30th November 2011

Formal and Operational Study of C-DEVS

Master's Thesis in Computer Science, 30th November 2011

African University of Science and Technology, Abuja

BY

IGNACE DJITOG

SUPERVISOR:

PROF. MAMADOU KABA TRAORE

To

*my family **Mahamat Tanro**, making me who I am now with her love and support, and to all my friends.*

ACKNOWLEDGEMENTS

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

I am particularly grateful to my supervisor, **Prof. Mamadou Kaba Traore** for his support, guidance and encouragement throughout this work. Also, my appreciation goes to the African University of Science and Technology (**AUST**), Abuja community for providing support and a wonderful research environment.

I would like to express my gratitude to the staff of Universite de Science et de Technologie d'Ati (**USTA**) in **Chad**, particularly **Dr. Ahmat-Charfadine Mahamat** and **Dr. Reounodji Frederick** for giving me opportunity to do my Master.

To my **colleagues** at African University of Science and Technology, they have been very wonderful. Special thanks to my classmates Aroyehun Segun Taofeek,Epie Essongolle Godfred,Ngolah Kenneth Tim,Mensah Kwabe na Patrik,Mohammad Hassan,Godwin Larry Eniwei,Aliyu Hamzat Olanrewaju,Eli- Ake Grace Eyitayo Kehinde,Arreytambe Tabot,Ighile Osayawe,Sorem-ekun Olamide Ezekiel.

Abstract

C-DEVS is a formalism for modeling and analysis of discrete event systems. It refers to the original formalism defined by Zeigler in 1976. While the simulation algorithms are well defined, their implementation is a challenge due to both correctness and efficiency issues. This work aims at studying the formalism and its operational semantics. We review and validate the meta-model for SimStudio - a Java implementation of the DEVS simulation protocol, and we debug its existing Java codes. We finally use formal methods to perform model checking and theorem proving on the C-DEVS simulation system to assess the properties of correctness.

Table of Contents

Table of Figures

Chapter 1. Introduction

Many natural systems in physics, astrophysics, chemistry and biology, and human systems in economics, psychology, social science and engineering have long relied on the unsuitable methods for their study during the twentieth century. Traditional mathematical methods such as differential equations have been used for centuries as the main tools for analysis, comprehension, design, and prediction for complex systems in varied areas.

The emergence of computers simulation provided alternative methods of analysis for both natural and artificial systems. Since the early days of computing, users translated their analytical methods into computer-based simulation to solve with a level of complexity unknown in earlier stages of scientific development.

Computational methods based on differential equations could not be easily applied in studying human-made dynamic systems such as traffic controllers, robotic arms, automated factories, production plants, and computer networks and so on. These systems are usually referred to as discrete-event systems because their states do not change continuously but, rather, because of the occurrence of events. This makes them asynchronous, inherently concurrent, and highly nonlinear, rendering their modeling and simulation different from that used in traditional approaches [1].

A number of techniques were introduced in order to improve the model definition for this class of systems, Petri Nets, Finite state Machines, min-max algebra, Timed Automata, and others [1].

Let us define the following key words.

Model: A model is a simplified representation of a system at some particular point in time or space intended to promote understand of the real system.

System: A system exits and operates in time and space.

Simulation: A simulation is the manipulation of a model in such a way that it operates on time or space to compress it, thus enabling one to perceive the interactions that would not otherwise be apparent because of their separation on time or space.

Modeling and Simulation (M&S) is the use of models, including emulators, prototypes, simulators, and either statically or over time, to develop data as a basis for making

managerial or technical decisions. The primary motivation for modeling and simulation is risk reduction, that is, to ensure that the simulation can support its user/developer objectives acceptably. This is the primary benefit in cost-benefit concerns about Verification and Validation (V & V), which is the core issue in the question of how much V & V is needed.

Modeling and simulation play increasingly important roles in modern life. It contributes to our understanding of how things function and are essential to the effective and efficient design, evaluation, and operation of new products and systems. Modeling and simulation results provide vital information for decisions and actions in many areas of business and government. Verification and validation are processes that help to ensure that models and simulations are correct and reliable. Although significant advances in V&V have occurred in the past 15 years, significant challenges remain that impede the full potential of modeling and simulation made possible by advances in computers and software.

1.1 DEVS formalism and its variants

DEVS (Discrete Event System Specification) defined by Zeigler in the 1970's is one of the existing theories of Modeling and Simulation allowing the modular description of discrete event models. It finds a way to specify systems whose states change either upon the reception of an input event or due to the expiration of a time delay. In order to attack the complexity of the system under study, the model is organized hierarchically (i.e., it is organized in a way such that every element is higher than its precedent), and higher-level components of the system are decomposed into simpler elements.

The separation between model and simulator and the hierarchical and modular nature of the formalism have enabled carrying out of formal proofs on the different entities under study. One of them is the proof of composability of the subcomponents (including legitimacy and equivalence between multicomponent models). The second is the ability to conduct proofs of correctness of the simulation algorithms, which results in simulators rigorously verified. The proofs are based on formal transformations *(morphisms)* between each of the representations, trying to prove the equivalence between the entities under study at different levels of abstraction. For instance, we can prove that the mathematical entity simulator is able to execute correctly the behavior described by the mathematical entity model, which represents the system under the experimental framework (which can also be represented formally).

Numerous extensions of the classic DEVS formalism have been developed in the last decades. Among them formalisms which allow having changing model structures while the simulation time evolves:

G-DEVS, Parallel DEVS, Dynamic Structuring DEVS, Cell-DEVS [2], dynDEVS, Fuzzy-DEVS, GK-DEVS, ml-DEVS, Symbolic DEVS, Real-Time DEVS, rho-DEVS [Wikipedia].

We will present DEVS with further details in the following chapter.

1.2 Formal analysis of DEVS simulation protocol.

The use of formal analysis for DEVS simulation protocol is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of the algorithm. The most likely reason for this is that the formal analysis leads into relatively simple and independent components, which allow for straightforward unit testing of model checking and theorem proving on the C-DEVS simulation protocol to assess the properties of correctness and efficiency. We recall that the simulation protocol of DEVS models considers two issues: time synchronization and message propagation. Time synchronization of DEVS is to control all models to have the identical current time. However, for an efficient execution, the algorithm makes the current time jump to the most urgent time when an event is scheduled to execute its internal state transition as well as its output generation. Message propagation is to transmit a triggering message which can be either an input or output event along the associated couplings which are defined in a coupled DEVS model.

1.3 Structure

We organize our work as follows: Chapter_2 describes the Discrete Event System and Specification for CDEVS and PDEVS, Chapter_3 is the literature review on PDEVS implementations and the use of the Unified Modeling Language (UML) for the implementation of Simstudio (meta- model). In Chapter_4 we introduce the formal methods, the benefits of the formal methods and survey of tools and methods. Chapter_5 describes the use of formal methods and survey of tools with Simstudio then Chapter_6 concludes our work and presents challenges and the future work to be done.

Chapter2. Discrete Even System Specification (DEVS)

2.1. DEVS (CDEVS and PDEVS with differences)

The Discrete Event System Specification (DEVS) is a modular and hierarchical formalism for modeling and analyzing general systems invented by Bernard Zeigler and was introduced to the public in his first book “Theory of Modeling and Simulation” in 1976.

DEVS provides means of specifying a mathematical object called a system. Basically, a system has a time base, inputs, states, outputs, and functions for determining next states and outputs given current states and inputs. The inputs in discrete even system occur at arbitrarily spaced moments, while those in continuous systems are piecewise continuous functions of time. Thus the insight provided by the DEVS formalism is in the simple way that it characterizes how discrete event simulation languages specify discrete event system parameters.

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

2.2 CDEVS

A real system modeled using DEVS can be described as a composition of atomic and coupled Models [2].

2.2.1 Classic DEVS (CDEVS) Atomic Model

A Discrete Event System Specification is a structure as

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

Where

X = {(p,v) / p ∈ IPorts, v ∈ Xp}

is the set of input events, where IPorts represents the set of input ports and Xp represents the set of values for the input ports;

Y = {(p,v) / p ∈ OPorts, v ∈Yp}

is the set of output events, where OPorts represents the set of output ports and Yp represents the set of values for the output ports;

S is the set of sequential states;

δext: $Q \times X \rightarrow S$

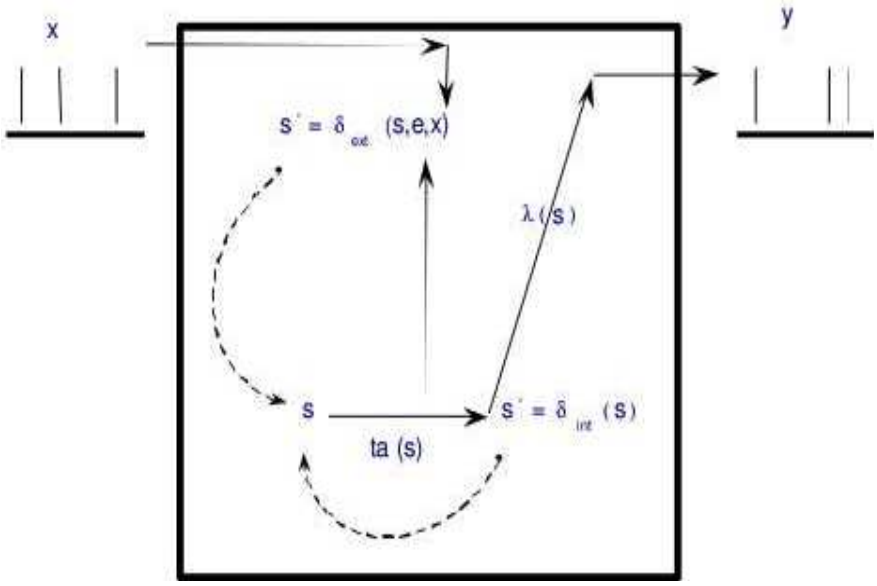
is the external state transition function, with Q = {(s,e)/s ∈S, e ∈ [0, ta(s)]}and e is the elapsed time since the last state transition;

δint: $S \rightarrow S$ is the internal state transition function;

λ: $S \rightarrow Y$ is the output function; and

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

The interpretation of these elements is illustrated in figure below.



DEVS in action 1

Figure2.1. DEVS in action

At any given moment, a DEVS model is in a state s ∈S. In the absence of external events, it remains in that state for a lifetime defined by ta(s). When ta(s) expires, the model outputs the value λ(s) through a port y ∈ γ, and it then changes to a new state given by δint(s). A transition that occurs due to the consumption of time indicated by ta(s) is called

an internal transition. On the other hand, an external transition occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by δext(s, e, x), where s is the current state, e is the time elapsed since the last transition, and x ∈ X is the external event that has been received. The time advance function can take any real value between 0 and ∞. A state for which ta(s) = 0 is called a transient state (which will trigger an instantaneous internal transition). In contrast, if ta(s) = ∞, then s is said to be a passive state, in which the system will remain perpetually unless an external event is received (can be used as a termination condition).

2.2.2 Classic DEVS Coupled Model

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

CM = <X, Y, D, {M_d / d ∈D}, EIC, EOC, IC, select>

Where

X = {(p,v) / p ∈ D IPorts, v ∈ Xp} is the set of input events, where IPorts represents the set of input ports and Xp represents the set of values for the input ports;

Y = {(p,v) / p ∈ OPorts, v ∈ Yp} is the set of output events, where OPorts represents the set of input ports and Yp represents the set of values for the output ports;

D is the set of the component names and for each d ∈ D;

M_d is a DEVS basic (i.e., atomic or coupled) model;

EIC is the set of external input couplings, EIC $\subseteq$ { ((Self , in_{Self}), (j, in_j)) / in_{Self} ∈ IPorts, j ∈ D, in_j ∈ Iportsj};

EOC is the set of external output couplings, EOC $\subseteq$ { ((i,out_i), (Self, out_{Self})) / out_{Self} ∈ OPorts,i ∈ D, out_i ∈ $OPorts_i$};

IC is the set of internal couplings, IC $\subseteq$ {((i,out_i), (j,in_j)) / i,j ∈ D, out_i ∈ $OPorts_i$, in_j ∈ $IPorts_j$ }; and

Select is the tiebreaker function, where select $\subseteq$ D → D, such that, for any nonempty subset E, select (E) ∈ E.

