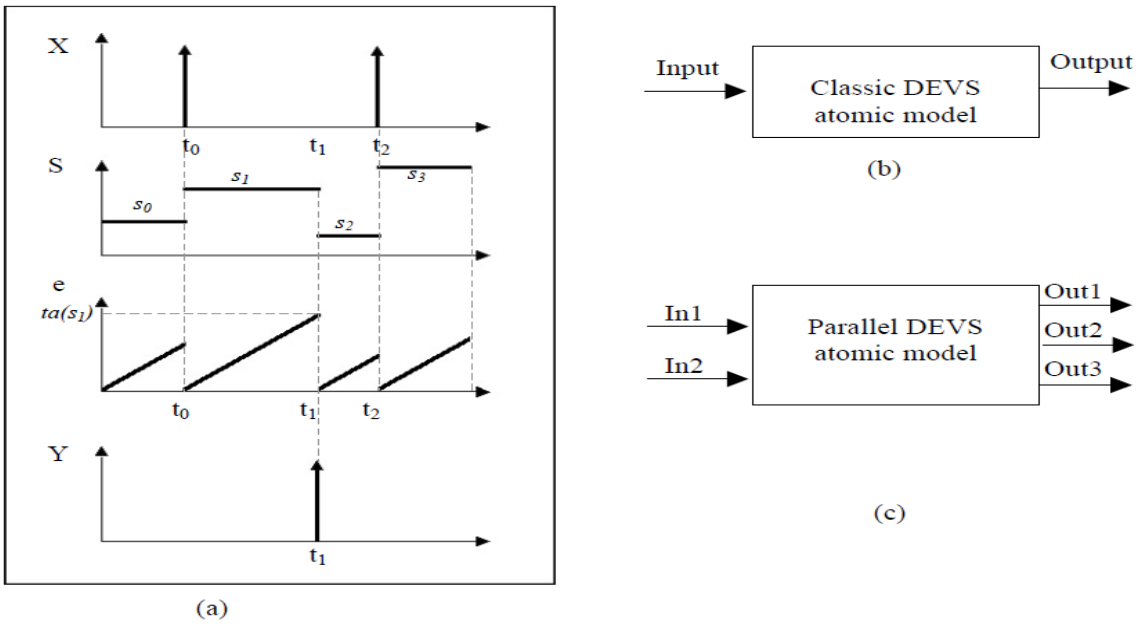


Figure 3: (a) DEVS behaviour

(b) Graphical notation for Classic DEVS and Parallel DEVS

2.2 Examples of CDEVS model and simulation

Generator: An atomic model example with single input and single output. The model generates output events, and the frequency of the output events is proportional to the input value.

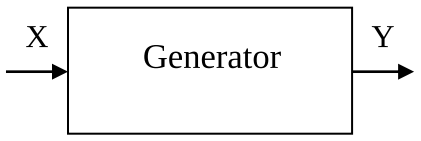


Figure 4: Example of CDEVS Atomic model

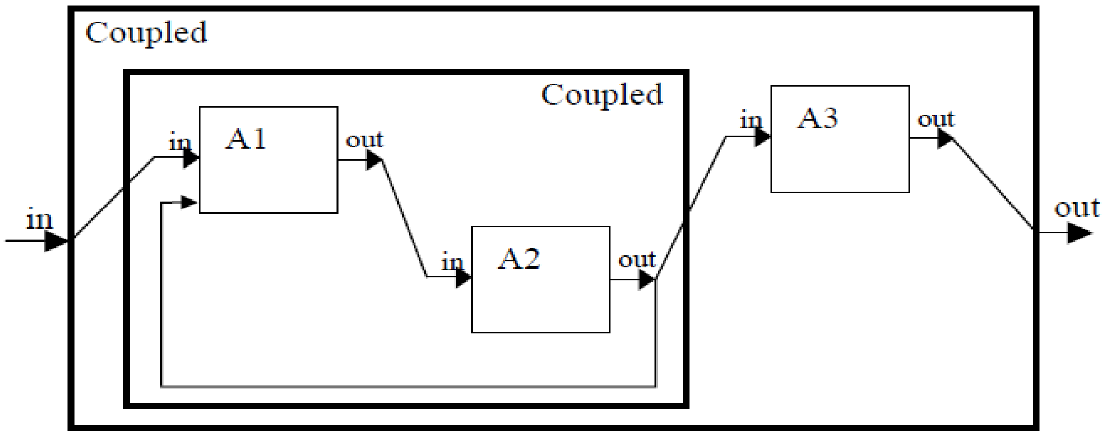


Figure 5: Simple CDEVS Coupled model with three Atomic models

2.3 Parallel DEVS (PDEVS)

About 15 years after the Classic DEVS was introduced, its 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.

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 real's with 0 and ∞. (time advance function)
- δext : Q x Xb M → 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
- δint : S → S is the internal transition function
- $\delta conf$: Q x Xb, M → S is the confluent transition function, where
- λ: S → Y is the output function

1. Instead of having a single input, here we have 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 is given in the figure 6 below.

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 CM = (X, Y, D, { Md | d Є 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 Є D, Md 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

2.4 The PDEVS Simulation Algorithm

The simulation process for DEVS models, whether *Atomic* or *coupled,* proceeds by iteration of a basic cycle as is illustrated in the Figure bellow. Processing can be carried out in two ways: event-driven or time-stepped. The event driven approach, which relates to the spirit of the conservative and optimistic schemes is usually much faster and more efficient. However, the time-stepped approach allows easier animation and can be employed for execution of models in real wall clock time, as opposed to simulated time.

Atomic Model Simulators

Let's take an atomic model as it undergoes simulation. In both the event-driven or time-stepped approaches, 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 for its model. Initially, the state of the model is initialized as specified by the modeler to a desired initial state, sinit. The event times, tL and tN are set to 0 and ta(s init), respectively.

In event-driven execution, if there are no external events, the clock, t is advanced to tN whereupon the output is generated and the internal transition function of the model is executed. The simulator then updates the event times as shown, 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 and the input is processed by the confluent or external event transition function, depending on whether text coincides with tN or not.

The time-stepped approach is employed in the atomic and coupled applets of DEVSJAVA. Here each *Atomic* model is assigned its own individual thread and can be executed in stand-alone fashion, as well as a component within a coupled model. To advance time, the loop in Figure below contains a sub-loop, in which the thread sleeps for 1 millisecond intervals until tN is reached whence it exits the sub-loop and executes the output and internal transition functions. While in the sub-loop the thread checks for notice of an external event, which may originate from the mouse or from the output of another model. In this case, it makes an early exist from the sub-loop and executes the confluent or external transition function as appropriate. While in the subloop, the thread continually repaints a panel associated with the applet with a sequence of images (sounds can be added) determined by the current *phase* of the model.

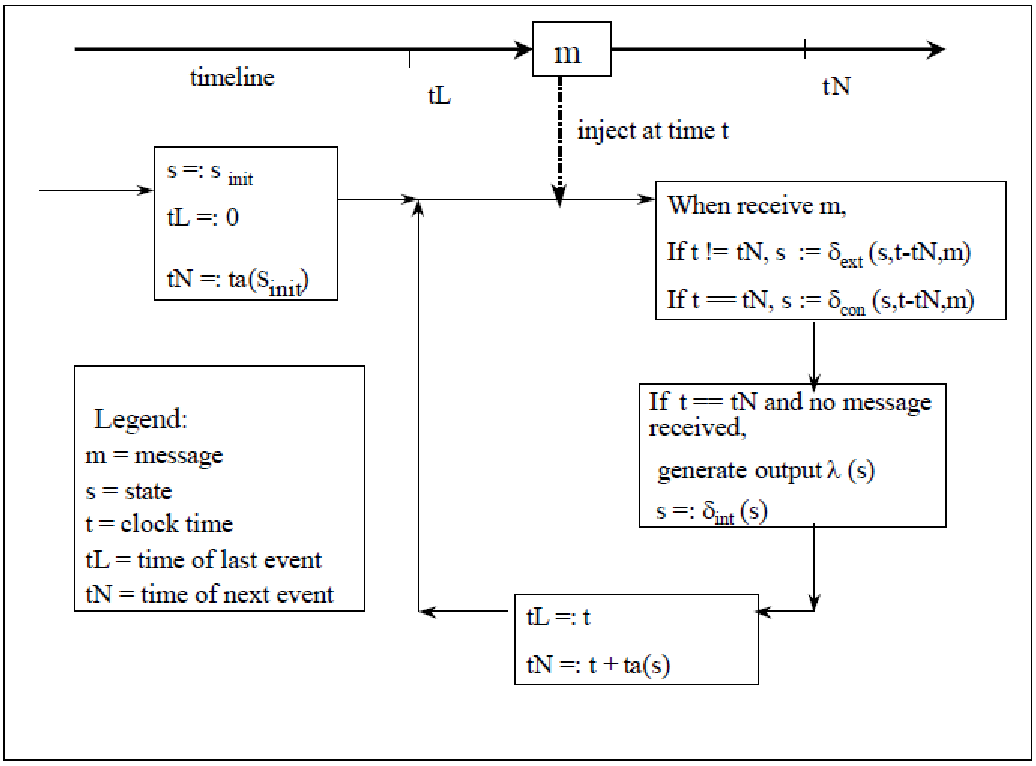


Figure 6: DEVS Simulation Process

Coupled Model Coordinators

As shown in Figure bellow, the Parallel DEVS scheme differs from the conservative and optimistic schemes in that there is a *coordinator* to synchronize the simulation cycle through its steps. To start a cycle, the coordinator, C collects the times of next event from the component simulators. It sends the minimum of these times back to the components, thereby allowing them to determine whether they are imminent, and if so to generate output. More than one component may be imminent and the outputs of all such imminent are sorted and distributed to others according to the coupling specification of the coupled model. The transition functions of the imminent components, as well as all other recipients of inputs, are then applied. As we have seen in the atomic simulator case, which transition is applied, depends on the state and input of a component – imminent with no inputs apply internal transition functions, imminent with inputs apply confluent transition functions, and non-imminent components with input apply external transition functions. The resulting changes in states may cause new values for time advances and this are sent to the coordinator. Processing then continues to the next cycle.

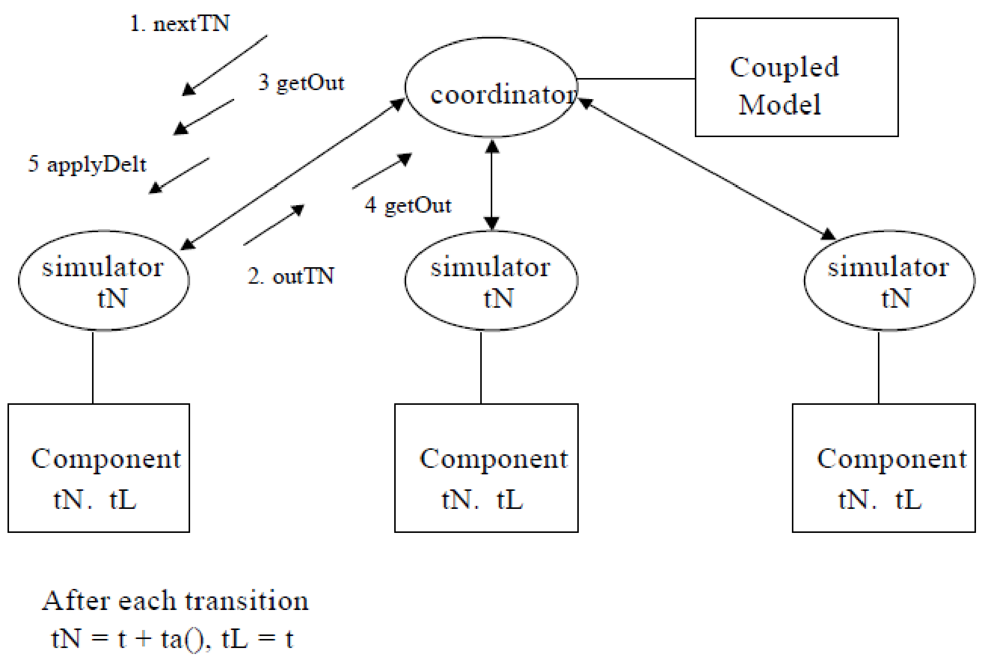


Figure 7: DEVS Simulation Protocol

Let's follow the sequence of steps in one simulation cycle:

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, Coordinator sends to each simulator a getOut message containing the global tN (the minimum of the tNs)
3. 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 getOut message.
4. 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.

As already mentioned, 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 it is imminent and its input message is not empty, it invokes its model's confluence 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.

2.5 Example of PDEVS model and simulation (by hand)

CROSS ROAD MODEL

The cross road have five atomic models that are later coupled together to form a single cross road . The cross road network can be of any form, ranging from two crossroads to multiple roads with a single intersection.

The atomic models are:

1. Car Generator
2. Road
3. Traffic light
4. Platform
5. Merge

CAR GENERATOR: This atomic model generates cars that are used in the simulation, each car with its own attribute. The car generator have two states, Generate Send and Generate don’t send. This depends on the present state of the road in which the generated cars will enter. The car generator has no input and it has one output port that output cars that are generated.