2.3 Examples of CDEVS model and Simulation.

We present an example for CDEVS modeling and simulation. The Generator, Processor, Transducer (GPT) model in Figure 2.2.

The top-level model GPT is a simple coupled model that is composed of two basic components: generator and QPT. Generator is an atomic model that creates jobs to be processed (at random times) and sends them through the out output port. QPT is a coupled model consisting of two main atomic components: a processor that consumes the jobs received (and informs that they are ready through the out output port) and a transducer in charge of calculating statistics. When a new job arrives through the arrived input port, the transducer computes the arrival time; when the job finishes, its end time arrives through the solved port, and we can use this information to compute metrics. In this case, we have also included a queue model, which is used as a buffer for the arriving jobs before they are processed. Based on Figure 2.2, we can define the coupled model for this example as

MGPT = < X, Y, D, {Md / d ∈ D}, EIC, EOC, IC, select>

Where

X = ∅ ;

Y = {(cpuUsage, R_0+); (Throughput, R_0+)};

D = {Generator, QPT};

Md = { MGenerator, MQPT} (Where MGenerator is an atomic model and MQPT a coupled one);

EIC = ∅ ;

EOC $\subseteq$ {((QPT, cpuUsage),(self, cpuUsage),(self, cpuUsage));((QPT, Throughput),(self, Throughput))} ;

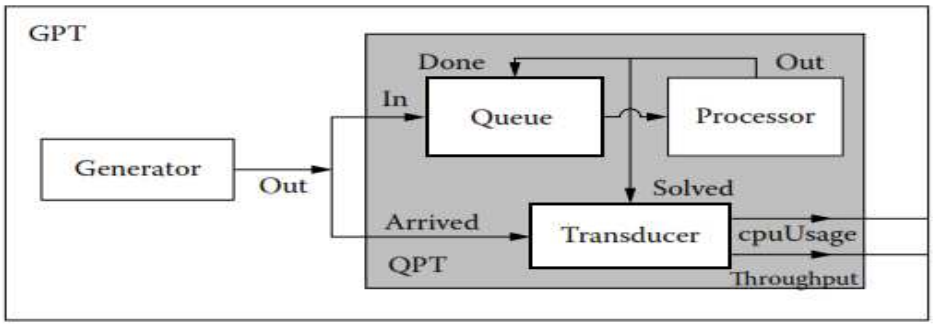


Figure 2.2: ***The GPT model***

IC ⊆ {((Generator, out),(QPT, in));((Generator, out),(QPT, arrived))};and

Select = {QPT, Generator}.

The QPT coupled model can be defined as

$M_{QPT} = <X, Y, D, \{M_d / d \in D\}, EIC, EOC, IC, select>$

Where

X = {(in, N); (arrived, N)};

Y = {cpuUsage, R_0^+); (Throughput, R_0^+)};

D = {Queue, Processor, Transducer};

Md = {M_{Queue}, $M_{Processor}$, $M_{Transducer}$};

EIC = {((Self, in), (Queue, in)); ((Self, arrived), (Transducer, arrived)) };

EOC ⊆ {((Transducer, cpuUsage), (Self, cpuUsage)); ((Transducer, Throughput), (Self, Throughput))};

IC ⊆ {((Queue, out), (Processor, in)); ((Processor, out), (Queue, done)); ((Processor, out), (Transducer, solved))};

Select = {Processor, Queue, Transducer}.

2.4.The CDEVS Simulation Algorithm

An interpretation of the dynamics of a DEVS is given by considering the devs-simulator for DEVS .The simulator employs two time variables *tl* and *tn*. The first holds the

simulation time when the last event occurred and the second holds the scheduled time for the next event. From the definition advance function of DEVS it follows that

$$tn = tl + ta(s).$$

Additionally, if it is given the global current simulation time, *t*, the simulator can compute from these variables the elapsed time since the last event,

$$e = t - tl,$$

and the time left to the next event,

$$\sigma = tn - ta(s) - e.$$

The time of next event, tn is sent to its simulator's parent to allow for correct synchronization of events.

Devs-simulator

variables :

parent *// parent coordinator*

tl *// time of last event*

tn *// time of next event*

DEVS *// associatied model*

With total state (s, e)

y *// current output value of the associated model*

when receive i –message (i, t) at time t

tl = t –e

tn = tl – ta(s)

*when receive * -message (*, t) at time t*

if *t != tn then*

error : bad synchronization

$y = \lambda(s)$

send y-message (y, t) to parent coordinator

$s = \delta_{int}(s)$

tl = t

tn = tl + ta (s)

when receive x –message (x, t) at time t with input value x

if not (tl ≤ t ≤ tn) then

error : bad synchronization

e = t – tl

s = δ_{ext} (s, e, x)

tl = t

tn = tl + ta(s)

end Devs- Simulator

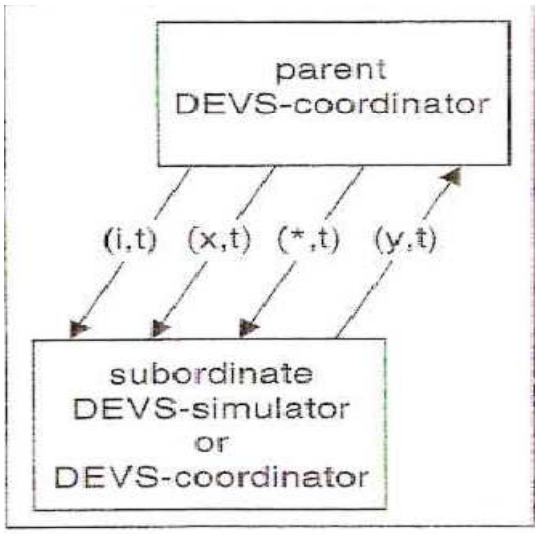
Algorithm Simulator for Basic DEVS.

As shown in the Algorithm above, for correct initialization of the simulator, a initialization message (i, t) has to be received at the beginning of each simulation run. When a devs-simulator receives at the beginning of each simulation run. When a devs-simulator receives such an initialization message, it initializes its time of last event tl by subtracting the elapsed time e from the time t provided by the message. The time of next event tn is computed by adding the value of the time advance ta(s) to the time of last event tl. This time tn is sent to the parent coordinator to tell it when the first internal event should be executed by this component's simulator.

An internal state transition message (*, t) causes the execution of an internal event. When a *-message is received by a DEVS-simulator, it computes the output and carried out the internal transition function of the associated DEVS. The output is sent back to the parent coordinator in an output message (y, t). Finally, the time of the last event tl is set to the current time and the time of the next event is set to the current time plus the value of the advance function ta(s).

An input message (x, t) informs the simulator of an arrival of an input event, x, at simulation time, t. This causes the simulator to execute the external transition function of the atomic DEVS given x and the computed value of elapsed time, e. As with internal events, the time of the last event tl is set to the current time and the time of the next event is set to the current time plus the value of the time advance function ta(s).

The figure bellow illustrates the DEVS simulator protocol.



DEVS Simulation protocol 1

Figure 2.3 : DEVS simulator protocol

2.5. Example of CDEVS model and simulation (by hand)

We apply the CDEVS simulation algorithm described above on a Crossroad as shown in the figure 2.4.

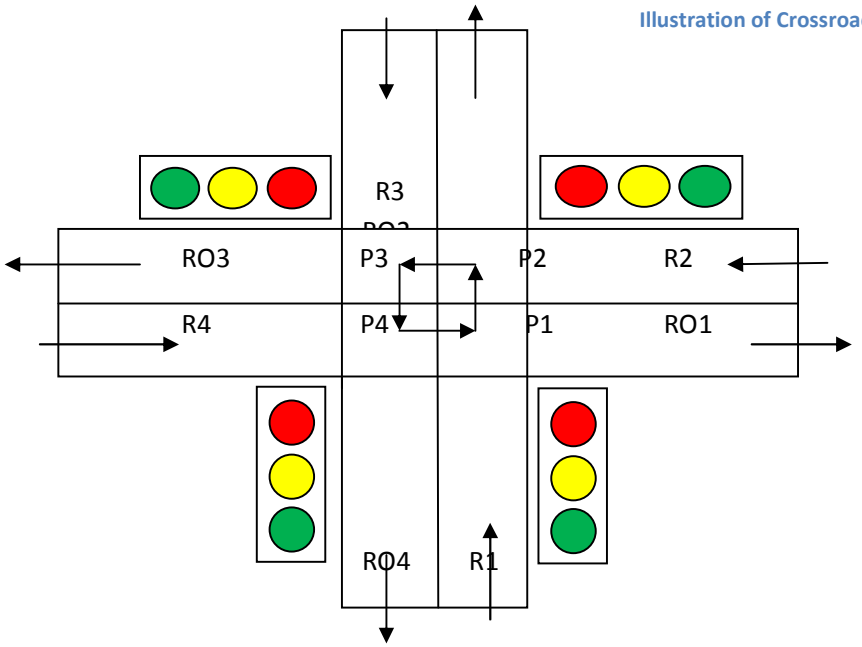


Illustration of Crossroad 1

Figure 2.4 : ***Illustration of Crossroad.***

R: Road; RO: Road out System; P: Place

2.5.1 Simulation by hand

We simulate by hand the crossroad coupled Model and show as bellow.