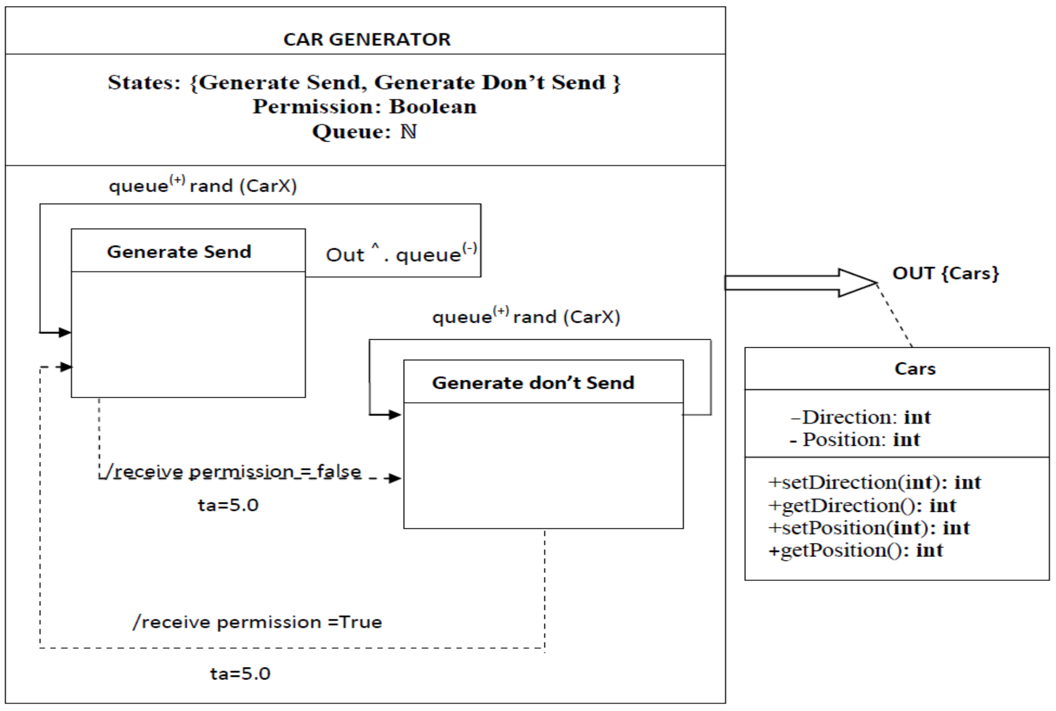


Figure 8: Car Generator Model

ROAD: This atomic model is to show the movement of cars when simulated. The road can be leading the cars to an intersection (platform) or going out of it the platform to another road. A car can be in a cell at any time. The road have three states, Free Road, Busy Road Moving, (Which means there is an empty cell in front of a the cell that the car is currently). and lastly Busy Road Not Moving (that is the cells in front of the cell that the car is currently are not empty).

The road has 2 input port structures and two output structures. The input structures are IN Element of cars received from the car generator or platform and IN element of OUT Platform which comes from the controller indicating whether a car at the end of the road can enter the intersection (platform). The output port structures are OUT element of cars which sends cars out to the intersection (platform) and the OUT element of IN Platform which goes to a place indicating whether the road can take more cars or not.

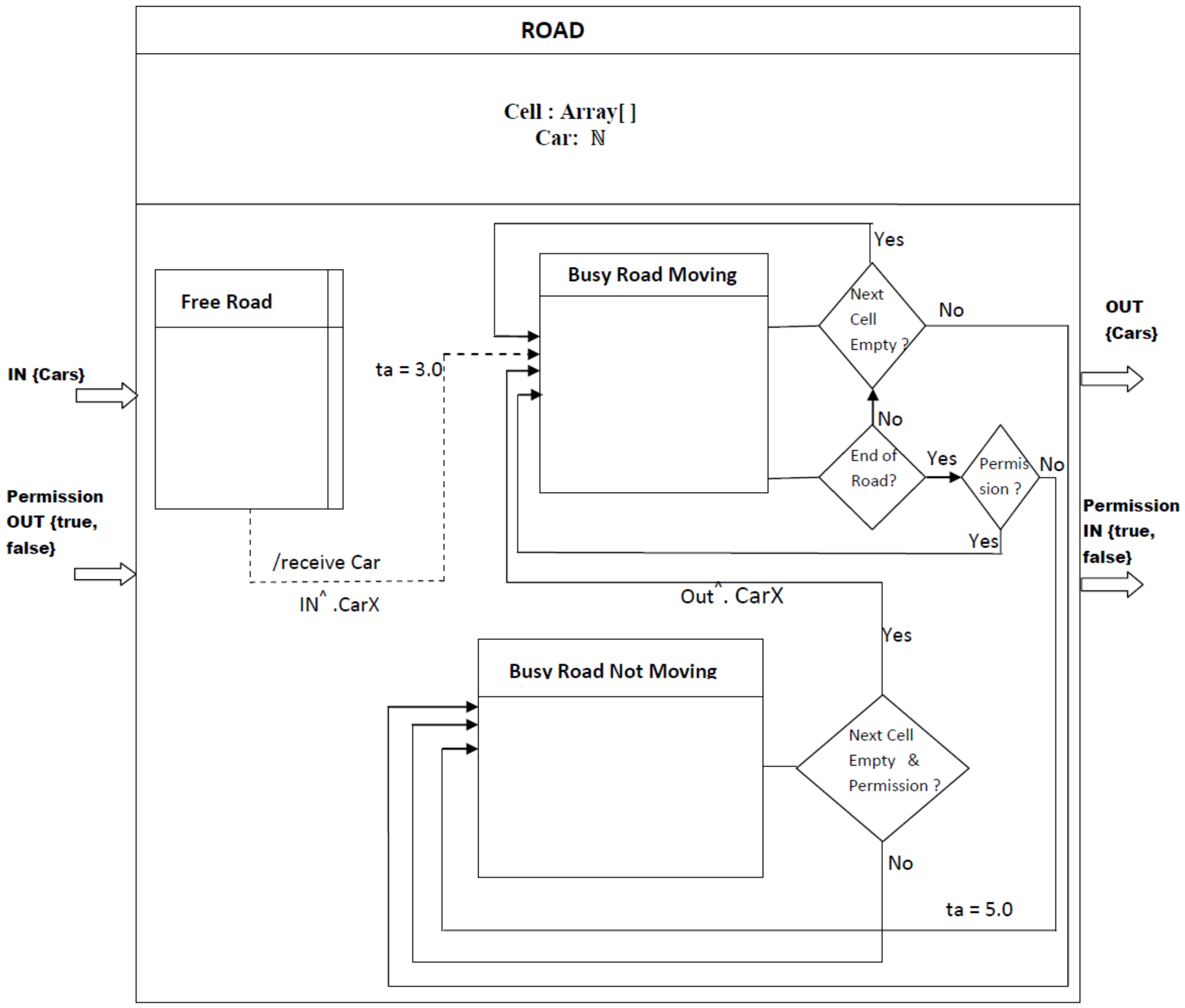


Figure 9: Road Model

TRAFFIC LIGHT: The traffic light atomic model displays light (Green, Red, Yellow or Black). The traffic light have seven states: Green, Yellow Before Red, Red, Yellow After Red, After Blink, Before Blink and Blink. The process has one input port structure which is used to switched the traffic light On and Off. The single output port structure is used to display light and it is connected to a merge.

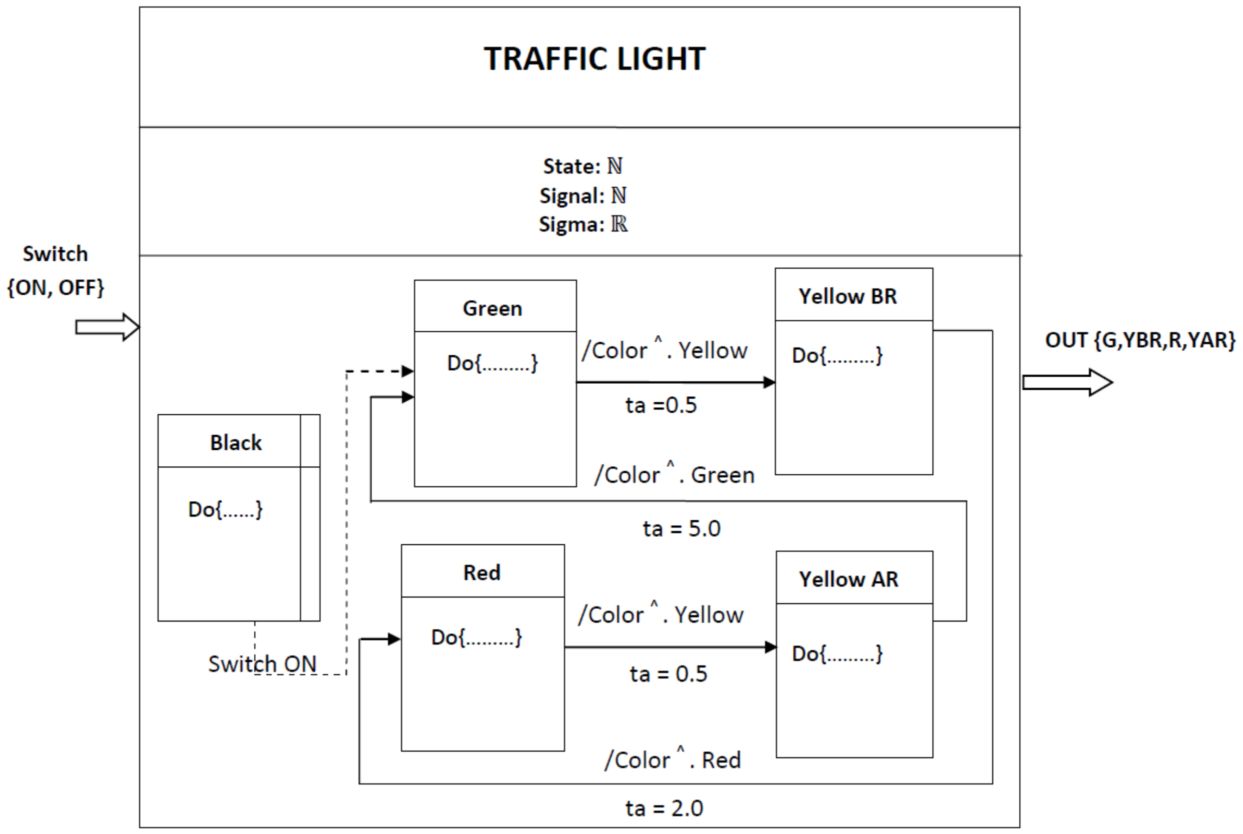


Figure 10: Traffic Light Model

PLATFORM: This atomic model is the intersection which can take only one car at a time. The place can be in four states. They are Platform Empty No Permission (When there is no car is in the intersection and there is no permission to move), Platform Empty Permission (When there is no car in the intersection and there is permission to move), Platform Not Empty No Permission (When a car is in the intersection and no permission to move) and Platform Not Empty Permission (When a car is in the intersection and There is permission to move).

The platform has two input port structures, IN element of cars from road and IN element of Out platform. There are two output port structure, Out element of Car and Out element of permission.

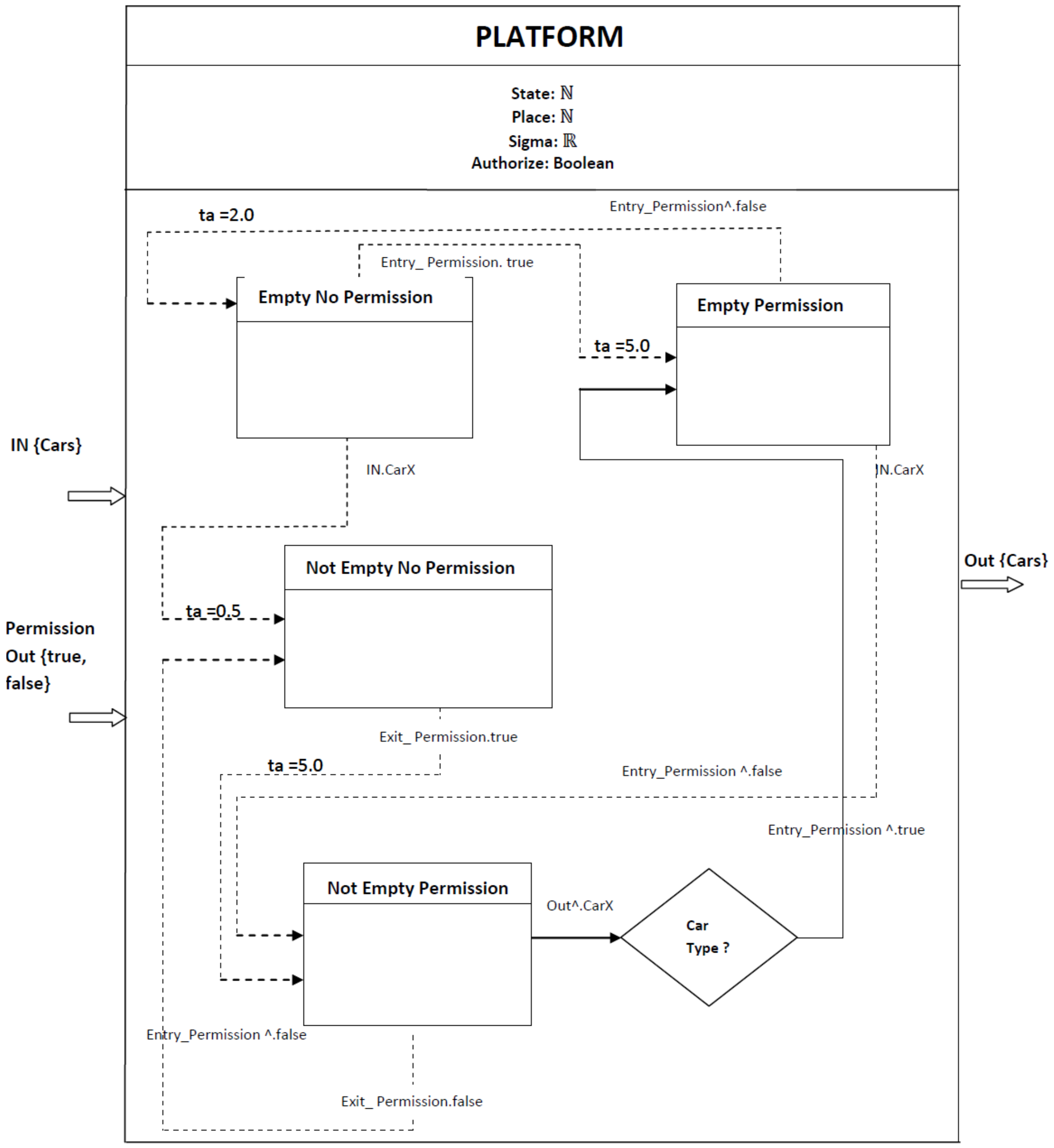


Figure 11: Platform Model

MERGE: The merge atomic model sends permission to the road indicating whether a car is allowed to enter the intersection or not. This permission is based on whether the platform is free and that the light is green.