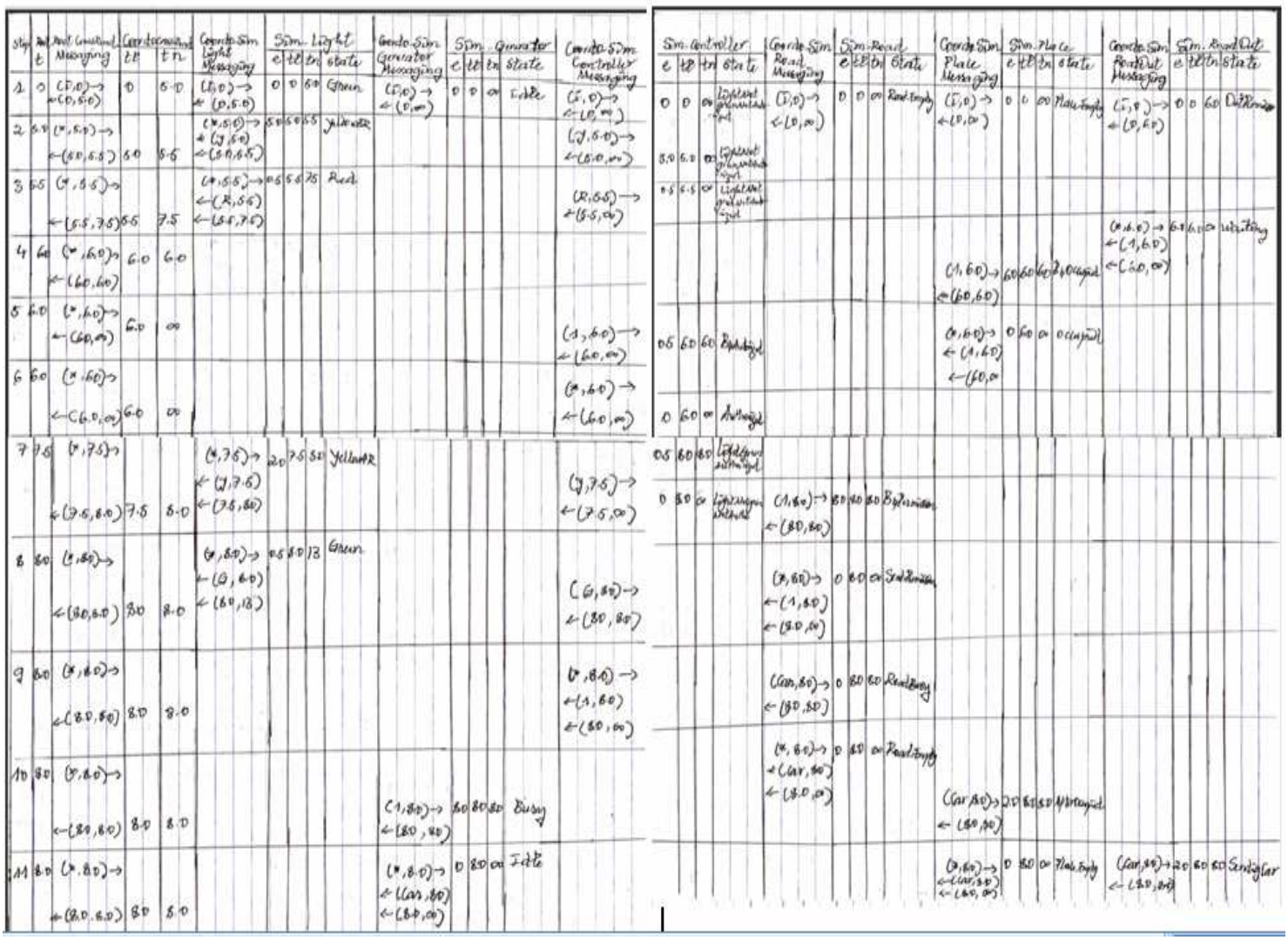


Figure 2.5: ***Simulation by hand of Crossroad.***

2.5.2 Simulation result Snapshot

The crossroad model simulated by hand above is implemented using C-DEVS Simulation algorithm and ran under eclipse. The result is showed in snapshot as follows.

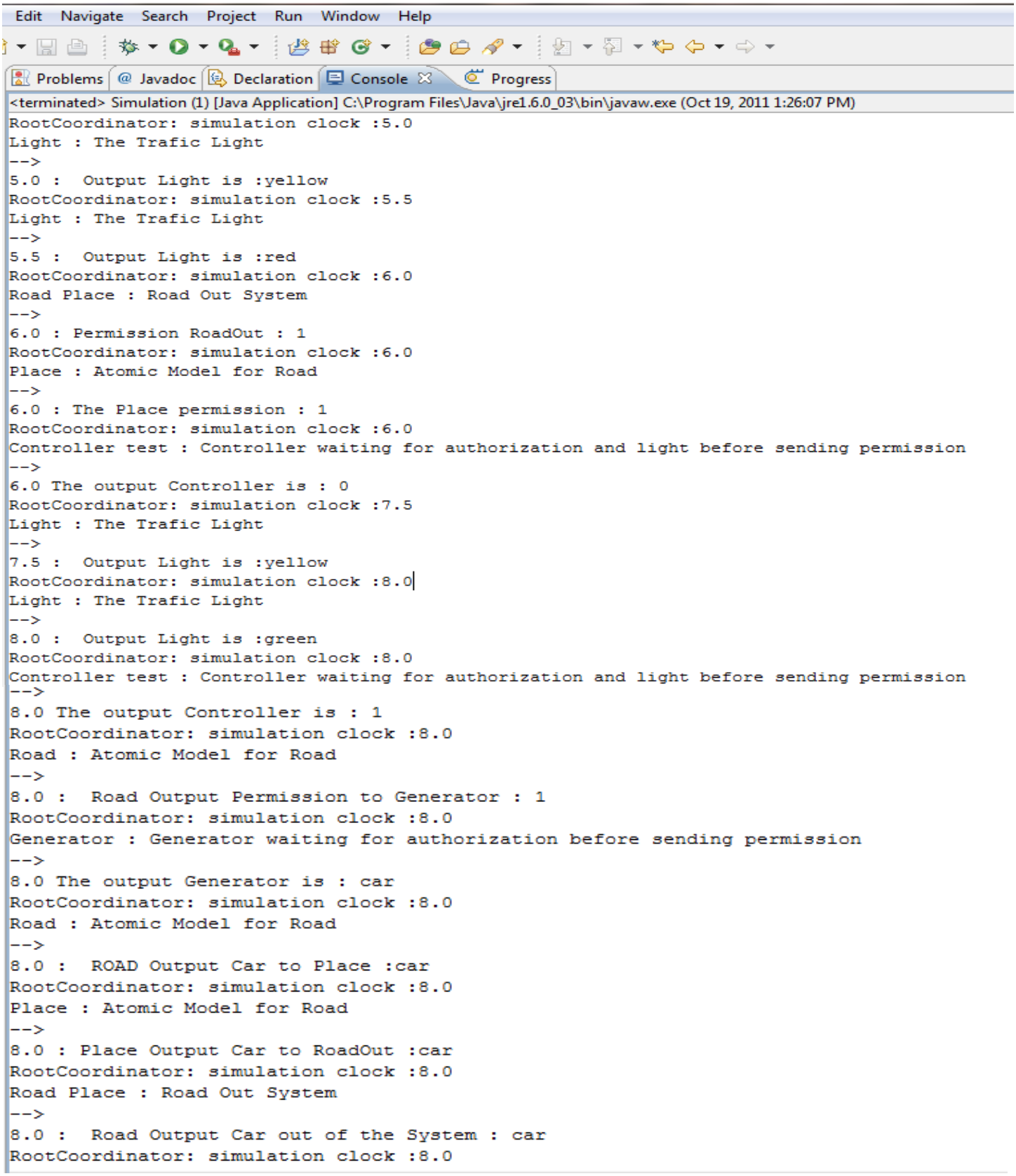


Figure 2.6: ***Simulation result.***

Simulation result 1

Chapter 3. Literature Review on PDEVS Implementations

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 input and properly interpreting it.

3.1 DEVS WITH SIMULTANEOUS EVENTS (PARALLEL DEVS)

3.1.1 PDEVS Atomic Model

Parallel DEVS (or PDEVS) is an extension to DEVS that provides a more flexible way of dealing with the ambiguities. Atomic models provide an additional confluent function to specify collision behavior for events that might be scheduled simultaneously and a mechanism for receiving multiple external events at the same time and processing them together. An atomic PDEVS model is defined as

M = <X, Y, S, δext, δint, δcon, λ, ta>

Where

X = {(p,v) / p ∈ IPorts, v ∈ Xp} is the set of input events, where IPorts represents the set of input ports and Xp represents the set of values for the input ports;

Y = {(p,v) /p ∈ OPorts, v ∈ Yp} is the set of output events, where OPorts represents the set of input ports and Yp represents the set of values for the output ports;

S is the set of sequential states;

δext : $Q \times X^b \rightarrow S$ is the external state transition function;

δint : $S \rightarrow S$ is the internal state transition function;

δcon : $Q \times X^b \rightarrow S$ is the confluent transition function;

λ : $S \rightarrow Y^b$ is the output function;

ta : $S \rightarrow R^+_0 \cup \infty$ is the time advance function, with Q = {(s, e) / s ∈ S, 0 ≤ e ≤ ta(s)} the set of total states.

PDEVS models use bags (multisets) of events for receiving inputs and collecting outputs (X^b, Y^b) instead of a single event. This allows multiple events to be processed simultaneously. Because external input events received by the component are added to the bag, external transition functions can combine the functionality of a number of external transitions into a single one, and simultaneous events (like the departure of a vehicle and a collision in the intersection) can be treated simultaneously. Also, PDEVS allows a better way to deal with collisions: the model specification includes a confluent transition function (δcon). When a collision between the internal and external functions occurs, the confluent function determines the new state of the model.

The flowchart of the simulator that would execute, and generate the behavior of the semantics of the PDEVS model described above is given below.

Semantics of an Atomic PDEVS Model 1

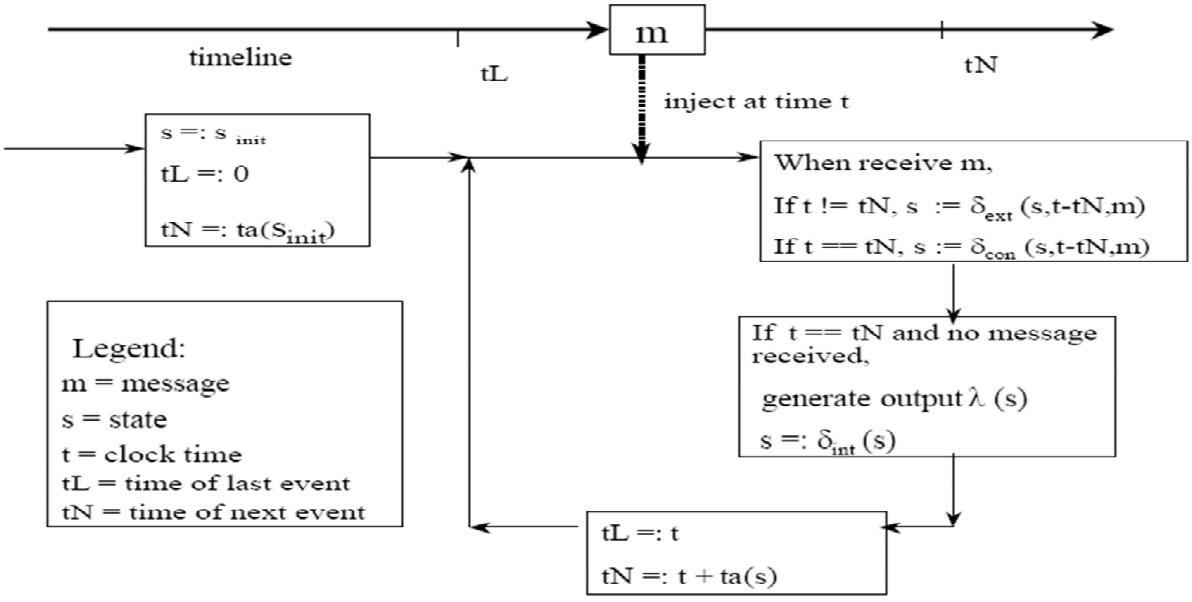


Figure 3.1 Semantics of an Atomic PDEVS Model

3.1.2. PDEVS Coupled Model

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

CM = < X, Y, D, {M_d / d ∈D}, EIC, EOC, IC >

where definitions for the set of input and output events (X and Y), components (D and M_d), and couplings (EIC, EOC, and IC) follow the specifications of C-DEVS coupled models already presented earlier.

3.2 Survey of DEVS Implementations (with examples)

3.2.1 Simstudio implementation (meta – models)

SIMSTUDIO is the virtual machine under development in the LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes) [3].

Let's recall that the Unified Modeling language (UML) is a general purpose modeling language intended and provides a complete modeling framework. Thus, using UML according to the principles of DEVS yields an executable modeling sub-language suitable for a DEVS framework allowing software architects to code in the abstract, meaning that within this framework a designer can experiment with code (e.g. Java) without being lulled into detailed design or implementation issues.

3.2.1.1 DEVS-Meta model: UML Class Diagram

The figure 3.2 bellow shows the Class Diagram of the CDEVS model.