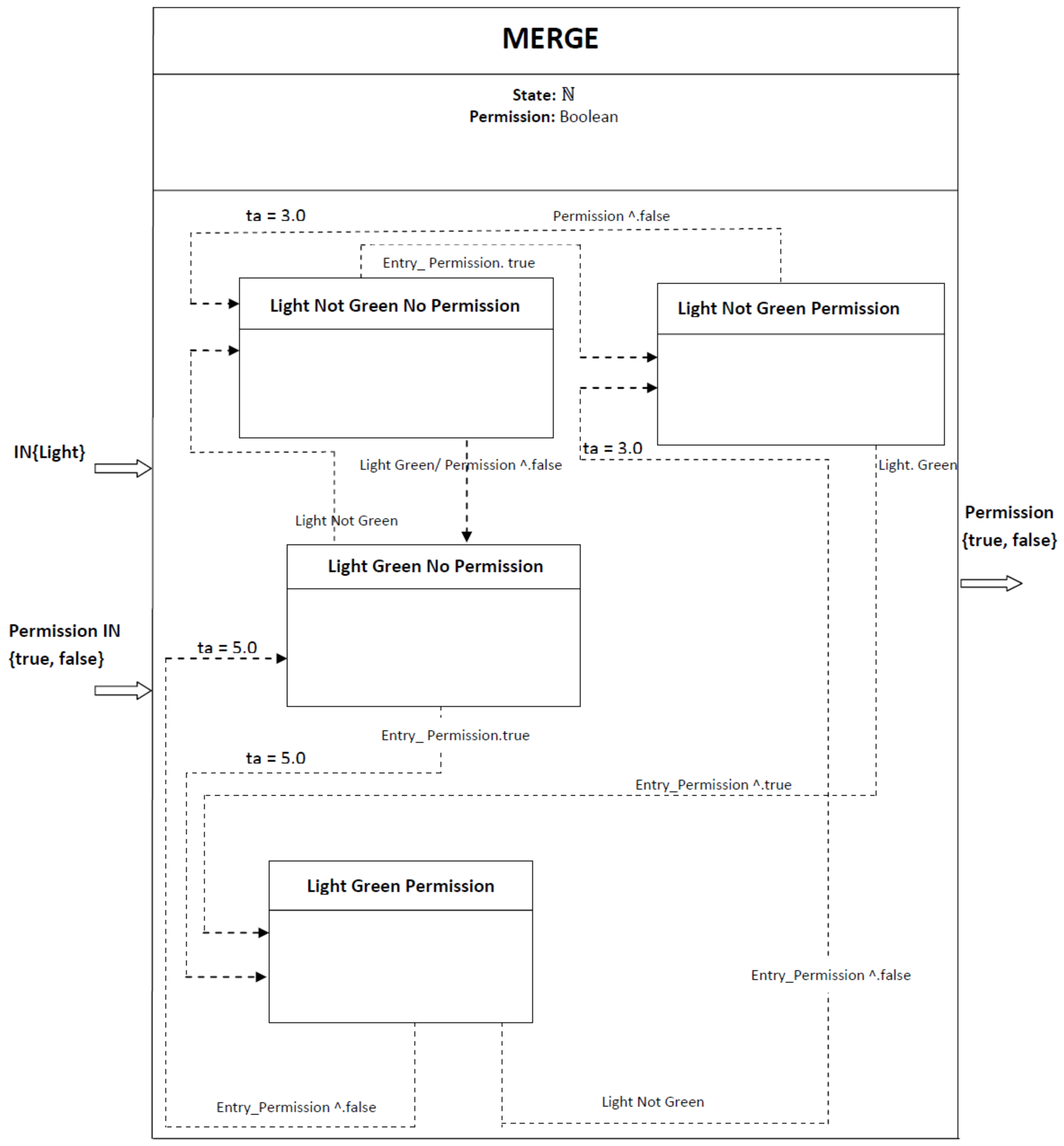


Figure 12: Merge Model

COUPLED MODEL

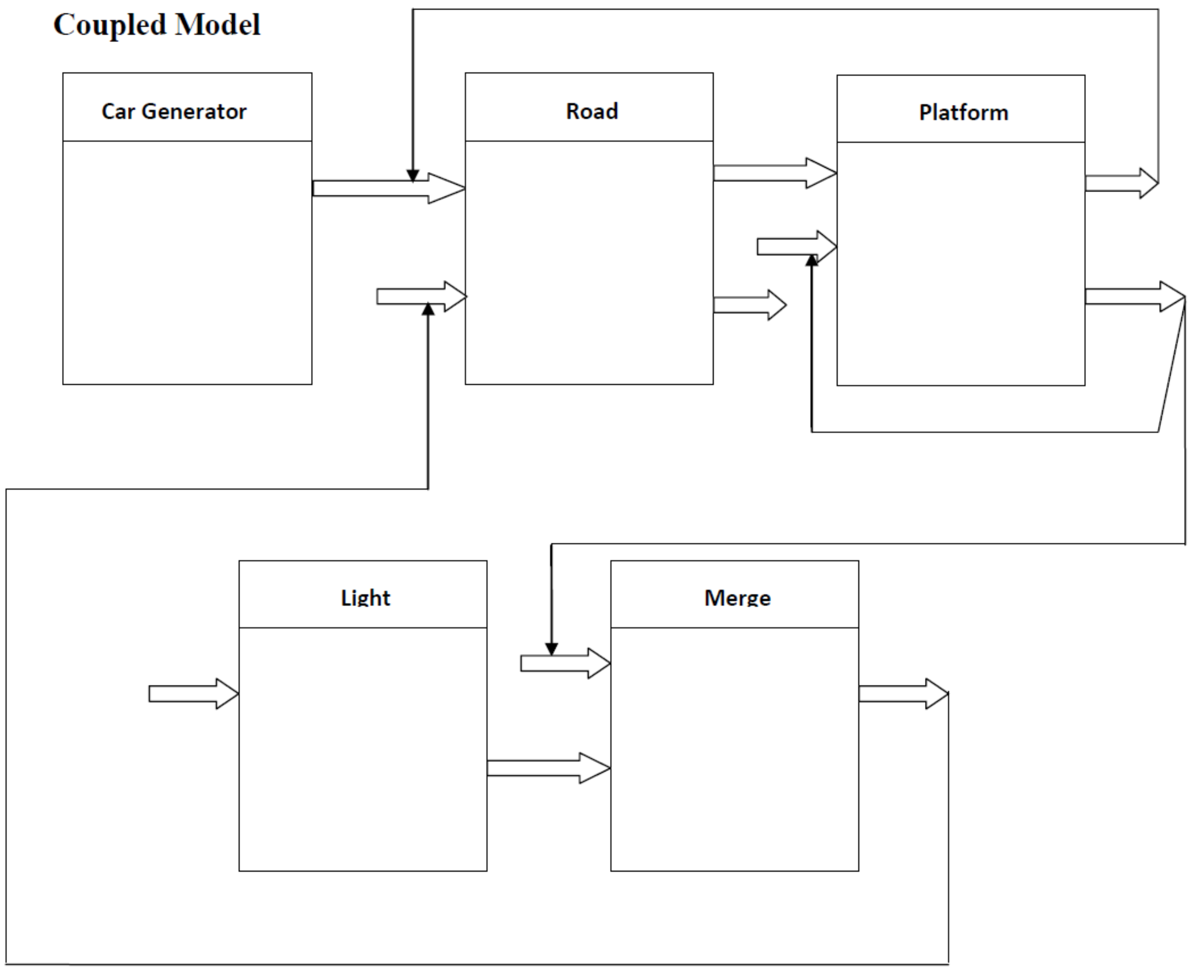
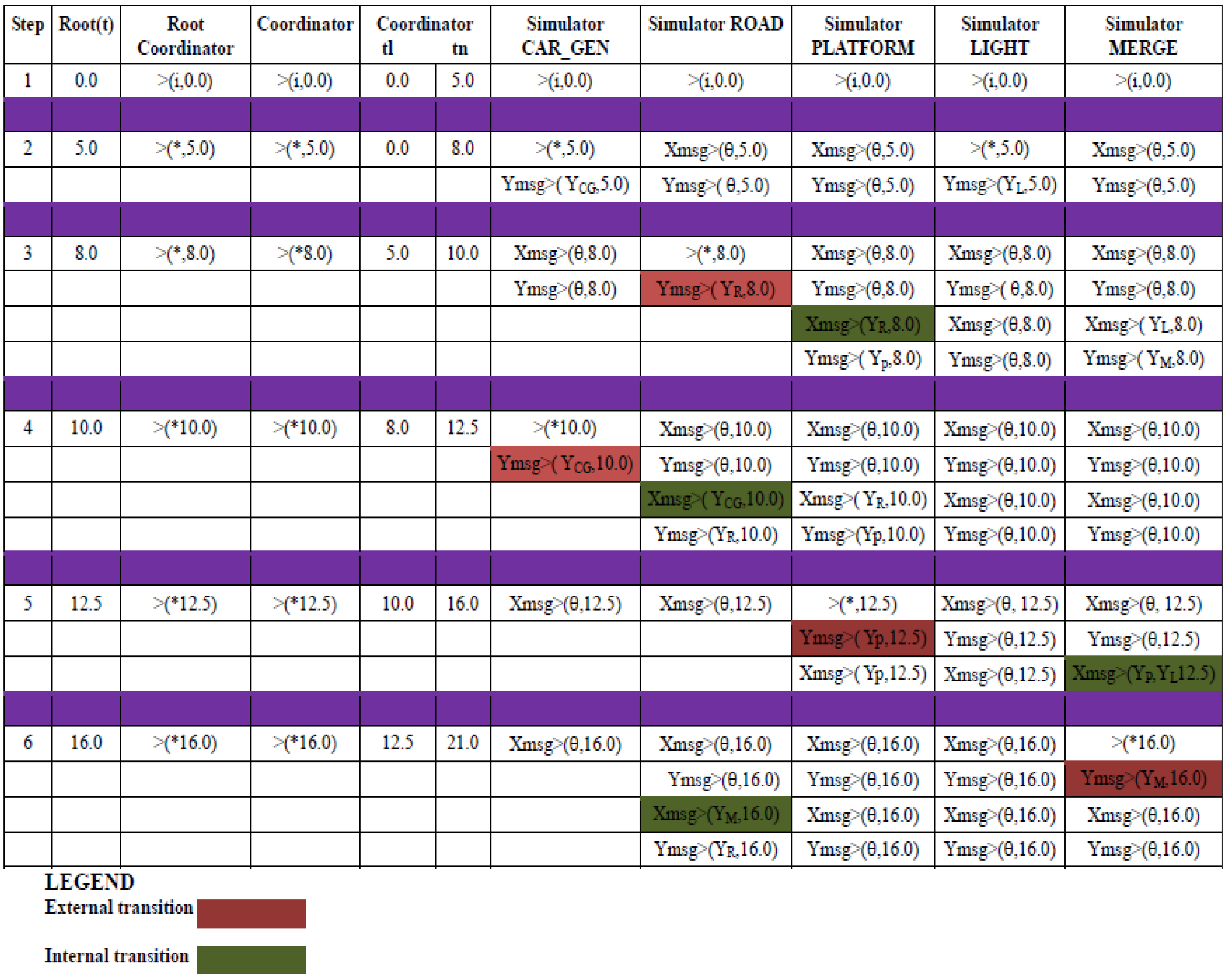


Figure 13: Cross Road Coupled Model

Simulation by hand



Step	Root(t)	Root Coordinator	Coordinator	Coordinator		Simulator CAR_GEN	Simulator ROAD	Simulator PLATFORM	Simulator LIGHT	Simulator MERGE
				tl	tn					
1	0.0	>(i,0.0)	>(i,0.0)	0.0	5.0	>(i,0.0)	>(i,0.0)	>(i,0.0)	>(i,0.0)	>(i,0.0)
2	5.0	>(*,5.0)	>(*,5.0)	0.0	8.0	>(*,5.0)	Xmsg>(θ,5.0)	Xmsg>(θ,5.0)	>(*,5.0)	Xmsg>(θ,5.0)
						Ymsg>(Y_{CG},5.0)	Ymsg>(θ,5.0)	Ymsg>(θ,5.0)	Ymsg>(Y_L,5.0)	Ymsg>(θ,5.0)
3	8.0	>(*,8.0)	>(*8.0)	5.0	10.0	Xmsg>(θ,8.0)	>(*,8.0)	Xmsg>(θ,8.0)	Xmsg>(θ,8.0)	Xmsg>(θ,8.0)
						Ymsg>(θ,8.0)	🟥 Ymsg>(Y_R,8.0)	Ymsg>(θ,8.0)	Ymsg>(θ,8.0)	Ymsg>(θ,8.0)
								🟩 Xmsg>(Y_R,8.0)	Xmsg>(θ,8.0)	Xmsg>(Y_L,8.0)
								Ymsg>(Y_p,8.0)	Ymsg>(θ,8.0)	Ymsg>(Y_M,8.0)
4	10.0	>(*10.0)	>(*10.0)	8.0	12.5	>(*10.0)	Xmsg>(θ,10.0)	Xmsg>(θ,10.0)	Xmsg>(θ,10.0)	Xmsg>(θ,10.0)
						🟥 Ymsg>(Y_{CG},10.0)	Ymsg>(θ,10.0)	Ymsg>(θ,10.0)	Ymsg>(θ,10.0)	Ymsg>(θ,10.0)
							🟩 Xmsg>(Y_{CG},10.0)	Xmsg>(Y_R,10.0)	Xmsg>(θ,10.0)	Xmsg>(θ,10.0)
							Ymsg>(Y_R,10.0)	Ymsg>(Yp,10.0)	Ymsg>(θ,10.0)	Ymsg>(θ,10.0)
5	12.5	>(*12.5)	>(*12.5)	10.0	16.0	Xmsg>(θ,12.5)	Xmsg>(θ,12.5)	>(*,12.5)	Xmsg>(θ, 12.5)	Xmsg>(θ, 12.5)
								🟥 Ymsg>(Yp,12.5)	Ymsg>(θ,12.5)	Ymsg>(θ,12.5)
								Xmsg>(Yp,12.5)	Xmsg>(θ,12.5)	🟩 Xmsg>(Yp,Y_L12.5)
6	16.0	>(*16.0)	>(*16.0)	12.5	21.0	Xmsg>(θ,16.0)	Xmsg>(θ,16.0)	Xmsg>(θ,16.0)	Xmsg>(θ,16.0)	>(*16.0)
							Ymsg>(θ,16.0)	Ymsg>(θ,16.0)	Ymsg>(θ,16.0)	🟥 Ymsg>(Y_M,16.0)
							🟩 Xmsg>(Y_M,16.0)	Xmsg>(θ,16.0)	Xmsg>(θ,16.0)	Xmsg>(θ,16.0)
							Ymsg>(Y_R,16.0)	Ymsg>(θ,16.0)	Ymsg>(θ,16.0)	Ymsg>(θ,16.0)

LEGEND
External transition 🟥

Internal transition 🟩

Figure 14: Hand Simulation

As explained in the PDEVS simulation algorithm, the Simulation by hand step is as follows:

1. The root coordinator at time t = 0.0 send I-message (initialization message) to the top most coordinators and tL (time of last event) and tN (time of next event) is updated.
2. An internal state transition message (S-message/star message), at time t = tN of the topmost coordinator is transmitted in a loop to the topmost coordinator.
3. An initialization message (I-message) is transmitted to all coordinators and simulators of the components of the coupled model, after which the time of the last event the tL and tN is calculated.
4. If an internal state transition message (S-message/star message) is received it will be forwarded to all components of a coupled model which are imminent.
5. The input message (X-message) is forwarded according to the coupling in the above

figure, while the other simulator that are not imminent receive an empty message (θ-message) and the event times tL and tN is updated.

6. All output messages (Y-message) from lower simulators and coordinators are saved and after all members in the set imminent of the coordinator have answered with an output message (Y-message) all events saved in mail are transmitted to their receivers and to the superior coordinator.

Note: At each step an empty input message (θ-message) is sent to all members of the set imminent who have no input event in the saved mail, the event times tL and tN are updated.

For Simulators (Atomic models):

- An initialization message (I-message) leads to the calculation of the time of the next internal event tN and the time of the last event tL of the atomic model.
- An internal state transition message (S_message) indicates an internal event. Therefore the output function $y = \lambda(s)$ of the atomic model is then calculated and the output events are transmitted with an output message (Y-message) ($\lambda(s)$, t) to the bigger coordinator.
- An input message (X-message) indicates an internal and/or external event and the appropriate transition function is carried out.
- The internal transition function $\delta int(s)$, if the simulation time $t = tN$ and the input event x contains an empty set.
- The confluent function $\delta conf(s, e, x)$, if the simulation time $t = tN$ and the input event x contains a non-empty bag of events.
- The external transition function $\delta ext(s, e, x)$, if $tL \leq t < tN$ and the input event x contains a non empty bag of events.
- After carrying out the appropriate transition function the last event time tL is set to the current simulation time t and the time of the next internal event tN is calculated.
- The loop continues until the simulation ends.

Chapter 3: Literature Review on PDEVS Implementations

3.1 Survey of PDEVS Implementations

PDEVS have been implemented by several persons, but we will do our survey based on the implementation presented by (Aminu, 2009) and the implementations presented by (Doyin, 2010).

The implementations are as follows:

1. An implementation, that realized from 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).
2. An implementation of the PDEVS simulation algorithm in a sequential manner.
3. An implementation of the PDEVS simulation algorithm that takes advantage of parallelism provided 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.

IMPLEMENTATIONS OF THE PDEVS SIMULATION SYSTEM

Implementation I

This implementation was prepared as a Java package (SIMSTUDIO_1_1) by (Aminu, 2009) in his thesis work. This consists of 6 packages namely Simulator, Model, Message, Exception, types 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.

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.
- 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.

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

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

Exception package: handle the following exceptions classes

- 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.

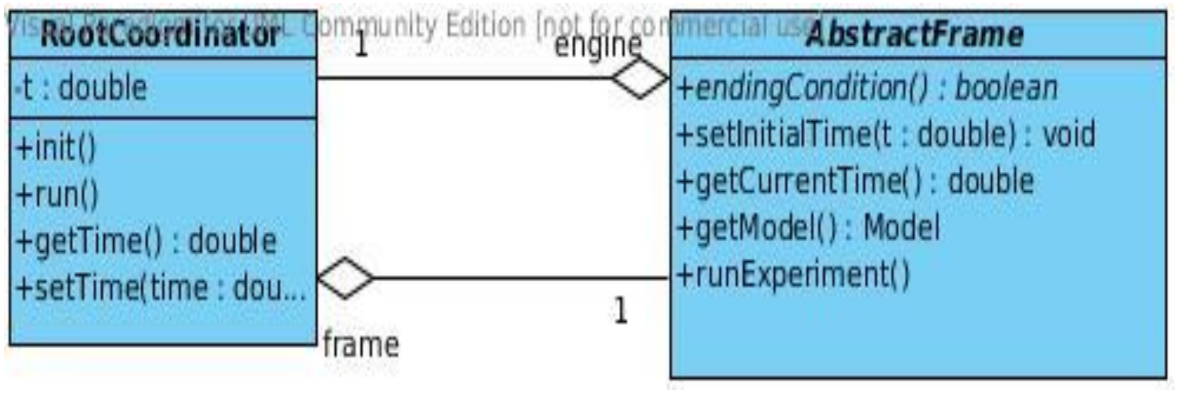
Types Package: Contains all the Devs Type classes.

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

Implementation II

This implementation was prepared as a Java package by Doyin[2010] in her thesis work. There are 3 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'srun() 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.



RootCoordinator
-t : double
+init()
+run()
+getTime() : double
+setTime(time : dou...
1
engine
frame
1
AbstractFrame
+endingCondition() : boolean
+setInitialTime(t : double) : void
+getCurrentTime() : double
+getModel() : Model
+runExperiment()
Visual Paradigm for UML Community Edition [not for commercial use]

Figure 15: Frame Package

Model Package: contains description required by a 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.
- 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.

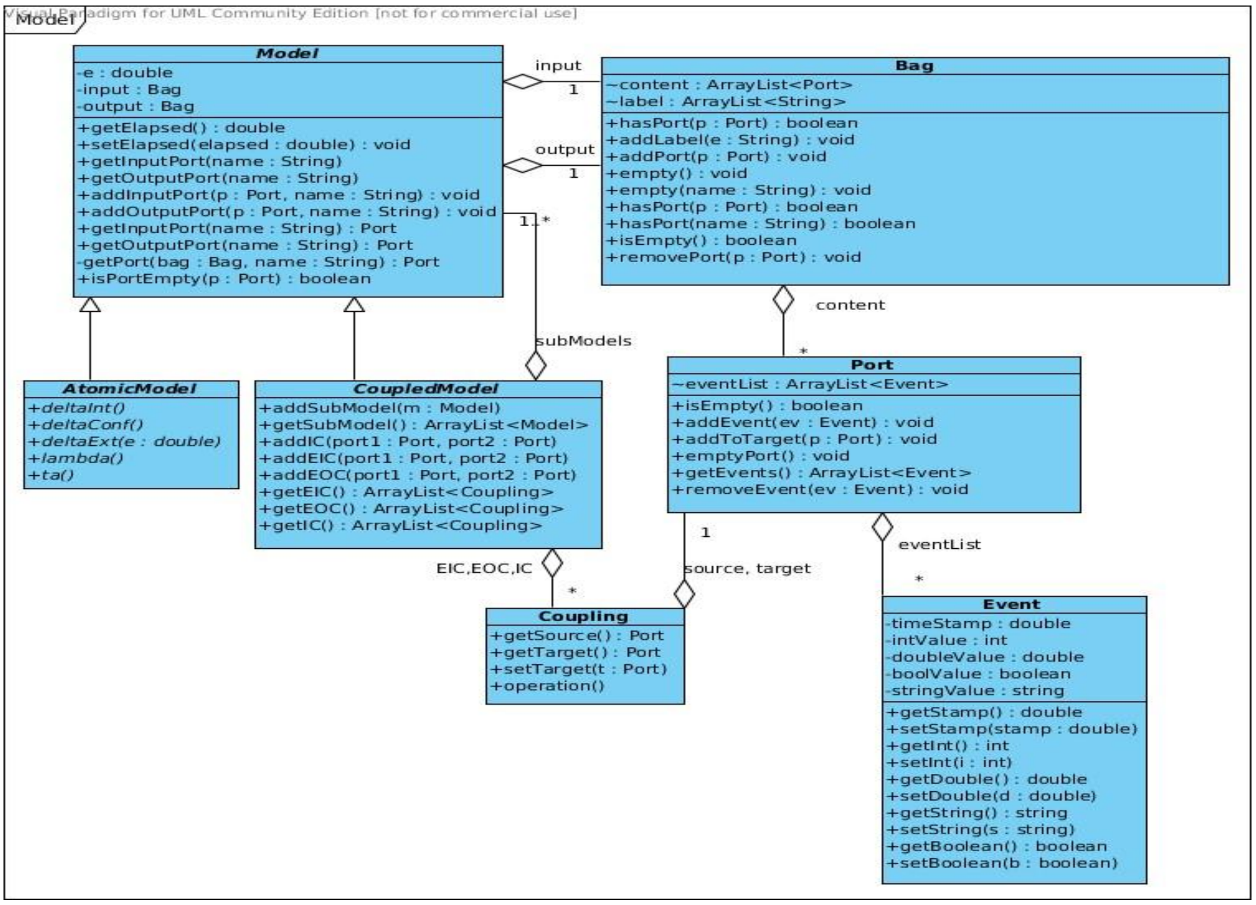

Visual Paradigm for UML Community Edition [not for commercial use]
Model
Model
-e : double
-input : Bag
-output : Bag
+getElapsed() : double
+setElapsed(elapsed : double) : void
+getInputPort(name : String)
+getOutputPort(name : String)
+addInputPort(p : Port, name : String) : void
+addOutputPort(p : Port, name : String) : void
+getInputPort(name : String) : Port
+getOutputPort(name : String) : Port
-getPort(bag : Bag, name : String) : Port
+isPortEmpty(p : Port) : boolean
input
1
output
1
1..*
subModels
Bag
~content : ArrayList<Port>
~label : ArrayList<String>
+hasPort(p : Port) : boolean
+addLabel(e : String) : void
+addPort(p : Port) : void
+empty() : void
+empty(name : String) : void
+hasPort(p : Port) : boolean
+hasPort(name : String) : boolean
+isEmpty() : boolean
+removePort(p : Port) : void
content
*
AtomicModel
+deltaInt()
+deltaConf()
+deltaExt(e : double)
+lambda()
+ta()
CoupledModel
+addSubModel(m : Model)
+getSubModel() : ArrayList<Model>
+addIC(port1 : Port, port2 : Port)
+addEIC(port1 : Port, port2 : Port)
+addEOC(port1 : Port, port2 : Port)
+getEIC() : ArrayList<Coupling>
+getEOC() : ArrayList<Coupling>
+getIC() : ArrayList<Coupling>
Port
~eventList : ArrayList<Event>
+isEmpty() : boolean
+addEvent(ev : Event) : void
+addToTarget(p : Port) : void
+emptyPort() : void
+getEvents() : ArrayList<Event>
+removeEvent(ev : Event) : void
1
source, target
eventList
*
EIC,EOC,IC
*
Coupling
+getSource() : Port
+getTarget() : Port
+setTarget(t : Port)
+operation()
Event
-timeStamp : double
-intValue : int
-doubleValue : double
-boolValue : boolean
-stringValue : string
+getStamp() : double
+setStamp(stamp : double)
+getInt() : int
+setInt(i : int)
+getDouble() : double
+setDouble(d : double)
+getString() : string
+setString(s : string)
+getBoolean() : boolean
+setBoolean(b : boolean)

Figure 16: Model Package

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.