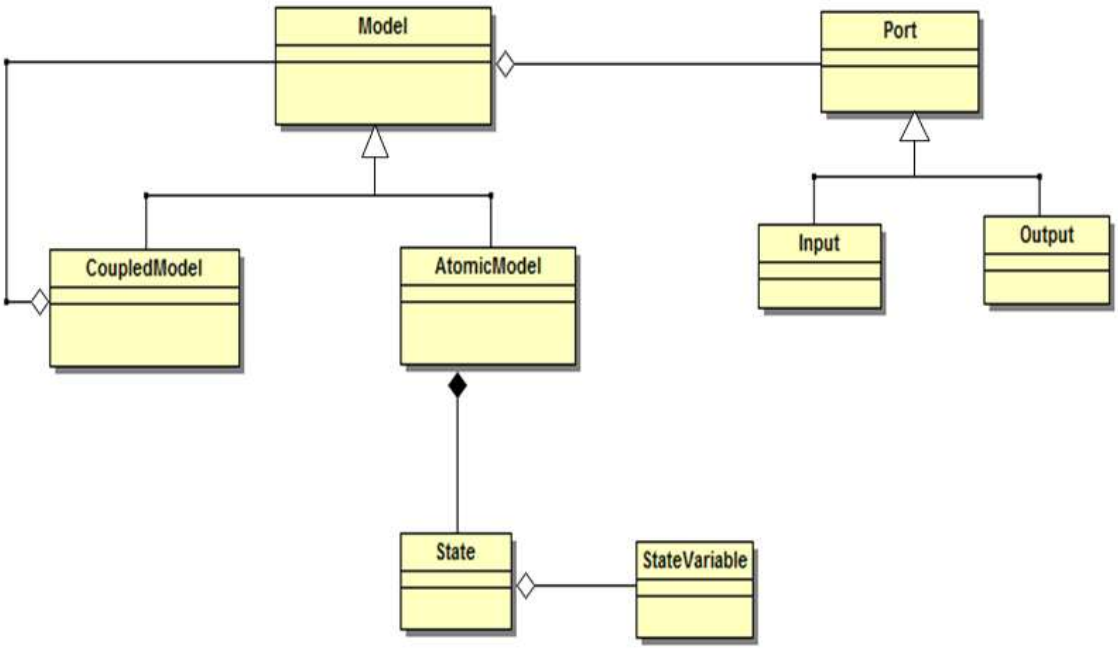
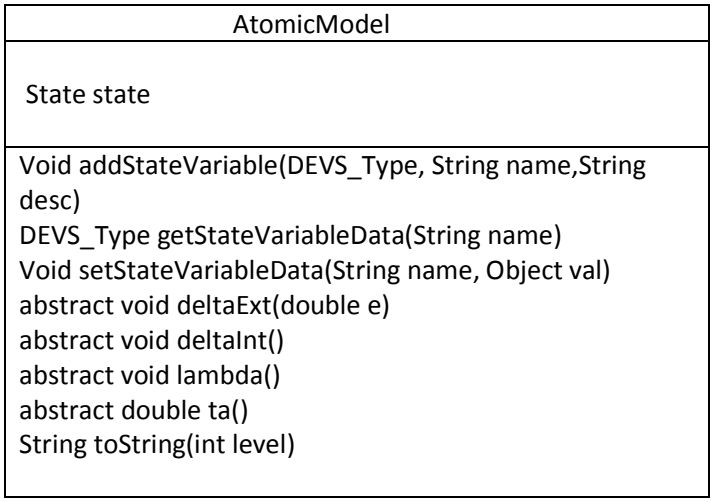
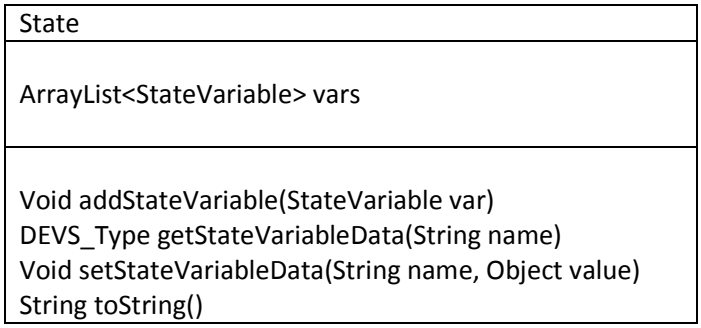
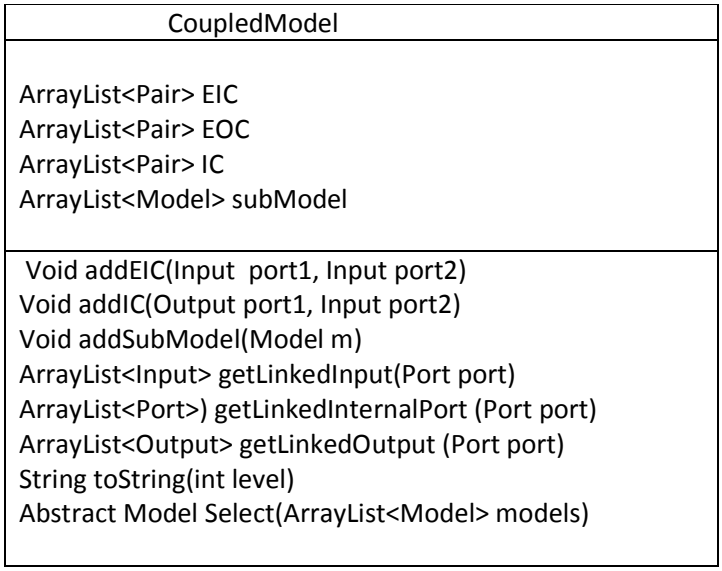
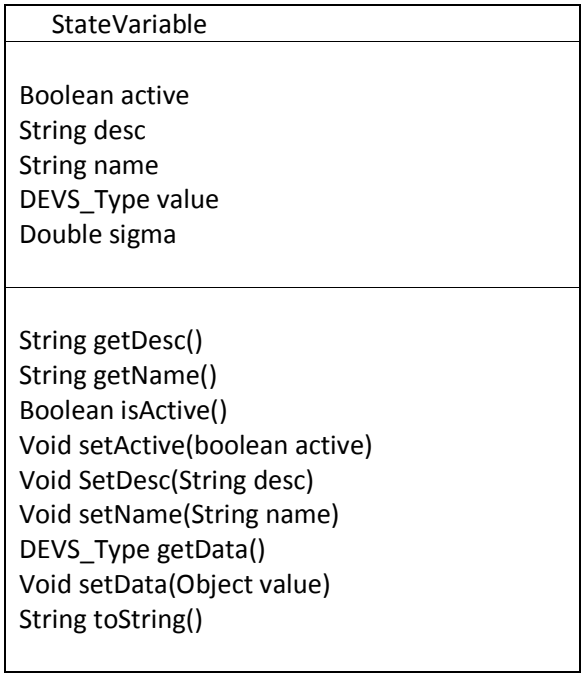


Figure3.2: ***The Class Diagram for CDEVS Model***


AtomicModel
State state
Void addStateVariable(DEVS_Type, String name,String desc)
DEVS_Type getStateVariableData(String name)
Void setStateVariableData(String name, Object val)
abstract void deltaExt(double e)
abstract void deltaInt()
abstract void lambda()
abstract double ta()
String toString(int level)


State
ArrayList<StateVariable> vars
Void addStateVariable(StateVariable var)
DEVS_Type getStateVariableData(String name)
Void setStateVariableData(String name, Object value)
String toString()


CoupledModel
ArrayList<Pair> EIC
ArrayList<Pair> EOC
ArrayList<Pair> IC
ArrayList<Model> subModel
Void addEIC(Input port1, Input port2)
Void addIC(Output port1, Input port2)
Void addSubModel(Model m)
ArrayList<Input> getLinkedInput(Port port)
ArrayList<Port>) getLinkedInternalPort (Port port)
ArrayList<Output> getLinkedOutput (Port port)
String toString(int level)
Abstract Model Select(ArrayList<Model> models)

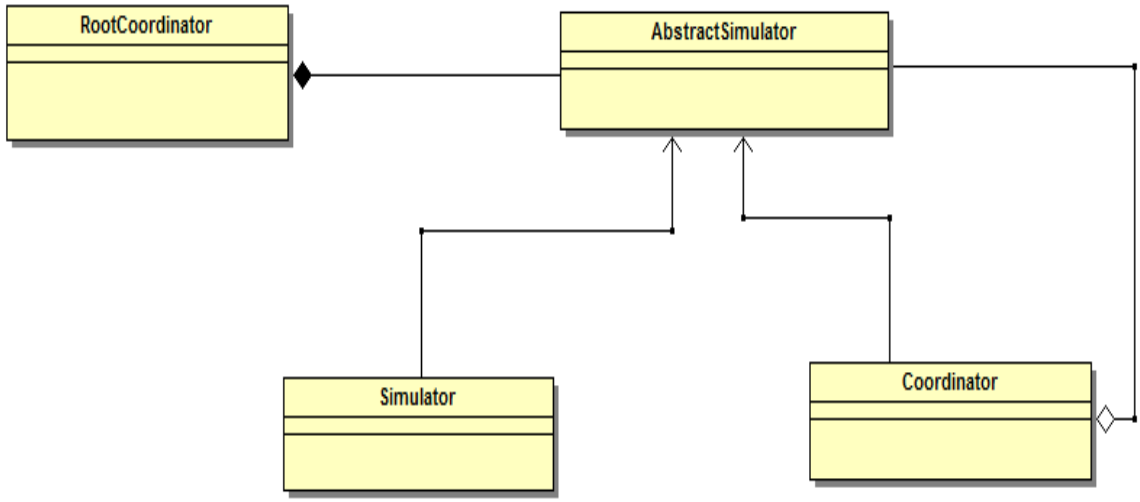

StateVariable
Boolean active
String desc
String name
DEVS_Type value
Double sigma
String getDesc()
String getName()
Boolean isActive()
Void setActive(boolean active)
Void SetDesc(String desc)
Void setName(String name)
DEVS_Type getData()
Void setData(Object value)
String toString()

Port
String desc Model model String name Object value
Boolean equals(Port p) String getDesc() Model getModel() String getName() Object getPortData() Void setDesc(String desc) Void setModel(Model inModel) Void setName(String name) Void setPortData(Object value)

Model
String desc String name ArrayList <Input> X ArrayList<Output> Y AbstractSimulator sim
Void addInputPortStructure(DEVS_Type type,String name,String desc) Void addOutPortStructure(DEVS_Type type, String name, String desc) Input getInputPortStructure(String name) Output getOutputPortStructure(String name) Void setInputPortData (String name, Object value) Voiod setOutPortData(String name, Object value) Object getInputPortData(String name) Object getOutputPortData(String name) AbstractSimulator getSimulator() String getDesc() Void setDesc(String desc) String getName() Void setName(String name) Abstract String toString(int level)

Remarks:

- Model: Both atomic and coupled model are sub-classes of the model class. The model class has some logic and attributes common to both coupled and atomic models.
- Input and Output are sub-class of Port, there are ports on which values can be received and sent. Each model is associated with ports.
- AtomicModel is composed of state which can change according to its stateVariable from an old state to a new state.