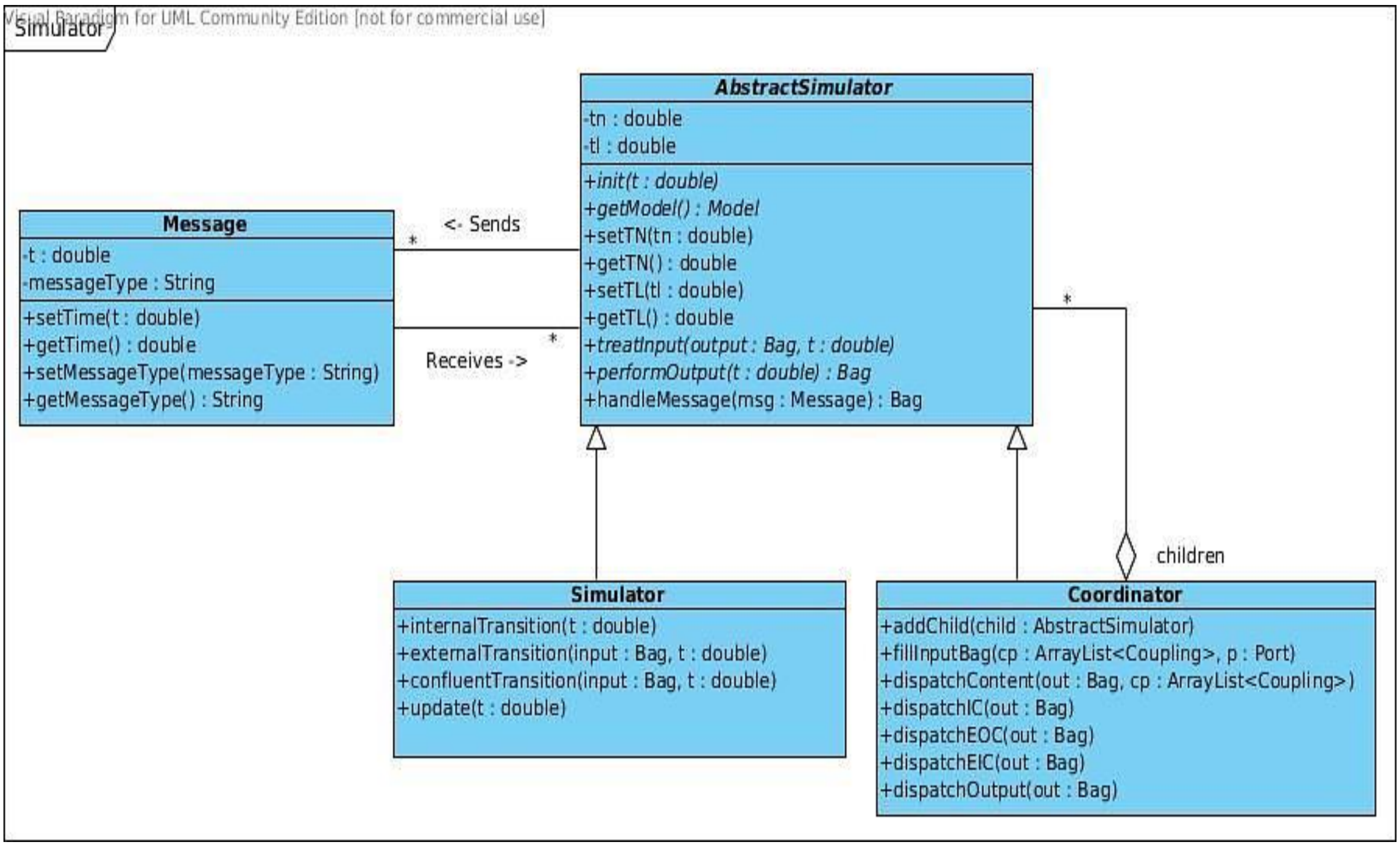

Visual Paradigm for UML Community Edition [not for commercial use]
Simulator
AbstractSimulator
-tn : double
-tl : double
+init(t : double)
+getModel() : Model
+setTN(tn : double)
+getTN() : double
+setTL(tl : double)
+getTL() : double
+treatInput(output : Bag, t : double)
+performOutput(t : double) : Bag
+handleMessage(msg : Message) : Bag
Message
-t : double
-messageType : String
+setTime(t : double)
+getTime() : double
+setMessageType(messageType : String)
+getMessageType() : String
<- Sends
Receives ->
*
*
*
children
Simulator
+internalTransition(t : double)
+externalTransition(input : Bag, t : double)
+confluentTransition(input : Bag, t : double)
+update(t : double)
Coordinator
+addChild(child : AbstractSimulator)
+fillInputBag(cp : ArrayList<Coupling>, p : Port)
+dispatchContent(out : Bag, cp : ArrayList<Coupling>)
+dispatchIC(out : Bag)
+dispatchEOC(out : Bag)
+dispatchEIC(out : Bag)
+dispatchOutput(out : Bag)

Figure 17: Simulator Package

Implementation III

The implementation III was also implemented as a Java package by Doyin[2010] in her thesis work. It contains three packages namely Frame, Model both of which have been described in the previous section and the Simulator described below.

This differs from the first two implementations because it makes use of multi-threading. 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.

3.2 SimStudio implementation (meta-models)

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.