The Class Diagram for CDEVS Simulator 1

Figure 3.3: ***The Class Diagram for CDEVS Simulator***

The details of attributes and methods are given as below :

AbstractSimulator
Double e Double tl Double tn AbstractSimulator parent

Abstract Model getModel() Abstract Simulator getParent() Void setParent(AbstractSimulator s) Double getTL() Void setTL(double tl) Double getTN() Void setTN(double tn) Abstract void init(double t) Abstract void internalTransition(S_Message msg, double t) Abstract void externalTransition(X_Message msg, double t) Abstract void transfer (X_Message msg, double t) Void handleMessage(Message msg)

Coordinator
CoupledModel model ArrayList<AbstractSimulator > subjects
Void addSubject(AbstractSimulator Sim) Model getModel() Void init(double t) Void internalTransition(S_Message msg, double t) Void externalTransition(X_Message msg, double t) Void transfert(Y_BagMessage msg, double t) Void updateTn()

RootCoordinator
AbstractSimulator Sim
Void init(double t) Void run() Void run(int limit)

Simulator
AtomicModel model
Model getModel() Void init(double t) Void internalTransition(S_Message msg, double t) Void externalTransition(X_Message msg, double t) Void transfert(Y_Message msg, double t)

Hierarchical models are coupled models with components that may be Atomic or Coupled models. As Figure 3.3 illustrates, the AbstractSimulator is a super class which drives the Atomic models. The Coordinator drives the coupled models and the Root Coordinator controls the whole simulation by communicating with models through the AbstractSimulator based on clock.

3.2.1.2 UML Sequence Diagram

The figure bellow shows the Sequence Diagram of the CDEVS Simulator. It represents its dynamic view.

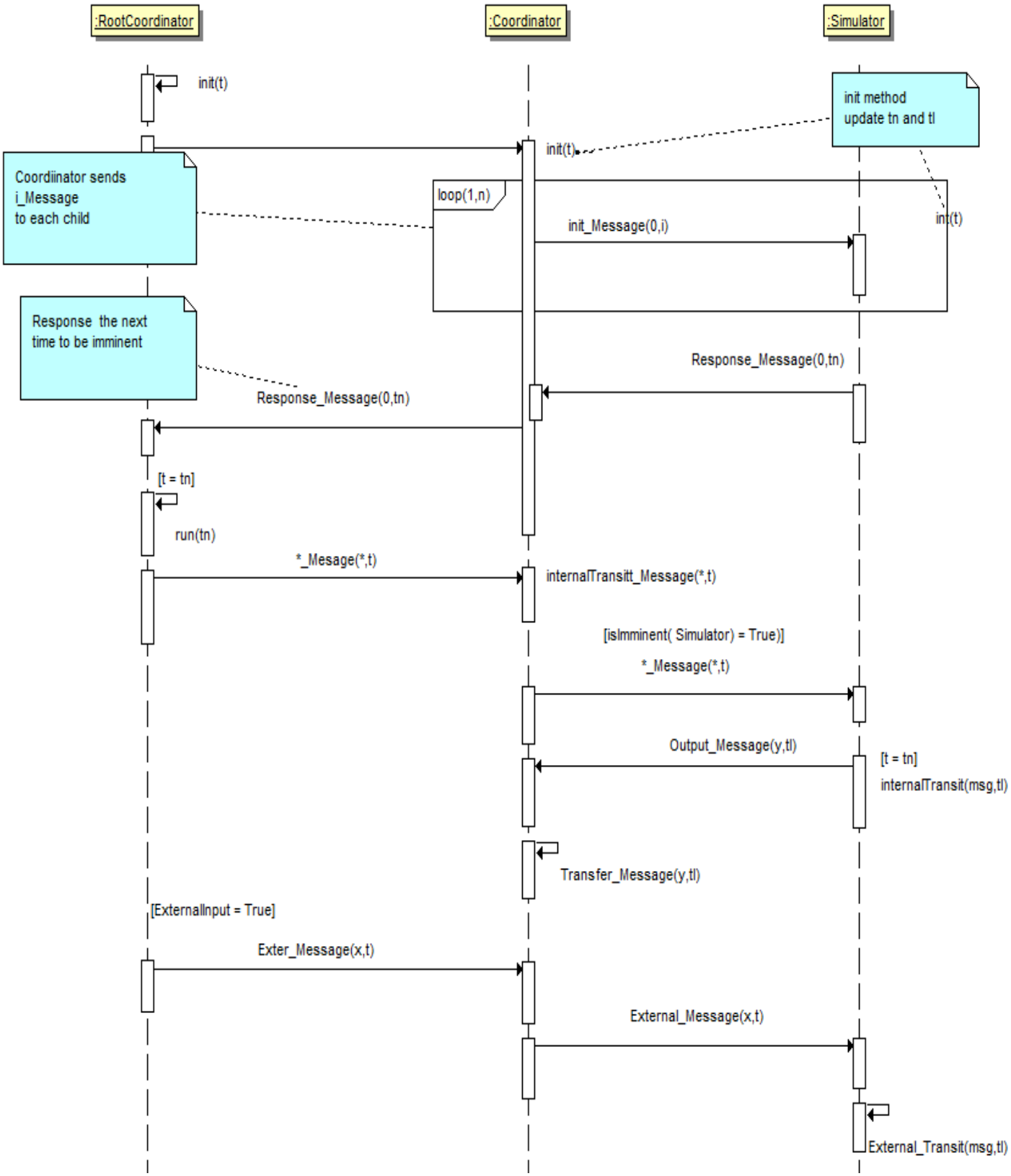


Figure 3.4: ***The Simulator Sequence Diagram for CDEVS***

3.2.2 Comparison of approaches

Many tools have been implemented based on DEVS theory and its extensions. The following list shows the different descriptions of some of the tools.

Comparison of DEVS tools 1

Name	*Description*	*Stand-alone Simulation*	*Distributed/ Parallel execution*
ADEVS	ADEVS is a C++ library for developing discrete event simulations based on the Parallel DEVS and DSDEVS formalisms. It includes support for standard, sequential simulation and conservative, parallel simulation on shared memory machines with POSIX threads. Developed by Jim Nutaro (University of Arizona, U.S.A.).	✓	
CD++	CD++ is a general toolkit written in C++, which allows the definition of DEVS and Cell-DEVS models. DEVS coupled models and Cell-DEVS models can be defined using a high level specification language. Different versions include Real-Time, Parallel and centralized simulators. Developed by Gabriel Wainer and his students (Carleton University, Canada; Universidad de Buenos Aires, Argentina).		✓
DEVS/C++	DEVS-C++, based on the parallel DEVS formalism [4], is a modular hierarchical discrete event simulation environment implemented in the object-oriented C++ language. Developed by Hyup J. Cho and Young K. Cho (University of Arizona, U.S.A.).		✓
DEVSJAVA	Modeling and Simulation environment for developing DEVS-based models. The software is written in Java and supports parallel execution on a uni-processor. It supports higher-level, application specific modeling. Models in DEVSJAVA can also be readily mapped to DEVS/HLA and DEVS/CORBA for distributed execution in combined logical/real-time settings. Developed by Hessam Sarjoughian (Arizona State University, U.S.A.) and Bernard Zeigler (University of Arizona, U.S.A.).		✓
	DEVSim++ is an environment for Object-		

DEVSim++	Oriented Modeling of Discrete Event Systems. Developed by Tag Gon Kim (KAIST, Korea).		✓
SmallDEVS	SmallDEVS is an experimental DEVS-based simulation package for Squeak Smalltalk. It allows for class-based, as well as prototype-based object-oriented model construction. Its reflective features make inspection and interactive manipulation with models and running simulations possible. Interactive modeling and simulation is supported by a GUI. Developed by Vladimir Janousek and Elod Kironsky (Brno University of Technlogy, Czech Republic).	✓	
DEVS/HLA	DEVS/HLA is based on the high-level architecture (HLA). It was used to demonstrate how an HLA-compliant DEVS environment could improve the performance of large-scale distributed modeling and simulation.		✓
PowerDEVS	PowerDEVS is a general purpose software tool for DEVS modeling and simulation oriented to the simulation of hybrid systems	✓	

Figure 3.5: ***Comparison of DEVS tools***

3.2.3 Problems with existing implementations

The DEVS tools have problems [4] to provide the bellow points:

- Interoperability and reuse
- Engineering-based approach (different for varied types of M&S life cycles)
- Facilities for automated tasks
- High performance/distributed simulation
- Hybrid systems definition

Chapter 4. Formal Methods

> *Professional Engineers are expected to use discipline, science, and mathematics to assure that their products are reliable and robust. We should expect no less of anyone who produces programs professionally.*
>
> *—David Lorge Parnas.*

This chapter introduces formal methods in general, reveals the benefits of formal methods and the survey of tools and methods.

4.1 Introduction to formal methods

Applied to computer systems development, formal method provides mathematically based techniques that describe system properties [5]. As such, they present a framework for systematically specifying, developing, and verifying systems.

A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides the means of precisely defining notions like consistency and completeness and, more relevantly, specification, implementation, and correctness. It is proving that a system realizes a specification and has been implemented correctly according to its properties without necessarily running it to determine its behavior.

A formal method also addresses a number of pragmatics considerations: who uses it, what it is used for, when it is used and how it is used? Most commonly, system designers use formal method to specify a system's desired behavioral and structural properties.

However, at any stage of system development anyone involved can make use of formal method. They can be used in the initial statement of a customer's requirement, through system, implementation, testing, debugging, maintenance, verification, and evaluation.

Formal method are used to reveal ambiguity, incompleteness, and consistency in a system.

When used early in the system development process , they can reveal design flaws that otherwise might be discovered only during costly testing and debugging phases. When

used latter, they can help determine the correctness of a system implementation and the equivalence of different implementations.

One tangible product of applying a formal method is a formal specification. A specification serves as a contract, a valuable piece of documentation, and means of communication among a client, a specifier, and an implementer.

Because of their mathematical basis, formal specifications are more precise and usually more concise than informal ones.

Since a formal method is a method and not just a computer program or language, it may or may not have tool support. If the syntax of a formal method's specification language is made explicit, providing standard syntax analysis tools for formal specifications would be appropriate.

If the language's semantics are sufficiently restricted, varying degrees of semantic analysis can be performed with machine aids as well. Thus, formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation.

4.2 Benefits of formal methods

Formal methods consist of writing formal descriptions, analyzing those descriptions and in some cases producing new descriptions —for example refinements— from them. In what way is this a useful [6] activity?

First, experience shows that the very act of writing the formal description is of benefit: it forces the writer to ask all sorts of questions that would otherwise be postponed until coding. Of course, that's no help if the problem is so simple that one can write the code straight away, but in the vast majority of systems the code is far too big and detailed to be a useful description of the system for any human purpose. 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.

4.3 Survey of tools and methods

The following is the list of some Formal methods tools and the methods (Input languages) used.

Tool	*Input Language (Tool's own language*	*Supported Techniques*	*Platforms*	*Availability*
ProofBuilder	First-order or propositional logic	Proof Assitasnt/Interactive Theorem Proving	Anything that supports Java	Free
SMV	finite state machine	Model Checking	Linux, Sparc/Solaris	Free
PVS	typed higher order logic	Proof Checking, Model Checking	Linux, Sparc/Solaris	Free
HOL	Higher-order logic	Proof Checking	Linux, Alpha, Sparc/Solaris	Free
Murphi	guarded command language	Model Checking, Symmetry	Linux, Sparc/Solaris, SGI Indy, HP-UX	Free
SPIN	Promela	Model Checking, Partial Orders	Windows NT, Windows 95, Linux, Sparc/Solaris	Free
VIS	Verilog	Model Checking, Synthesis	Windows 95, Linux, Alpha, Sparc/Solaris, DEC MIPS, HP Snake	Free
Reactis Tester	Simulink/Stateflow)	automatic test-suite generation via state exploration	Windows (all varieties), Linux	Commercial
AtelierB	B Method	Model checking , Proof checking	Windows /Linux	Free
LTSA	FSP	Model Checking, Partial Orders, Synthesis from MSCs	Windows (all varieties), Linux, Macintosh	Free
Alloy	Declarative relational logic	Satisfiability Checking	Windows (all varieties), Linux, Macintosh, Sparc/Solaris	Free

Formal Methods tools and methods 1

Figure 4.1: ***Formal Methods tools and methods***

Chapter 5. Simstudio and Formal Methods

In this chapter we discuss the meta-models implementation of the Simstudio package and map it to a formal specification leading into relatively simple and independent component, which allowed for straightforward unit testing.