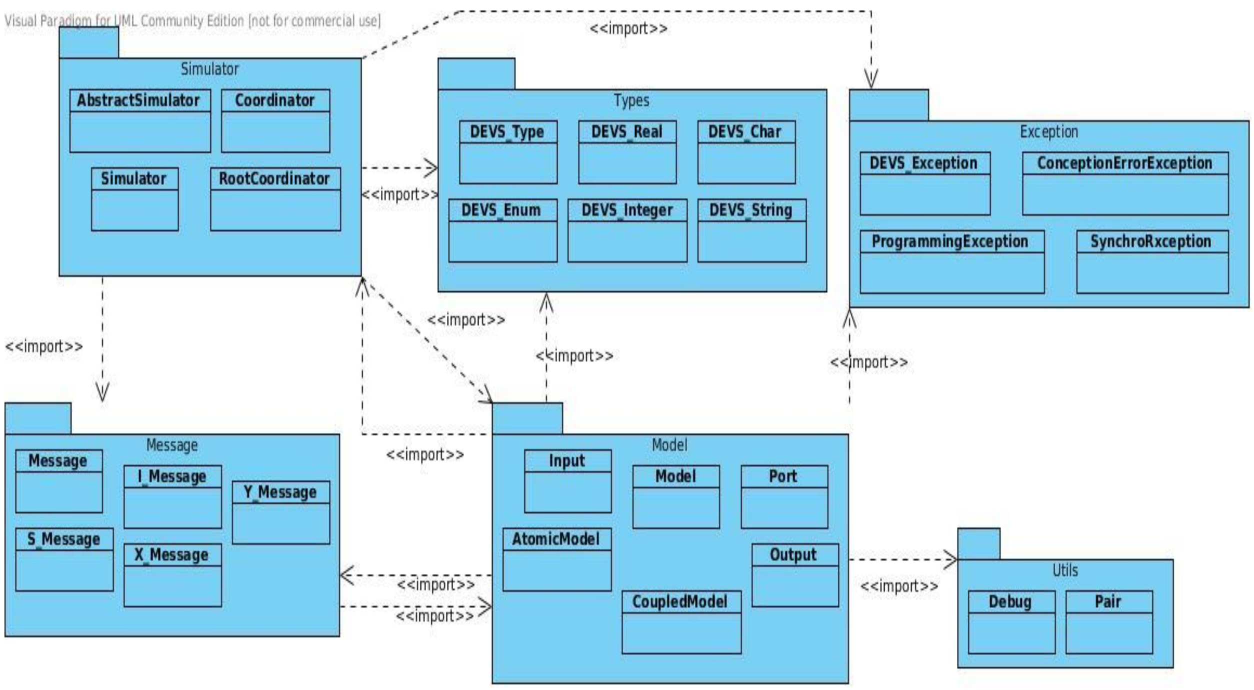


Figure 18: Package and Class View of Implementation I

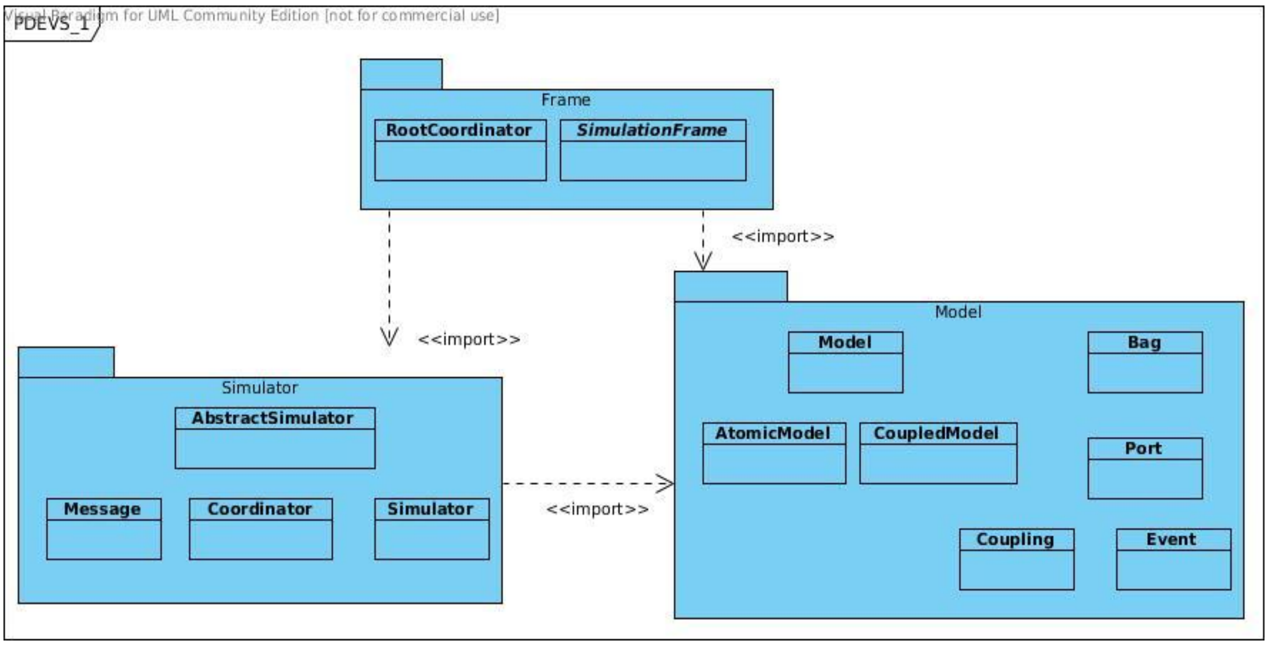


Figure 19: Package and Class View of Implementation II

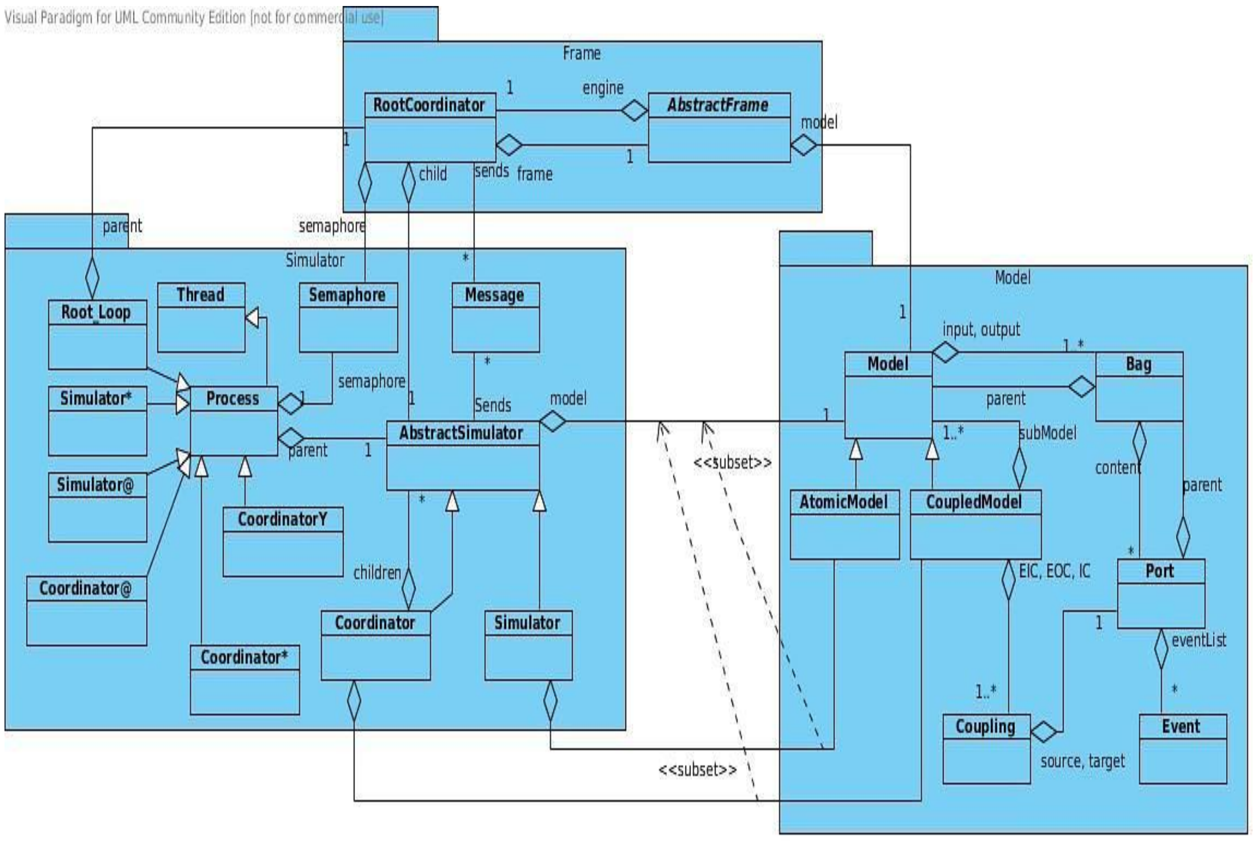


Figure 20: Class diagram of implementation II

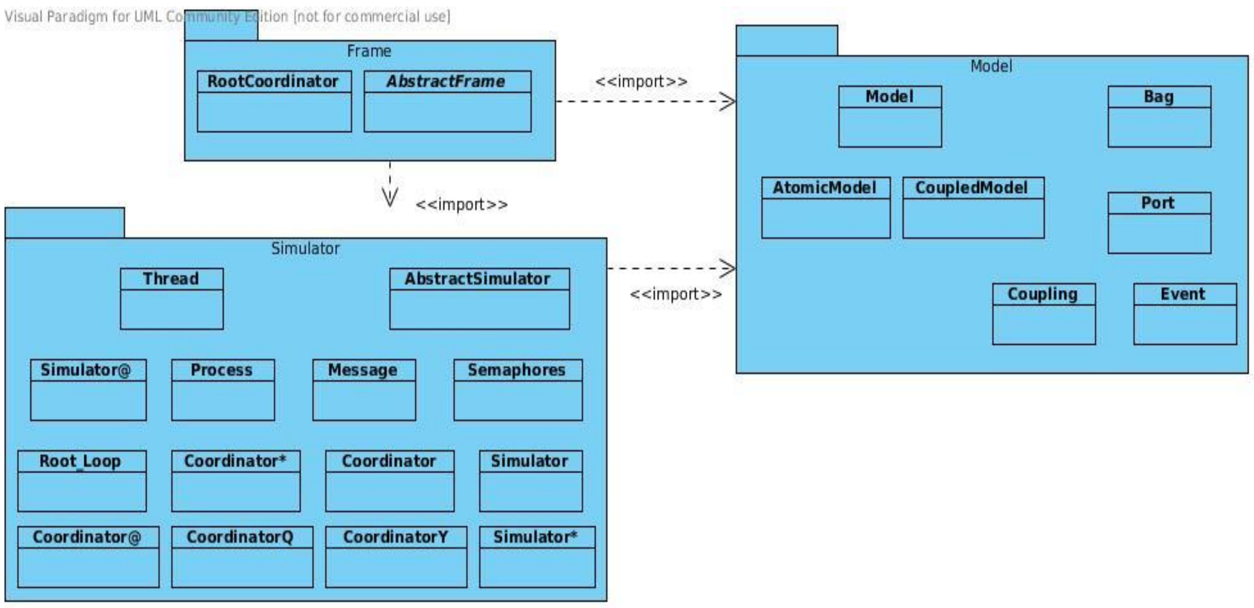


Figure 21: Package and Class View of Implementation III

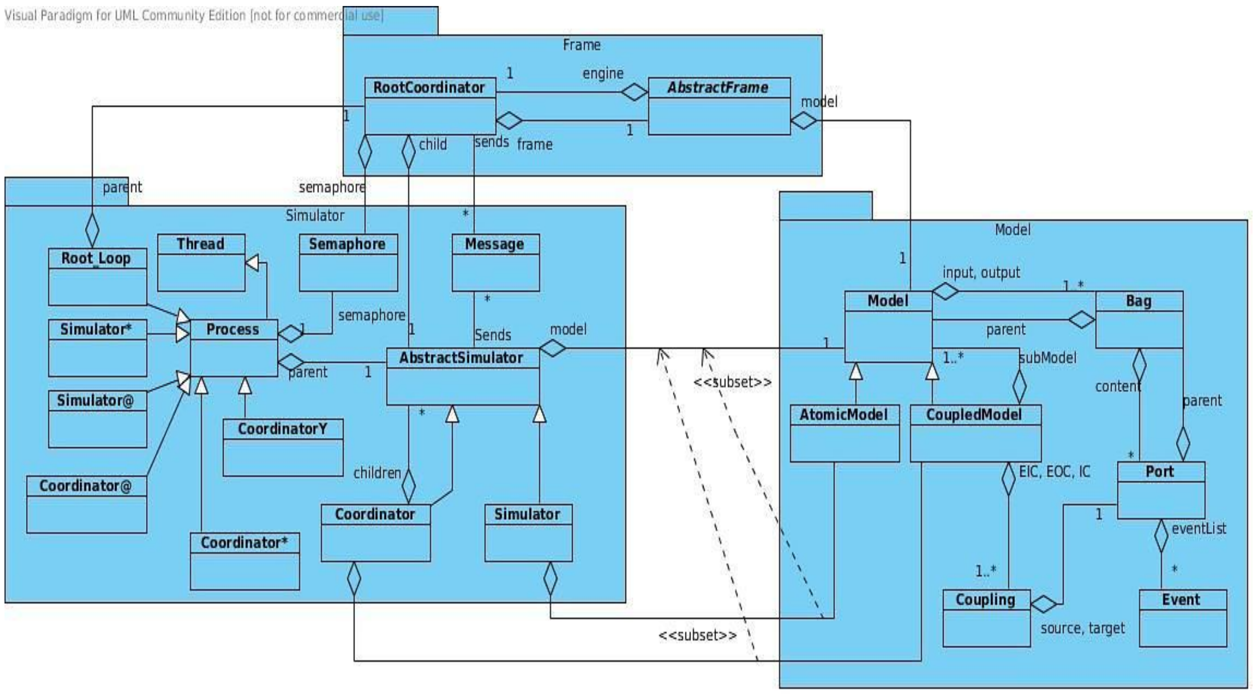


Figure 22: Class Diagram of Implementation III

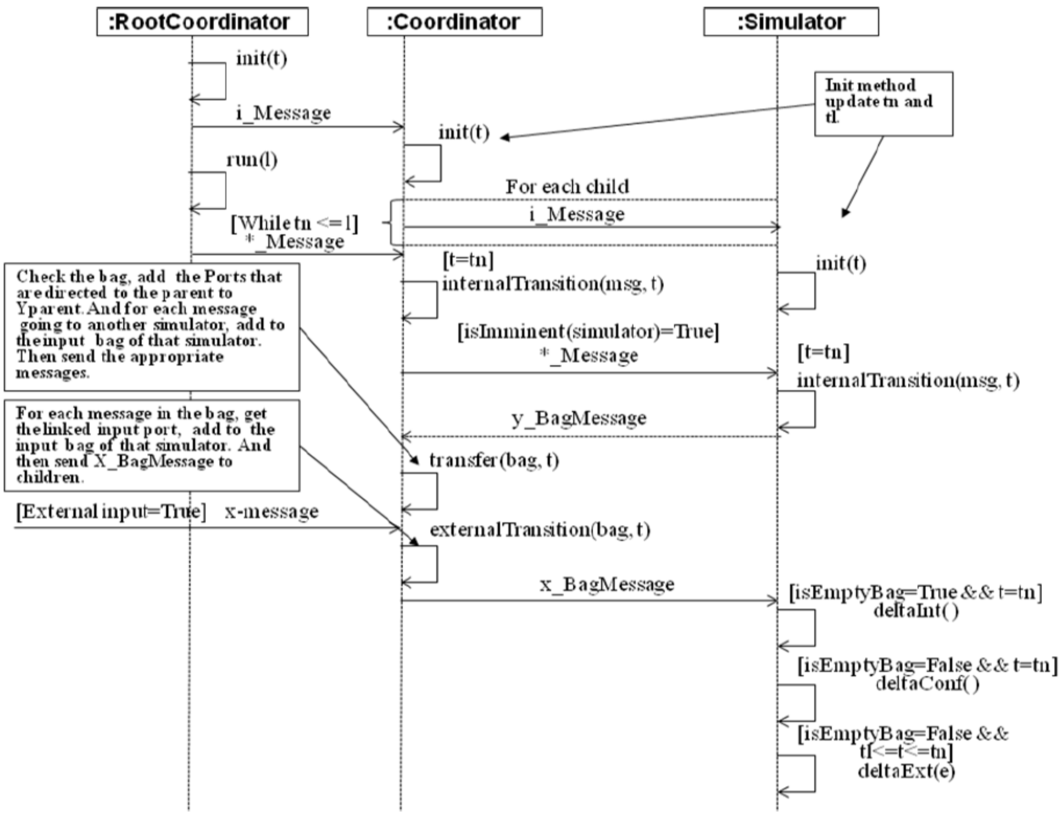

:RootCoordinator
:Coordinator
:Simulator
init(t)
i_Message
init(t)
Init method update tn and tl.
run(l)
For each child
i_Message
[While tn <= l]
* _Message
[t=tn]
internalTransition(msg, t)
init(t)
Check the bag, add the Ports that are directed to the parent to Yparent. And for each message going to another simulator, add to the input bag of that simulator. Then send the appropriate messages.
[isImminent(simulator)=True]
* _Message
[t=tn]
internalTransition(msg, t)
y_BagMessage
For each message in the bag, get the linked input port, add to the input bag of that simulator. And then send X_BagMessage to children.
transfer(bag, t)
[External input=True] x-message
externalTransition(bag, t)
x_BagMessage
[isEmptyBag=True && t=tn]
deltaInt()
[isEmptyBag=False && t=tn]
deltaConf()
[isEmptyBag=False && tl<=t<=tn]
deltaExt(e)

Figure 23: Simulation Sequence Diagram for PDEVS

3.3 Other Implementations

Other PDEVS implementation to solve specific problems includes:

1. The Event Queue Problem and PDEVS: The event queue problem is one of the oldest problems in the field of discrete event simulation. Additional event queue methods which are not part of the standard event queue realizations are used to process models developed.
2. Conflict Management in PDEVS shows that the hypothesis of maximal parallelism does not allow PDEVS to adequately model and simulate systems where simultaneous state transitions are conflicting to one another.
3. Actor-Based Simulation of PDEVS Systems over HLA: Shows a parallel simulation engine with a time management compliant with HLA, which embodies a tie-breaking mechanism for simultaneous events.

3.4 Comparison of approaches

The comparison was done on the already implemented PDEVS simulators discussed in the previous chapter: SimStudio1_1_1, threaded and non-threaded PDEVS simulators. However during the analysis, we repeated the simulation several times to analyze each of the simulators. This was done using a uniprocessor.

We expect to get less simulation time for the PDEVS due to exploitation of parallelism, which is actually the case, the other two simulators shows almost the same behaviour in terms of the duration of simulation at each run. There was an improvement in the time spent in the non-threaded PDEVS simulator when compared to the SimStudio1_1_1, thereby 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 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 by the choice of the model that was used for testing and the amount of messages that were exchanged in the simulator during simulation.

3.5 Problems with existing implementations

The PDEVS simulation engine has been implemented, but there still exists some bugs that needs to be corrected, as well as the need to use formal method to validate the correctness of both the threaded and the thread – less implementation. The existing simulator also have issues with communication overhead and performance.

CHAPTER 4: FORMAL METHODS

4.1 Introduction To Formal Methods Concepts, Approaches and Formalism

A method is said to be formal if it has a well – defined mathematical basis, given by a formal specification language. The mathematical basis therefore provides the means of precisely defining notions like consistency, completeness, specification, implementation and correctness, more relevantly it provides the means of proving that a specification is reliable and has been implemented correctly.

Formal methods consist of writing formal descriptions, analyzing those descriptions and in some cases producing new descriptions. Formal method can be used at any stage of the system development to expose design flaws, ambiguity, incompleteness and inconsistency in a system, and when formal methods are used later they help to determine the correctness of a system implementation. A formal method should posses a set of guiding principle that tells the user the circumstance under which the method can and should be applied as well as how it can be applied effectively. A real product of applying formal method is formal specification, since a formal method is a method and not just a computer program, it may or may not have tool support.

Formal Methods is the use of ideas and techniques from mathematics and formal logic to specify and reason about computing systems to increase design assurance and eliminate defects. Formal Methods tools allow comprehensive analysis of requirements and design and complete exploration of system behavior, including fault conditions. Formal Methods provides a disciplined approach to analyzing complex safety critical systems.

A formal specification, on the other hand, is a description that is abstract, precise and in some senses complete. The abstraction allows a human reader to understand the big picture; the precision forces ambiguities to be questioned and removed; and the completeness means that all aspects of behavior, for example error cases are described and understood. Second, the formality of the description allows us to carry out rigorous analysis. By looking at a single description one can determine useful properties such as consistency or deadlock-freedom. By writing different descriptions from different points of view one can determine important properties such as satisfaction of high level requirements or correctness of a proposed design.

Proof is no more a guarantee of correctness than testing, and in many cases far less of one. Formal methods are descriptive and analytic: they are not creative. There is no such thing as a formal design process, only formal ways of describing and analyzing designs. So we must combine formal methods with other approaches if we actually want to build a real system. formal methods contribute to demonstrably cost-effective development of software with very low defect rates. It is economically perverse to try to develop such software without using them. The reason that, contrary to popular belief, formal methods actually save money since formal methods help us discover errors early in the lifecycle, they actually reduce the overall cost of the project.

WHAT IS FORMAL METHOD?

Formal methods are mathematical techniques, often supported by tools, for developing software and hardware systems. Mathematical rigor enables users to analyze and verify these models at any part of the program life-cycle: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution.
Some examples of commonly known formal method includes:

1. SML: Standard Meta-Language is a strongly typed functional programming language originally designed for exploring ideas in type theory. SML has become the formal methods workhorse because of its strong typing and provability features.
2. HOL: Higher Order Logic, is an automated theorem proving system. As with most automated theorem proving systems, HOL is a computer-aided proof tool: it proves simple theorems and assists in proving more complicated statements, but is still dependent on interaction with a trained operator. HOL has been extensively used for hardware verification, the VIPER chip being a good example.
3. Petri Nets: Petri Nets are a good example of a very 'light' formal specification. Originally designed for modeling communications, Petri Nets are a graphically simple model for asynchronous processes.
4. Z : is based on set theory
5. VDM: Supports a model – oriented specification style and defines a set of built-in data types, which specifiers use to define other types.
6. Larch: is a property – oriented method that combines both axiomatic and algebraic specifications into a two tiered specification.
7. Temporal Logic: is a property – oriented method for specifying properties of concurrent and distributed systems.
8. CSP: CSP uses a model oriented method for specifying concurrent processes and a property oriented method for stating and proving properties about the model.
9. Transition axioms: Lamport's transition axiom method combines an axiomatic method

for describing the behavior of individual operation with temporal logic assertions for specifying safety and aliveness properties.

Formal verification is the process of using formal methods to prove the existence of user required properties in the proposed model of the system, i.e. to prove that the model is correct. A process of applying a manual or automatic formal technique for establishing whether a given system satisfies a given property or behaves in accordance to some abstract description (formal specification) of the system.

FORMAL METHODS CONCEPTS

Formal techniques include: Formal Specifications, Formal Proofs, Model Checking and Abstraction.

Formal Specifications: Translation of a non-mathematical description (diagrams, tables, English text) into a formal specification language.

- Concise description of high-level behavior and properties of a system
- Well-defined language semantics support formal deduction about specification

What is a formal specification language?

A formal specification language provides a formal method's mathematical basis. An example of a formal specification language is the Backus-Naur form. To write a correct specification is very difficult possibly as difficult as writing a correct program, because a specification needs to be adequate, consistent, unambiguous, complete and minimal.

Benefits of Formal Specifications

- Higher level of rigor enables a better understanding of the problem
- Defects are uncovered that would likely go unnoticed with traditional specification Methods
- Identify defects earlier in life cycle
- It can guarantee the absence of certain defects
- Formal specification language semantics allow checks for self-consistency of a problem specification
- Formal specifications enable formal proofs which can establish fundamental system properties and invariants

- Repeatable analysis means reasoning and conclusions can be checked by colleagues
- Encourages an abstract view of system, focusing on what a proposed system should accomplish as opposed to how to accomplish it
- Abstract formal view helps separate specification from design
- Enhances existing review processes by adding a degree of rigor

Formal Proofs

- Complete and convincing argument for validity of some property of the system description
- Constructed as a series of steps, each of which is justified from a small set of rules
- Eliminates ambiguity and subjectivity inherent when drawing informal conclusions
- May be manual but usually constructed with automated assistance

Model Checking

Model Checking is a formal verification technique, which is based on the exhaustive exploration of a given state space trying to determine whether a given property, expressed as a temporal logic formula, is satisfied by a system.