5.1 Improvements on Simstudio

- Package and class view of the Implementation

The CDEVS Simulator Algorithms is implemented in Java. This implementation was prepared as a Java package (SIMSTUDIO_1_1) by Aminu [3] in his work. The Simulator consists of 6 packages as shown below in Fig 5.1

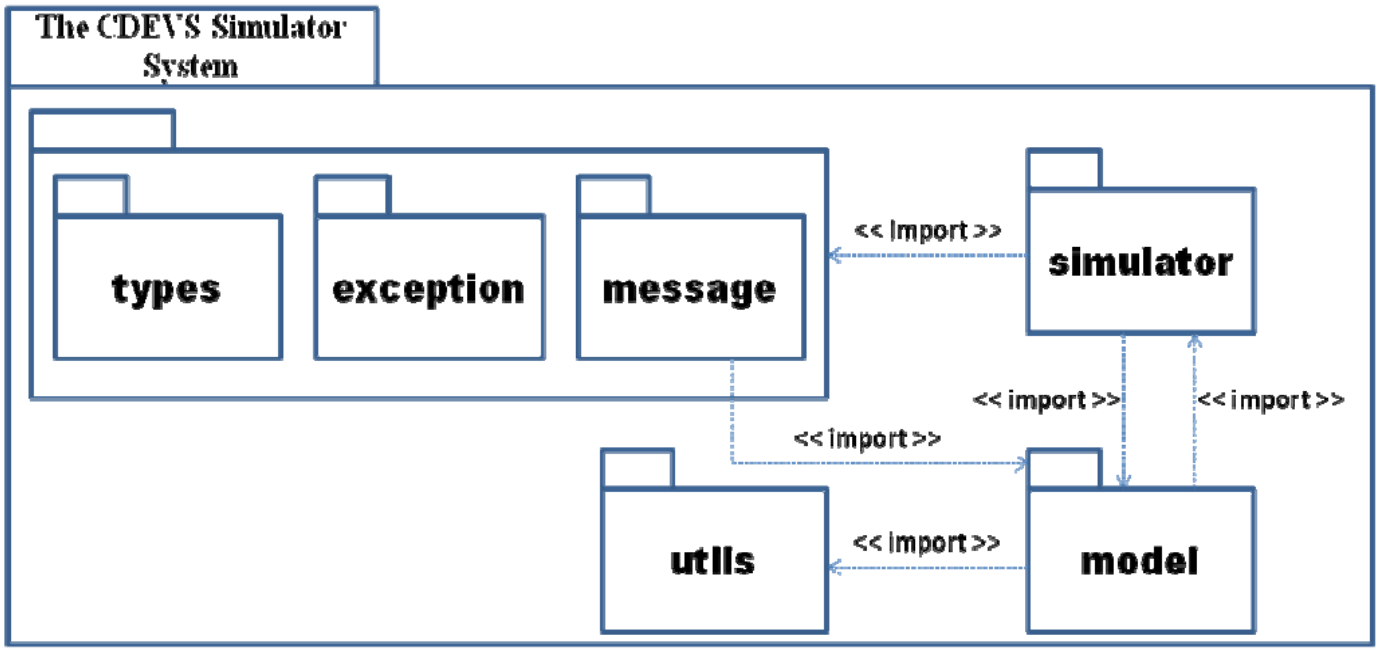
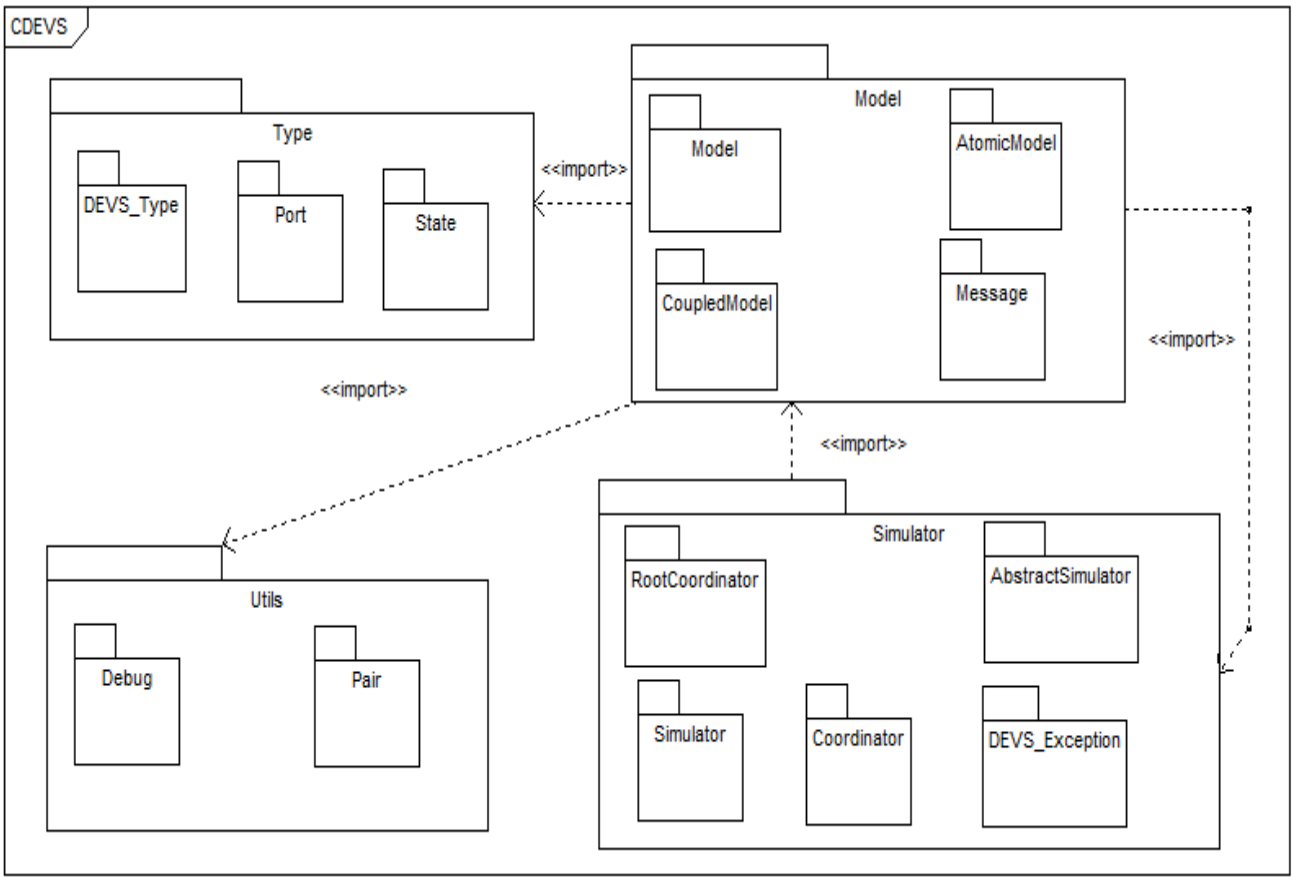


Figure 5.1: ***CDEVS UML Package Diagram***



CDEVS Package Diagram of Implementation 1

Figure 5.2: ***CDEVS Package Diagram of Implementation.***

We redesigned it and made it more efficient as illustrated in the Figure 5.2.

Remarks :

- ❖ Model Package : The model package contains descriptions of a model and a model can either be coupled or atomic model.
  - o AtomicModel: It has attributes and methods about the structure of the atomic model.
  - o CoupledModel: It has attributes and methods about the structure of the coupled model.
  - o Message: It defines common specifications of the messages for the communication between models.

- Model: It's an abstract class and contains attributes about the Simulator that's driving the model, the name, the description and the identifier of the model. The Atomic and the Coupled Model extend the class Model.

❖ Simulator Package: This package contains components to run the Simulators and the Coordinators in the system.
- AbstractSimulator: This is an abstract Class that contains the basis common to both Simulator and Coordinator classes.
- Coordinator: It drives a coupled model. A coordinator is a specific simulator.
- Simulator: It drives an atomic model.
- RootCoordinator: It drives a Simulator and not a model.
- DEVS_Exception: This Class is a general exception Class. All the DEVS exception such as ConceptionErrorException, ProgrammingException and SynchroException derive from this one. It can be used whenever the exception Classes that have been defined cannot be used.

❖ Type Package: It contains Classes that are used to define DEVS_Type, Port and State.

❖ Untils Package: It contains Classes such as Debug and Pair that are used for debugging purpose during the simulation.

➢ Debugging: FinBugs applied to Simstudio

FindBugs [11] is a bug pattern detector for Java. FindBugs uses a series of ad-hoc techniques designed to balance precision, efficiency, and usability. One of the main techniques FindBugs uses is to syntactically match source code to known suspicious programming practice. FindBugs is an open source static analysis tool. Already in use at Google, FindBugs needs to also provide hooks into other bug tracking and web source viewers. In the table bellow we describe the bug patterns, the number of bugs in the classes where the bugs are found. The bugs found are fixed in the Simstudio package.

Description	Class	Number
Equal Objects Must Have Equal Hashcodes	Port	1
Bad Covariant Definition of Equals	Port	1
Method concatenates string using "+" in a loop	AtomicModel	2
Method concatenates string using "+" in a loop	CoupledModel	2
Method concatenates string using "+" in a loop	State	1
Method invokes inefficient new string(String) constructor	AtomicModel	1

Method invokes inefficient new string(String) constructor	CoupledModel	1
Method invokes inefficient new string(String) constructor	StateVariable	1
Method invokes inefficient new string(String) constructor	DEVS_Char	1
Method invokes inefficient new string(String) constructor	DEVS_Integer	1
Method invokes inefficient new string(String) constructor	DEVS_Interval	1
Method invokes inefficient new string(String) constructor	DEVS_Real	1
Method invokes inefficient new string(String) constructor	DEVS_RealInterval	2
Possible null Pointer Dereference	Coordinator	1
Test for floating point equality	DEVS_RealInterval	1

5.2. Towards integration of formal analysis with Simstudio

After many investigations we finally find an appropriate formal method to integrate with Simstudio and the formal specification is written and tested as we describe thereafter.

5.2.1 Formal Method: The B Method

B is a formal method for specifying, designing, and coding software systems [7].The specification of a program takes the form of an abstract machine, which roughly corresponds to the modules in many programming languages such as Ada and Haskell. Abstract machines can be composed of other abstract machines. Abstract machines can be refined into other abstract machines. Ultimately, the chain of refinement ends at a concrete machine, the implementation, which is still written in more or less the same notation as the original abstract machine but which is mechanically translatable into executable code. Implementations can be self-sufficient or they can depend on the services of other abstract machines (which are separately refined into implementations).

The B notation is based on set theory, the language of generalized substitutions and first order logic. Specifications are called B components. A B component is a B abstract machine (BAM), a refinement component or an implementation component of a BAM. A BAM (Figure 5.3(a)) consists of a set of variables, invariance properties relating to those variables and operations. The variable values are only modifiable by operations which must preserve the invariant (Figure 5.3(b)).

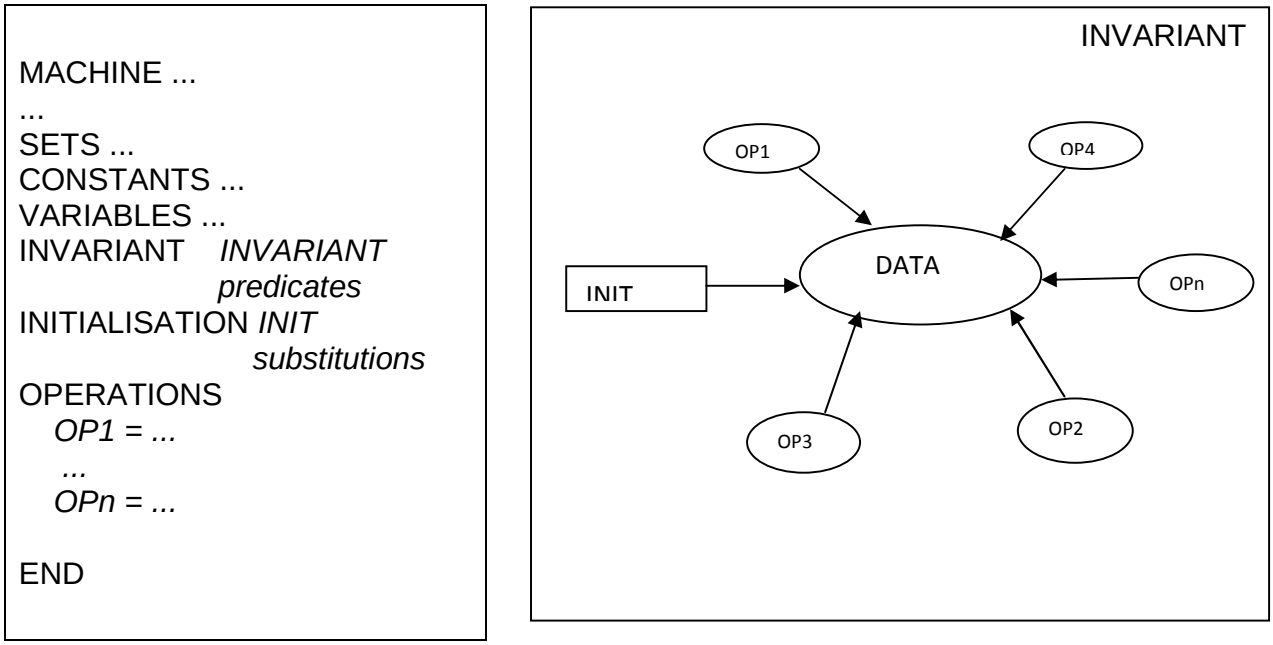


Figure 5.3: ***Intuitive view of a BAM***

At every stage of the specification, proof obligations ensure that operations preserve the system invariant. There are also proof obligations for the refinement correctness to be discharged when a refinement is postulated between two B components. It has to be noticed that B was designed to be automated easily. The generation of proof obligations (of the invariant preservation and the refinement correctness) obeys the simple rules that can be easily implemented in a piece of software. Furthermore, support tools like AtelierB and B-Toolkit provide utilities to discharge automatically and interactively the generated proof obligations. Analyzing the non-discharged proof obligations with the B support tools is an efficient and practical way to detect errors encountered during the specification development. The goal of B is to obtain a proved model.

5.2.2 UML to B

The Unified Modeling Language (UML) has become a de-facto standard notation for describing analysis and design models of object-oriented software systems.

UML is used to implement the Meta model of our Simstudio (UML class diagram, UML sequence diagram).

However, the fact that UML lacks a precise semantics is a serious drawback of UML-based techniques.

B is a formal software development method that covers software process from the abstract specification to the executable implementation. Thus an appropriate combination of object-oriented techniques and formal methods can give rise a practical and rigorous software development. For this objective, a practical approach is to derive B specifications from UML specification.

This UML-B integration has the following advantages:

(i) The B specification is mathematically manipulable enabling reasoning and proof to be performed

(ii) The B toolkit is available for type analysis, proof assistance and animation

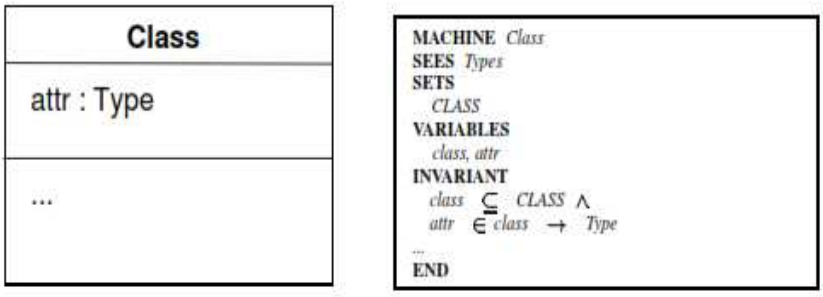
(iii) The translation demonstrates the semantics of the UML version.

From the informal description of requirements, we successively build the object models with different degrees of abstraction. These models cover from conceptual models through logical design models to the implementation models of the software. This also means that the developed models are successively refined. We verify the consistency of each object model by analyzing the derived B specification. We verify the conformance between object models by analyzing the refinement dependency among them that is formally expressed in B.

5.2.3 Formal specification of Simstudio

E.Meyer [9] and Nguyen have proposed a set of precise and implementable rules for modeling in B the concepts of the class diagrams. Given a class in Figure 5.4(a) a BAM is created in Figure 5.4(b)

All the class is formalized as a BAM 1



(a) (b)

Figure 5.4: ***All the class is formalized as a BAM***

By the followed manner we derive all the class from the implementation of Simstudio Meta model in Chapter 3, Figure3.2: *The class Diagram for CDEVS Model* and *Figure3.3: The class Diagram for CDEVS Simulator* to the formal specification in B method.

```
MACHINE Model
SEES Types
EXTENDS
        Port
SETS
        ModelSets
VARIABLES
        ModelInstance,desc_,id_,ID_
        name,sim_,X,Y,object,devs_type
INVARIANT
        ModelInstance <: ModelSets &
        desc_ : ModelInstance --> String &
        id_ : ModelInstance --> Integer &
        ID_ : ModelInstance --> Integer &
        level : ModelInstance --> Integer &
        name : ModelInstance --> String &
        sim_ : ModelInstance --> AbstractSimulator
        X : ModelInstance --> Input
        Y : ModelInstance --> Output
        devs_type : ModelInstance --> DEVS_Type
        object : ModelInstance --> Object
INITIALISATION
        ModelInstance := {} ||
        ID_ := 0 ||
        sim_ := {} ||
        x := {} ||
        y := {}
OPERATION
Return <-- modelcreate(name,desc) =
        PRE     name : name_ & desc : desc_ &
                ModelInstances /= ModelSets
        THEN
                ANY new
                WHERE
                new : ModelSets - ModelInstances
                THEN
```

```
            ModelInstances := ModelInstances \/ {new}
            || name_(new) := name
            || desc_(new) := desc
            || id_ (new):= ID + 1
            || Return := new
    END ;
    addInputPortStructure(type,name,desc) =
    PRE
            type : Types & name : name & desc : desc_
    THEN
            add(type,name,desc)
    END;
    addOutputPortStructure(type,name,desc) =
    PRE
            type : Type & name : name & desc : desc_
    THEN
            add(type,name,desc)
    END;
    desc <-- getDesc()
    END;
    setDesc(desc) =
    PRE
            desc : desc_
    THEN
            desc_ := desc
    END;
    id < -- getId()
    END;
    InputPort <--- getInputPortData(name) =
    PRE
            name : name
            VAR i IN
            WHILE i : X DO
            IF i.getName = name
            THEN
                    InputPort := i.getPortData()
            ELSE
                    InputPort := {}
            END
    END;
    OutputPort <--- getOutputPortData(name) =
    PRE
            name : name
            VAR j IN
            WHILE j : Y DO
            IF j.getName = name
```

```
            THEN
                OutputPort := j.getPortData()
            ELSE
                OutputPort := {}
            END;
    name <-- getName()
    END;
    setName(name) =
    PRE
            name : name
    THEN
            name := name
    END;
    sim <-- getSimulator()
    END;
    input <-- getInputPortStructure(name) =
    PRE
            name : name
            VAR i IN
            WHILE i : X DO
            IF i.getName = name
            THEN
                input := i
            ELSE
                input := {}
            END
    END;
    Output <--- getOutputPortStructure(name) =
    PRE
            name : name
            VAR j IN
            WHILE j : Y DO
            IF j.getName = name
            THEN
                Output := j
            ELSE
                Output := {}
            END;
    setInputPortData(name,value) =
    PRE
            name : name & value : object
            VAR i IN
            WHILE i : X Do
            IF i.getName = name
            THEN
            i.setPortData(value)
```

```
        END;
        setOutputPortData(name,value) =
        PRE
                name : name & value : object
                VAR j IN
                WHILE j : Y DO
                IF j.getName = name
                THEN
                        i.setPortData(value)
                        IF sim_.getParent() := {}
                        THEN
                                sim_.getParent().handleMessage(Y.Message(j,sim_.getTL)
                        END
                END
        END;
        theString <-- toString() =
                theString := String(0)
        END;
        theString <-- toString(level) =
                PRE level : level
        END;
END
```

BAM Class Model 1

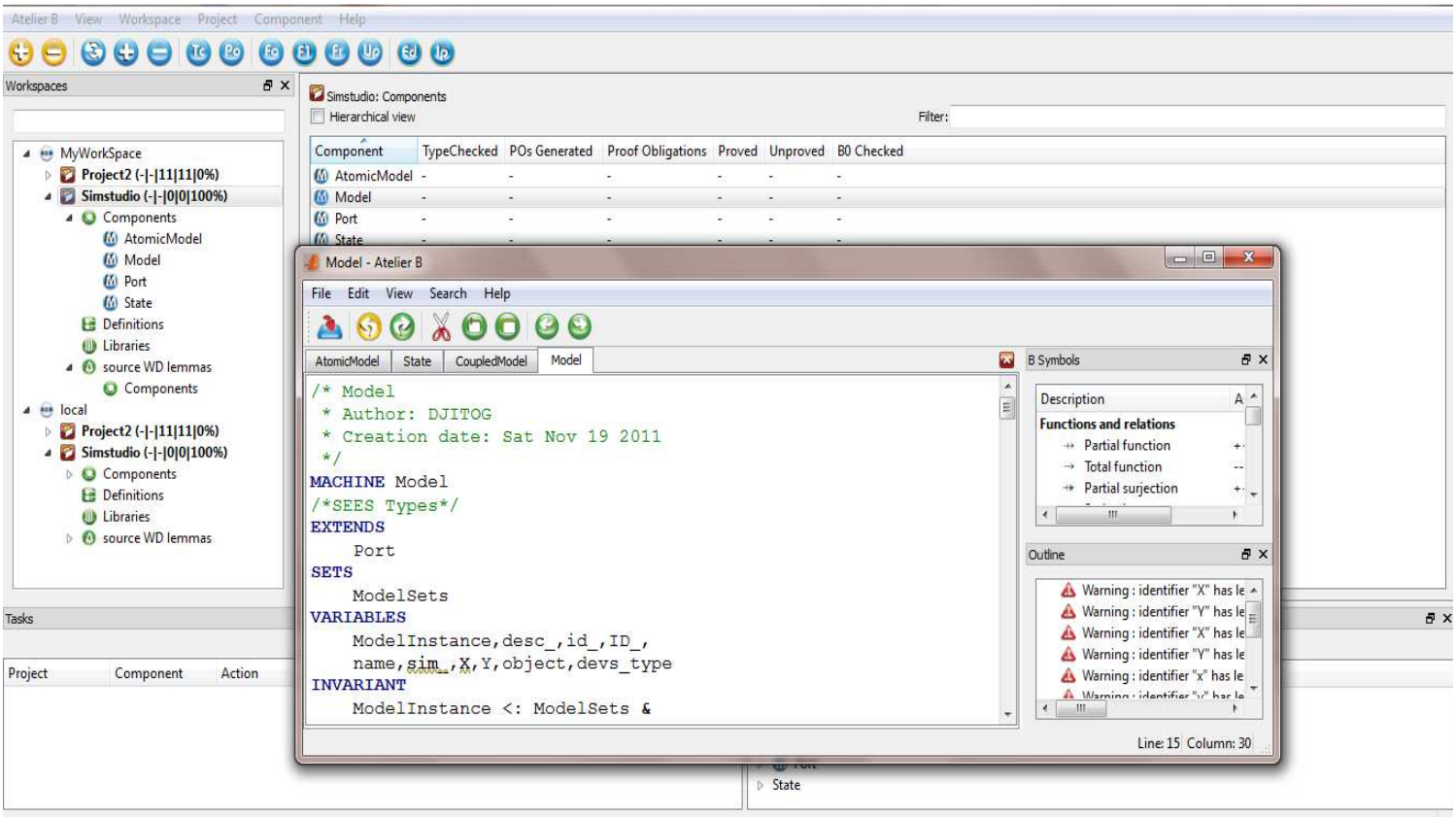
Figure 5.5: ***BAM Class Model***

The complete formal specification of Simstudio is in the package attached to this thesis.