- Operational rather than analytic
- State machine model of a system is expressed in a suitable language
- Model checker determines if the given finite state machine model satisfies requirements expressed as formulas in a given logic
- Basic method is to explore all reachable paths in a computational tree derived from the state machine model

Abstraction

- Simplify and ignore irrelevant details
- Focus on and generalize important central properties and characteristics
- Avoid premature commitment to design and implementation choices

FORMAL METHODS APPROACHES

Formal design can be seen as a three step process, following the outline given here:

- Specification language: A well-defined syntax and semantics is the first step towards the development of tools for formal methods. The specification language that most formal methods use is a mixture of diagrammatical and mathematical notation.

- Formal Specification: During the formal specification phase, the engineer rigorously defines a system using a modeling language. Modeling languages are fixed grammars which allow users to model complex structures out of predefined types. This process of formal specification is similar to the process of converting a word problem into algebraic notation. In many ways, this step of the formal design process is similar to the formal software engineering technique developed by Rumbaugh, Booch and others. At the minimum, both techniques help engineers to clearly define their problems, goals and solutions.
- Verification: As stated above, formal methods differ from other specification systems by their heavy emphasis on provability and correctness. By building a system using a formal specification, the designer is actually developing a set of theorems about his system. By proving these theorems correct, the formal Verification is a difficult process, largely because even the simplest system has several dozen theorems, each of which has to be proven. Even a traditional mathematical proof is a complex affair, Wiles' proof of Fermat's Last Theorem, for example, took several years after its announcement to be completed. Given the demands of complexity and Moore's law, almost all formal systems use an automated theorem proving tool of some form. These tools can prove simple theorems, verify the semantics of theorems, and provide assistance for verifying more complicated proofs.
- Implementation: Once the model has been specified and verified, it is implemented by converting the specification into code. As the difference between software and hardware design grows narrower, formal methods for developing embedded systems have been developed. LARCH, for example, has a VHDL implementation. Similarly, hardware systems such as the VIPER and AAMP5 processors have been developed using formal approaches.

4.2 Benefits Of Formal Methods

The benefits of using Formal Methods include:

1. Product-focused measure of correctness: The use of Formal Methods provides an objective measure of the correctness of a system, as opposed to current process quality measures.
2. Early detection of defects: Formal Methods can be applied to the earliest design

artifacts, thereby leading to earlier detection and elimination of design defects and associated late cycle rework.

3. Guarantees of correctness: Unlike testing, formal analysis tools such as model checkers consider all possible execution paths through the system. If there is any way to reach a fault condition, a model checker will find it. In a multi-threaded system where concurrency is an issue, formal analysis can explore all possible inter-leavings and event orderings. This level of coverage is impossible to achieve through testing.
4. Analytical approach to complexity: The analytical nature of Formal Methods is better suited for verification of complex behaviors than testing alone. Provably correct abstractions can be used to bound the behavioral space of systems with adaptive or non-deterministic behaviors.
5. Formal methods are not intended to guarantee absolute reliability but to increase the confidence on system reliability. They help minimizing the number of errors and in many cases allow finding errors impossible to find manually.

It is well known that the early activities in the lifecycle are the most important. According to the 1995 Standish Chaos report [3], half of all project failures were because of requirements problems. It follows that the most effective use of formal methods is at these early stages: requirements analysis, specification, high-level design. For example it is effective to write a specification formally rather than to write an informal specification then translate it. It is effective to analyze the formal specification as early as possible to detect inconsistency and incompleteness.

4.3 Survey of tools and methods

FORMAL TOOLS

There are many tools available that support formal methods, but this review will be focusing on formal method tools for Java, such as PMD, FindBugs, JLint, ESC/Java2, Bandera, Java Path Finder, Check Style etc., which intend to increase the productivity and accuracy in all the phases of the formal development of systems. However, these tools vary in their capabilities and properties, the extent to which they are used in industry and the extent to which they are able to support most of the stages of formal development. There are different tools for different programming language.

FindBugs Tool
FindBugs looks for bugs in Java programs. It is based on the concept of bug patterns. A bug pattern is a code idiom that is often an error. Bug patterns arise for a number of reasons:

- Difficult language features
- Misunderstood API methods
- Misunderstood invariants when code is modified during maintenance
- Variety mistakes: typos, use of the wrong boolean operator

FindBugs can be run as a standalone application and it can also be install as a plug in to Eclipse, NetBeans etc. FindBugs uses static analysis to inspect Java bytecode (already compiled Java Code) for occurrences of bug patterns. FindBugs finds real errors in most Java software, its analysis is sometimes imprecise, and FindBugs can report false warnings, which are warnings that do not indicate real errors. Survey analysis shows that the rate of false warnings reported by FindBugs is generally less than 50%.
FindBugs is free software, available under the terms of the Lesser GNU Public License. It is written in Java, and can be run with any virtual machine compatible with Java 5. It can analyze programs written for any version of Java. FindBugs was originally developed by Bill Pugh. It is maintained by Bill Pugh, David Hovemeyer, and a team of volunteers.

ESC/Java2 Tool: "Extended Static Checker for Java," is a programming tool that attempts to find common run-time errors in Java programs at compile time. The underlying approach used in ESC/Java is referred to as extended static checking, which is a collective name referring to a range of techniques for statically checking the correctness of various program constraints.

JLint Tool: Analyzes Java bytecode, performing syntactic checks and dataflow analysis. JLint also includes an interprocedural, inter-file component to find deadlocks by building a lock graph and ensuring that there are never any cycles in the graph. JLint is not easily expandable.

PMD Tool: Performs syntactic checks on program source code, but it does not have a dataflow component. In addition to some detection of clearly erroneous code, many of the “bugs” PMD looks for are stylistic conventions whose violation might be suspicious under some circumstances.

Bandera Tool: is a verification tool based on model checking and abstraction. To use Bandera, the programmer annotates their source code with specifications describing what should be checked, or no specifications if the programmer only wants to verify some standard synchronization properties. In particular, with no annotations Bandera verifies the absence of deadlocks. Bandera includes optional slicing and abstraction phases, followed by model checking. Bandera can use a variety of model checkers, including SPIN and the Java PathFinder.

CHAPTER 5: SIMSTUDIO AND FORMAL METHODS

5.1 SimStudio

SimStudio is an operational framework that must serve to capitalize theoretical advances in Modeling and Simulation (M&S) as well as to gather M&S tools and make them accessible through a web browser. From a software perspective, SimStudio is a middleware for the federation of simulators and the collaborative building of simulations. From a hardware perspective, SimStudio is a mean to aggregate intensive computing resources through the http protocol. (Traoré, 2008).

5.2 Improvements on SimStudio (meta-models and discussions)

Creating models and analyzing simulation results can be a difficult and time-consuming task, especially for non-experienced users. Although several DEVS simulators have been developed, the software that aids in the modeling and simulation cycle still requires advanced development skills, and they are implemented using non-standard interfaces, which makes them difficult to extend.

The threaded and the non-threaded packages are both from the PDEVS algorithm (Pawletta, Schwatinski, 2010). The implemented PDEVS threaded simulator coordination was reviewed, and the following improvement was made:

1. The new technique uses a non-hierarchical approach that simplifies the structure of the simulator and reduces the communication overhead. The results obtained allowed us to achieve considerable speed-ups. Distributed simulation can speed-up the execution of models significantly.
2. In order to kill the threads during simulation and reduce overhead. The non threaded was not supposed to be done in a way that concurrency will be lost, which the PDEVS algorithm was meant to use, but we figured out that in the real sense the messages sent to the bags are being treated one after the other if you use a single system for simulation, thus we were not supposed to gain much if we use threads so we went ahead to use threads, maybe there will an improvement if more than one system is used.
3. We introduced a new simulation algorithm and present partitioning and load balancing techniques that are tailored to the efficient distributed execution of PDEVS. We base our elaborations on the idea of minimizing inter-processor communication, since this is a major bottleneck in distributed PDEVS simulation. Additionally, experimental

results are provided which compare the performance of this new approach to alternative algorithms.

Class Diagrams

The metal models remain the same for both the threaded and the thread-less implementation.

Some explanation about the newly Improved PDEVS packages

Model Package

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 ArrayList<Input> getAllInputPorts ( ) : Returns the list of input ports of a model. ArrayList<Output> getAllOutputPorts ( ) : Returns the list of output ports of a model. In place of the addInputPortStructure and addOutputPortStructure that add an input and output structures to the model, and there corresponding getter methods return the structures in PDEVS.
- To add data on an 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

- AtomicModel keep track of the states of the atomic model. The abstract methods deltaInt, deltaExt, deltaConf , lamda, and ta are for the Modeler to implement in such a way that it suits the 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.
- Then deltaConf do the work of the selecting Model.

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.

Simulator Package

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, x-BagMessage or y-BagMessage) and then call the appropriate method which can be an internal Transition, or external Transition, or init, or transfer.
- 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

- 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 update Tn update tn_ (time of next event).
- void addToYparent (Port p): Add a port that is directed to the parent simulator to Yparent. ArrayList<Port> getYparent ( ): Returns Yparent for the simulator.
- Internal Transition, external Transition and transfer send and receive messages based on the PDEVS simulator algorithm.

Simulator

- The Simulator has model_ which is the atomic model it simulates.
- As in the case of the coordinator the internal Transition, external Transition and transfer send and receive messages based on the PDEVS 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.

Message Package

Message

- This is a standard class, used to define common specifications of the messages. All the other classes(*_message, X_message and Y_message) in the package just call the constructor of this class.
- **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.

The Utils, Exception and Type Package remain the same as implemented in the CDEVS we just worked on the message, model and Simulator package.