5.3 Use of formal tools with Simstudio: The Atelier B

Atelier B [10] is a set of tools allowing developing applications using the B method. It assists developers in the formalization of their applications, performing automatically on specifications and their refinements, syntax analysis, type checking, generation and demonstration of proof obligations.

Based on the implementation of the Simstudio Meta model each class is specified as a B component. The screenshot bellow shows the formal specification of Simstudio in B Method as the input language of the tool Atelier B.

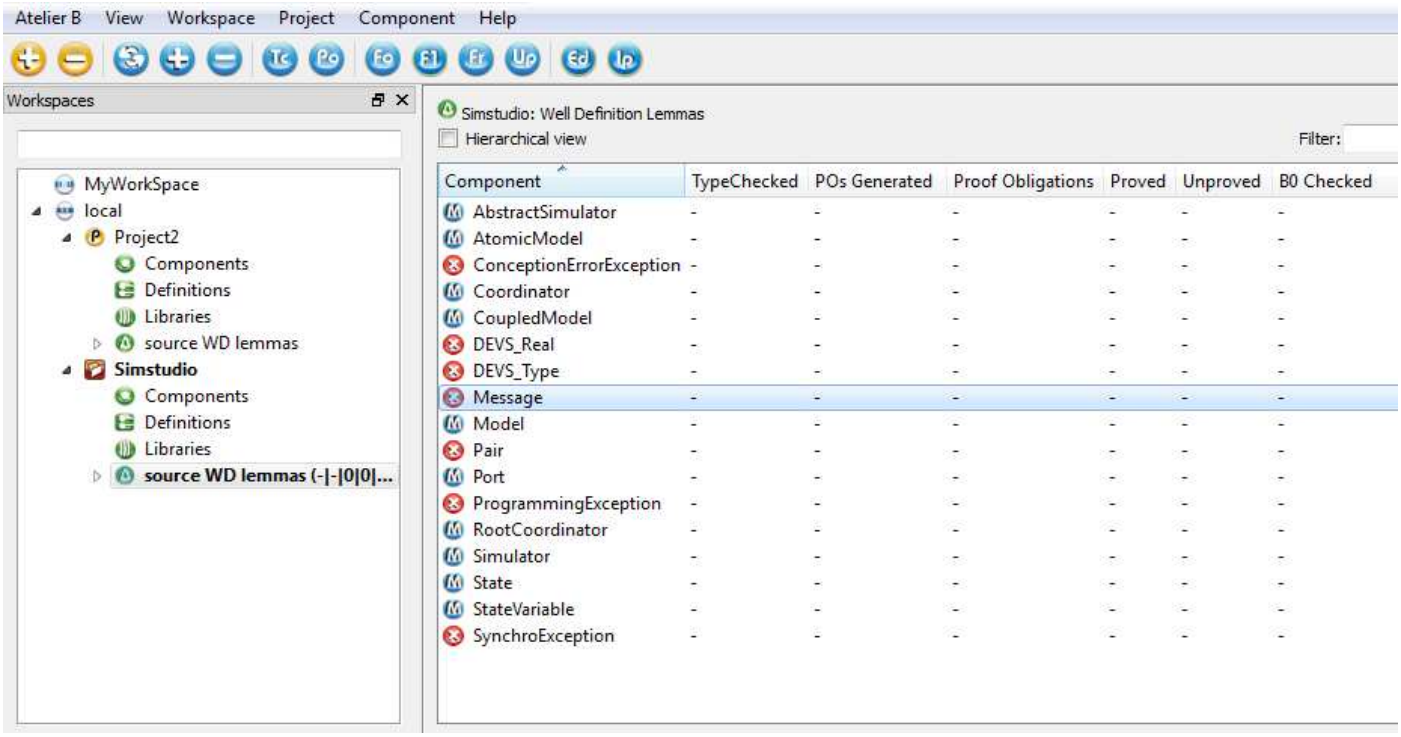


Integration of Atelier B with Simstudio 1

Figure 5.6: ***Integration of Atelier B with Simstudio***

5.4 Results and Discussions

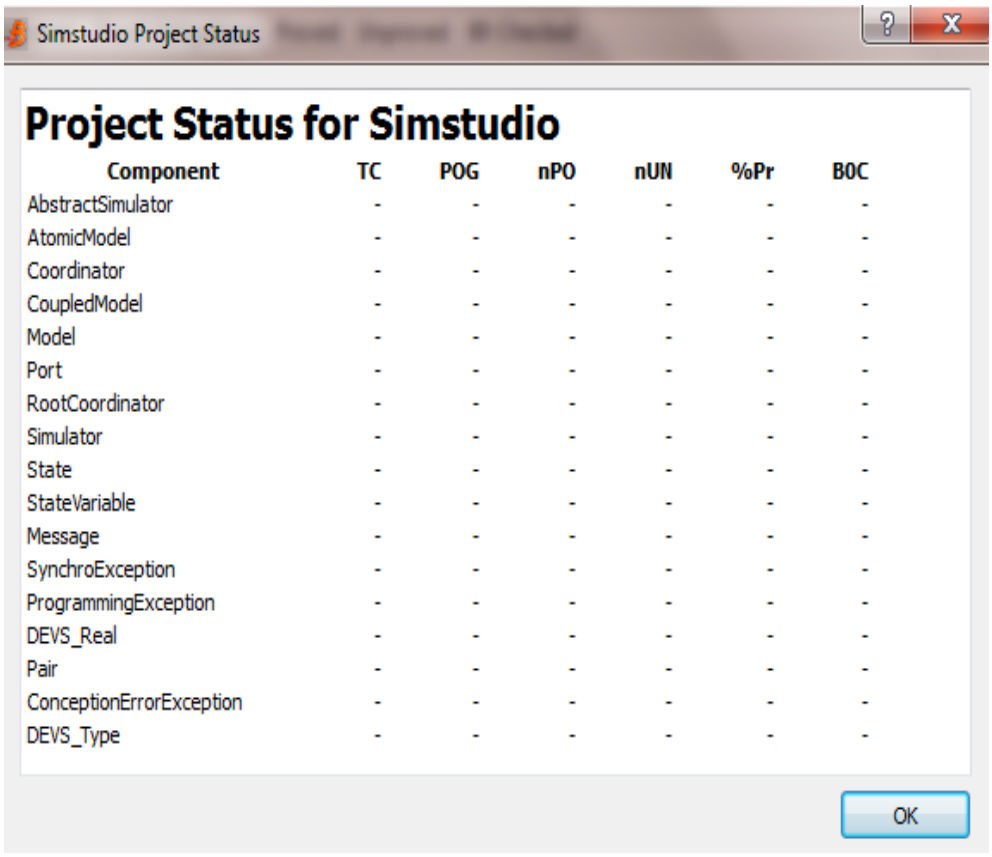
- The result of the formal specification for all the components of Simstudio written above is displayed as followed :



Simstudio Typeched and Proof Obligations 1

Figure 5.7: ***Simstudio Typechecked and Proof Obligations.***

- The figure bellow shows the status :



Simstudio Status 1

Figure 5.8: ***Simstudio Status.***

Discussions: Description of the Status

- ✓ TC : Syntactical analysis,
- ✓ POG : Proof Obligations(PO) generated,
- ✓ nPO : The number of PO,
- ✓ nUn : The number of PO remains to prove,
- ✓ % Pr : The percentage of PO proved,
- ✓ BOC : Syntactical analysis before the translation of the component to a programming language like C, ADA, etc ...

The status result shows that the output for TC is not completed. It is the raison why the Proof Obligation is not generated since the type checking is combined with the syntax analysis of B components and it ensures that the sources for the selected machines comply with the B language syntax.The type check of a component is automatically applied to all the components “required” by the current component, through the following links SEES, USES, INCLUDES, IMPORT, EXTENDS and REFINES. It follows that In the Figure 5.7 the following components Message, SynchroException, ProgrammingException, DEVS_Real, Pair, ConceptionErrorException and DEVS_Type are not present in the project while they are in red color (error component) and we notice that these components are not implemented in the UML implementation of the Simstudio (meta-model) in Figure3.2 and Figure3.3.

Chapter6. Conclusions

We started from the review of the Meta model of the Simstudio package to get the formal analysis of the simulation protocol. We followed the set of rules proposed by E.Meyer and Nguyen to map the UML class diagram to a formal analysis of the simulation protocol. After written a formal specification in B Method as a formal method we implemented it with its powerful tool Atelier B to perform the properties of Type checking and Proof Obligation for the correctness. The result showed the type checking is not completed.

Challenges:

During the formal analysis, the choice of the formal method and the tool to use for the formal specification of the simulation protocol was a serious challenge. So many tools have been testing before getting the Atelier B tool and the B method to support a specification from a UML to B. Again learning a formal specification language is difficult.

Future work:

We propose to achieve the generation of the Proof Obligation so that the property of the correctness should be completed. A question is to know whether the meta model of the Simstudio implemented in the class diagram for CDEVS Model and the class diagram for CDEVS Simulator will be redrawn and make it as one class diagram to avoid the type check of a required component linked to other components, through the following closes SEES, USES, INCLUDES, IMPORT, EXTENDS and REFINES used by **Atelier B**?

This can end up with a Simstudio package formally proved rather than the current one.

REFERENCES:

[1]. Gabriel A. Wainer, Discrete – Event Modeling and Simulation: A practioner's Approach, 2009.

[2]. Zeigler.B.P Theory of modeling and simulation. New York: Wiley Interscience. 1976

[3]. Aminu.M, Approach to a Simulation virtual Machine: Object oriented implementation of DEVS and PDEVS. Master's thesis, African University of Science and Technology, Nigeria, December 2009.

[4]. SIMAUD, DEVS Standardization, Boston MA, April 5, 2011.

[5]. Jeannette M.Whing, A specifier's Introduction to Formal Methods, September 1990.

[6]. Anthony Hall, "Realizing the Benefits of Formal Methods", Independent consultant, UK.

[7]. Antti-Juhani Kaijanaho, "The formal method known as B and a sketch for its implementation", Master's Thesis in Information Technology (Software Engineering) 20th December 2002.

[8]. Hung LEDANG and Jeanine SOUQUIERES, "Formalizing UML Behavioral Diagrams with B", Universite de Nancy 2.

[9]. E.Meyer, Developpements formels par objects: utilization conjointe de B et d'UML. Phd thesis, LORIA – Universite Nancy 2, Nancy(F), mars 2001.

[10]. Clearsy system engineering, "Atelier B user Manual, version 4.0", Parc de la duranne - 320 av. Archimede.

[11]. David Hovemeyer and William Pugh, "Finding Bugs is easy", University of Maryland.