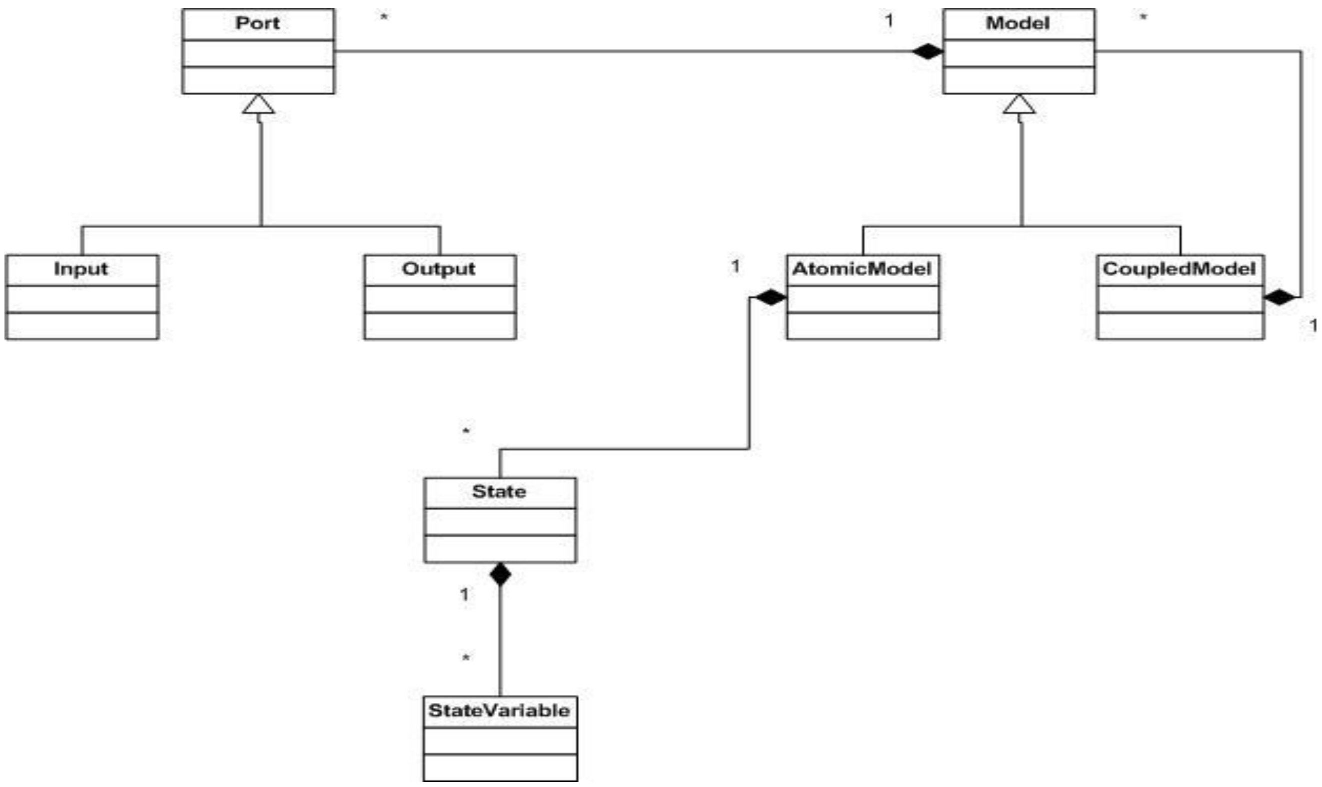


Figure 24: Model Class Diagram

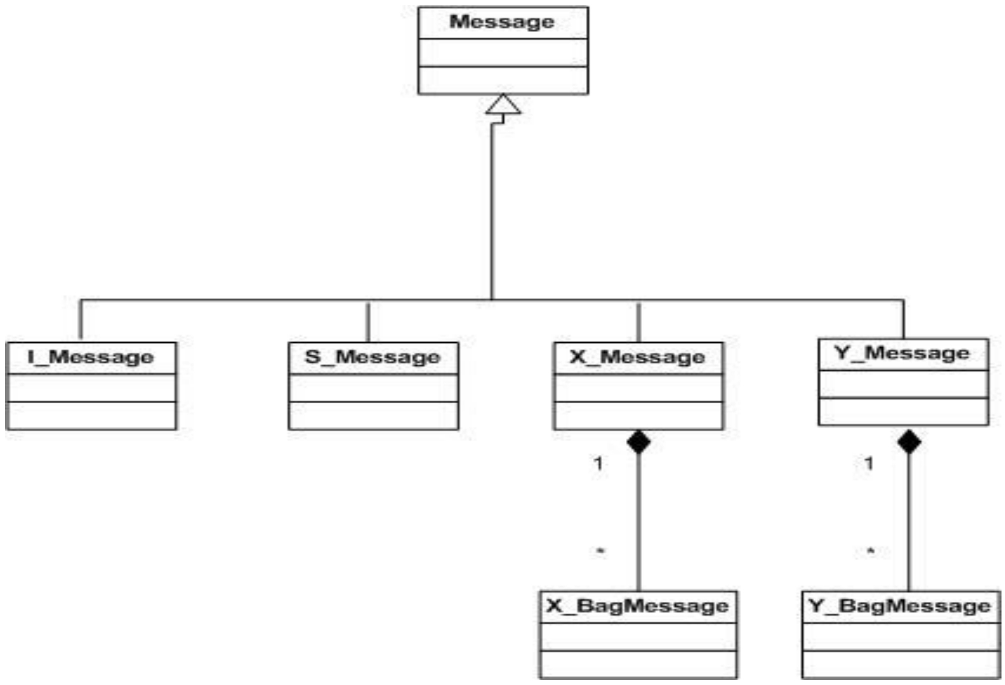


Figure 25: Message Class Diagram

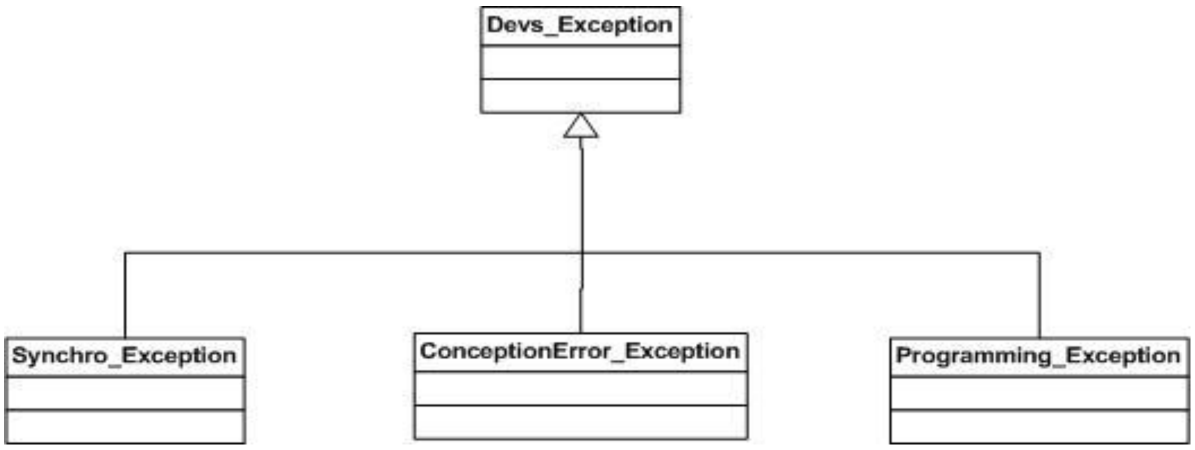


Figure 26: Exception Class Diagram

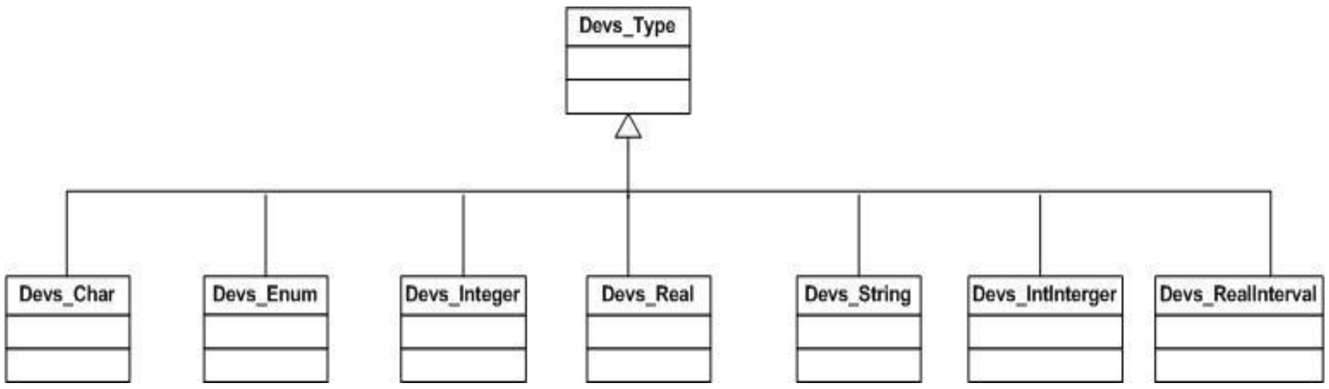


Figure 27: Type Class Diagram

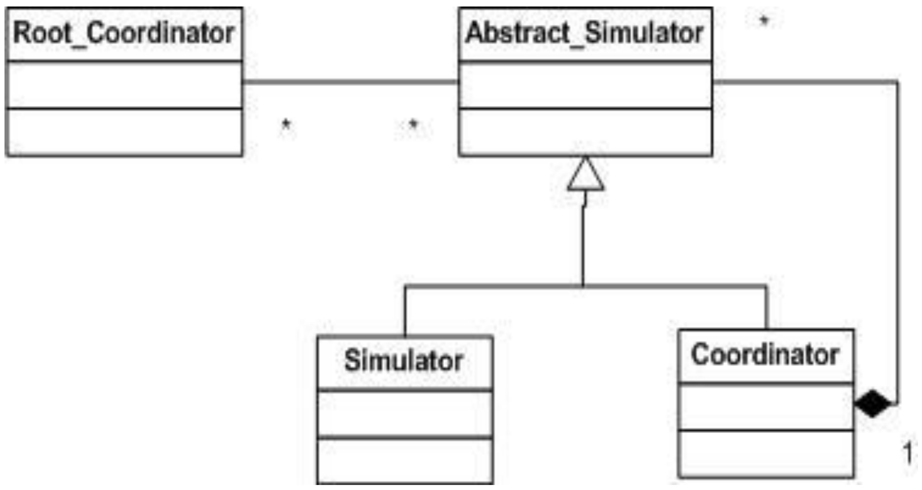


Figure 28: Simulator Class Diagram

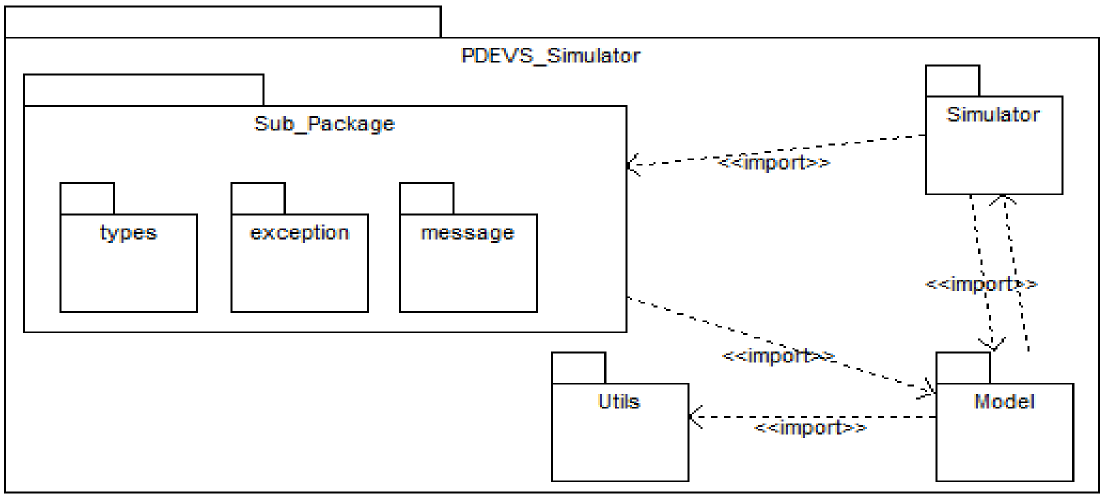


Figure 29: PDEVS Simulator Package Diagram

5.6 Towards Integration Of Formal Analysis With Simstudio

Formal = Mathematical

Methods = Structured Approaches, Strategies

Using mathematics in a structured way to analyze and describe a problem.

Formal method is use for software verification and model checking, using formal method requires knowledge of mathematics in the following areas:

- Set theory
- Functions and Relations
- First-order predicate logic
- Before-After predicates

Our goal is to integrate Formal Analysis with SimStudio, because the specification language of formal tools will make use of short notation, forces you to be precise, helps to Identify ambiguity, gives room for Clean form of communication and Makes you ask the right questions.

Formal method is not programming, because Programming describes a solution and not a problem and Programming is constructive

Using Formal methods is not design, because we do not only describe the software, we describe the full system (software and environment) and there is no separation between software and environment. We do so in an incremental way and that helps us to understand the system.

There are numerous languages out there for formal method tools and most tools invent their own language, nearly all are based on the same mathematical concepts.

For this work the tool used for now is FindBug tool which we discussed in the previous chapter.

5.7 Use Of Formal Tools With Simstudio

Formal methods treat system components as mathematical objects and provide mathematical models to describe and predict the observable properties and behaviours of these objects.

There are several advantages to using formal tools for the specification and analysis of PDEVS Simstudio system.

1. The early discovery of ambiguities, inconsistencies and incompleteness in informal requirements
2. The automatic or machine-assisted analysis of the correctness of specifications with respect to requirements
3. The evaluation of design alternatives without expensive prototyping

5.8 Results and Discussions

Two different implementation

- The threaded PDEVS simulator implementation and
- The thread-less PDEVS simulator implementation

That have reduced communication overhead and increased performance, we had also used FindBug tool to check for errors and ensure that the bugs are properly fixed. Formal method tool is intended to be use to perform model checking and/or theorem proving on the P-DEVS simulation system to proof the properties of correctness.

The benefits of the improved implementation that we have presented is that we have achieved our aim for using thread, which is to have a system that is fast, this was done by killing each thread as soon as they complete their processes.

CHAPTER 6: CONCLUSIONS

6.1 Summary of work

Our work discussed DEVS formalism and its operational semantics through various implementations strategies. We studied the different implementations of Parallel DEVS in an effort to improve the current implementations, so that execution of models can be fast using parallel simulation.

The new implementation was then subjected to formal methods, to perform model checking and/or theorem proving on the P-DEVS simulation system so that properties of correctness can be assessed.

Using a model to test each implementation, a large scale application was built (Cross Road) and the evaluation and comparison of the performances of our implementations shows that;

the PDEVS implementation III have high-speed than the PDEVS implementation II, that is the thread in implementation III reduce the execution time and consumed less computer memory than the other implementations.

6.2 Challenges

The major challenge we faced was with the existing implementation, it was tasking to understand how it works, what the problem was, that is why the threaded implementation was not giving us the desired performance. Secondly we had little challenge modeling our case study (Cross Road) Atomic models.

6.3 Future work

We achieved studying the formalism and its operational semantics through various implementations strategies by evaluating the efficiency of the simulation results. The Formal analysis of the simulation protocol was however not totally complete.

In future we need to use formal method tool to proof the correctness of the PDEVS simulation protocol which involves; chosen a tool that suits our application, downloading the tool, learning how the tool works, learning the tool specification language, writing the specification of the PDEVS simulation protocol (our application) using the language learnt and then feeding the specification into the formal method tool as an input and getting the output. So as to properly do the formal analysis of the simulation protocol.

REFERENCES

(1). Gabriel A. Wainer. 2009. ***Discrete-Event Modeling and Simulation***. A Practitioner's Approach by Taylor & Francis Group, LLC.

(2). Zeigler, B.; Sarjoughian S. 2003. ***Introduction to DEVS modeling and simulation with Java$^{TM:}$ Developing component – based simulation models***.

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

(4). Zeigler, B.; and Hessam, S; Sarjoughian. 2005. ***Introduction to DEVS Modeling and Simulation with JAVA: Developing Component - Based Simulation Models*** January (draft version)

(5). Zeigler, B. et al, *2000*. ***Theory of modeling and simulation,*** 2nd edition. New York: Academic Press,

(6). Chow, A. et al. 1994. **"*Parallel DEVS: A parallel, hierarchical, modular modeling formalism.*"** Proceedings of the Winter Computer Simulation Conference. Orlando, FL. USA.

(7). Unified Modeling Language, www.omg.org/uml/

(8). Douglas, W. Jones. 1986. ***Implementations of Time***, Proceedings of the 18th winter Simulation Conference,

(9). Ighoroje. B, et al. 2010. ***The DEVS Driven Modeling Language***,

(10). Wikipedia 2007. http://en.wikipedia.org/wiki/DEVS.

(11). Zeigler, Bernard P. 1976. ***Theory of Modeling and Simulation (First Edition)***. Wiley Inter-science, New York.

(12). Nick, R. et al. ***Comparison of Bug Finding Tools for Java.*** University of Maryland, College Park.

(13). Jim, W. et al. ***Formal Methods: Practice and Experience***. Newcastle University.

(14). Michael, C. 1998. ***Formal Methods***. Carnegie Mellon University. Spring, http://www.ece.cmu.edu/~koopman/des_s99/formal_methods/

(15). Jeannette, M. 1990. ***A Specifier's Introduction to Formal Methods***. Carnegie Mellon University,

(16). Axel, V. 2000. ***Formal Specification: a Roadmap***. Université catholique de Louvain,

(17). Kefalas, P.et al**. *Developing Tools for Formal Methods.*** City Liberal Studies, Affiliated College of the University of Sheffield, Computer Science Department.

(18). Murali, R. ***Formal Methods Analysis of complex systems to ensure correctness and***

reduce cost. Honeywell Laboratories. Minneapolis.

(19). Anthony, H. ***Realising the benefits of formal methods***. Independent consultant UK

(20). Lecture Slide. – ***Introduction to formal methods***

(21). Traore, M. K. 2008. **—*SimStudio: a Next Generation Modeling and Simulation Framework***. Proceedings from the Spring Simulation Multiconference.