STRATEGIES TO IMPLEMENT PARALLEL DISCRETE EVENT SYSTEM SPECIFICATION ALGORITHMS

A Thesis Presented to the Department of

Computer Science,

African University of Science and Technology,

Abuja, Nigeria.

In Partial Fulfillment of the Requirements

For The Degree of

MASTER OF SCIENCE

By

ADEKUNLE Onaopepo Husamat

December, 2014.

STRATEGIES TO IMPLEMENT PARALLEL DISCRETE EVENT SYSTEM SPECIFICATION ALGORITHMS

By

ADEKUNLE Onaopepo Husamat

A THESIS APPROVED BY THE DEPARTMENT OF COMPUTER SCIENCE

RECOMMENDED: ………………………………………………………..

Supervisor, Professor Mamadou Kaba Traore

………………………………………………………..

Head, Department of Computer Science

APPROVED: ……………………………………………………………..

Chief Academic Officer

………………………………………………………………

Date

ABSTRACT

Discrete Event System Specification (DEVS) formalism is a Modeling and Simulation (M&S) framework that provides a means of specifying systems. It separates a model from its simulator. The former describes the structure and the behavior of a system, while the later generates the trajectories of these descriptions. P-DEVS (Parallel DEVS) is the version of DEVS that allows one to express at the modelling level, the parallelism present in a system. Though the algorithm is well defined, its implementation remains challenging. This thesis presents implementations of the Parallel DEVS simulation algorithm developed as simulation engines. This work proposes the classification of the algorithms that exists based on implementation approach. Our goals include the implementation of the algorithms, benchmarking and analysis. The implementations are classified as Object Oriented based approach (OOP) that uses OOP paradigm to be realized, process based approach that uses threads of execution for realization and hybrid simulators that uses a mix of the paradigms to be realized.

ACKNOWLEDGEMENT

I wish to express my profound gratitude to Professor Mamadou Kaba Traore for supervising, supporting and guiding me throughout this thesis work. His advices were the beacon that saw me through the challenges encountered in the course of the work.

I also want to say a big thanks to all my colleagues, especially, Adelani David, Adebajo Fisayomi, Ogbonna Amanze, Sam Manthalu, Gondwe Walusungu, Tiako Fani Michelle for all the support and encouragement and making sure my stay at the African University of Science and Technology a very memorable one.

I am also very grateful to my parents and family for the immense love and support and provision of the means to be a part of the African University of Science and Technology community.

Finally, I appreciate the Almighty God for graciously granting me life, wisdom and strength to see this Masters program through. He alone deserves all of the praises, glory and adoration.

DEDICATION

To the Almighty God for keeping me hale and hearty throughout this wonderful experience. Also to my wonderful parents for their immeasurable support, love and prayers.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

1 INTRODUCTION

Complex Information Technology (IT) based business, engineering, and military systems are at the root of this century's global challenges of economy, climate and energy. We are used to building such systems directly in the real world and letting use and Mother Nature tell us how good they are. However, it is getting increasingly dangerous, costly, risky, or even unethical to do so. Building a model of the system and testing within a virtual space is more and more the only workable alternative where by "virtual" we include a wide range of representations of the eventual fielded reality including models wholly within a single computer, network distributed emulations, physically analogous and immersive environments. Modeling and Simulation (M&S) supply the basis for such environments. Computer based modeling refers to the construction of such environments while computer simulation connotes the experimentation using them to study alternative designs and architectures.

Introduced in the last century as a rigorous systems-theory basis for discrete event modeling and simulation, the DEVS (Discrete Event System Specification) formalism[1][2] has become an engine for advances in M&S technology and the support of “build and test in virtual reality”.

1.1 CONTEXT

Discrete Event System Specification (DEVS) presents a modular hierarchical realization of Modeling and Simulation (M&S) of discrete events, which has found importance in fields ranging from science of weather forecast to economics of population and its effects. DEVS is also a unifying model that can be used to specify all systems since its semantics allow for reactive and autonomous systems at the same time. Moreover, most systems exhibit inherent parallelism[3] as pertains to the system itself in terms of more than one event can occur together at a single time, and the DEVS formalism that deals with this situation is the Parallel-DEVS (P-DEVS)[4]. It is an extension to the DEVS, and provides a way to deal with simultaneously scheduled events. It supports parallel execution of internal and external events that occur at the same simulation time. It eliminates the serialization constraints existing in the original DEVS definition and enables more efficient execution of models in parallel and distributed environments.

1.2 Problem Statement

The algorithms for proper Simulation of P-DEVS models exists and is well defined. However, proper analysis of the different algorithms in terms of implementation approach and possible suitability to specific types of systems have not been fully investigated. Implementation could yet also prove to be challenging due to subtleties of the technologies used.

1.3 Objectives and Motivation

The main objective of the study is to illustrate the global view for implementation of P-DEVS and progressively experiment on the different implementations for proper analysis. This will be achieved by creating prototypes of the different implementations and using test cases to analyze them.

The outcomes of this work are:

- A package containing different implementations of P-DEVS (object-oriented, process-oriented and hybrid).
- Toy case used for benchmarking and the results.

The motivation for our study is the need to analyze holistically, the different algorithms based on different approaches to implementation since most researchers just implement one algorithm with a single or few related approaches.

1.4 Approach Adopted

In recent times, several formalisms are in existence and are being used to model and simulate different types of systems. In this work, we focus on the DEVS formalism, which has been proven to be a universal formalism to represent Discrete Event Systems. DEVS is a sound formal framework based on generic dynamic systems concepts that supports provably correct, efficient, event-based simulation. The framework enables the construction of models in a hierarchical, modular fashion, allowing component reuse and reducing development and testing time. P-DEVS is an extension to DEVS that provides a better way to handle simultaneously scheduled events, while keeping all the major properties of the original formalism.

We present theoretical analysis of the different P-DEVS algorithms using a unified example of DEVS components and UML sequence diagram to illustrate the flow of activities. The example proposes the use of a highly interconnected model as illustrated below.

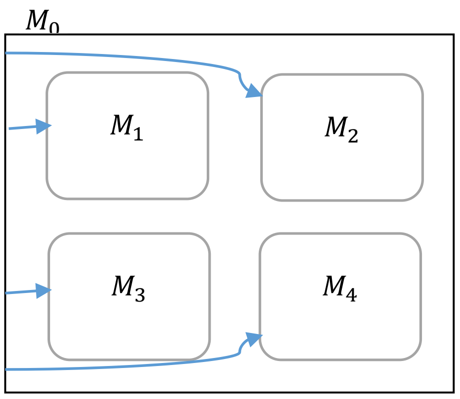


Figure 2: All connections of Model 0 at time t

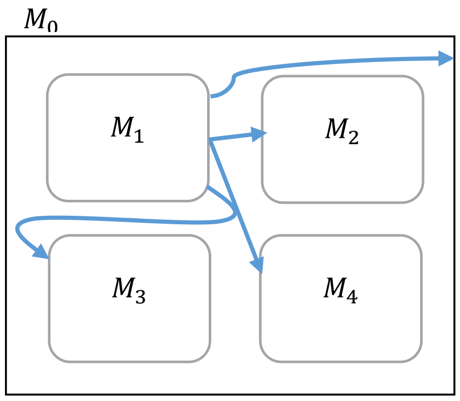


Figure 1: All connections of Model 1 at time t

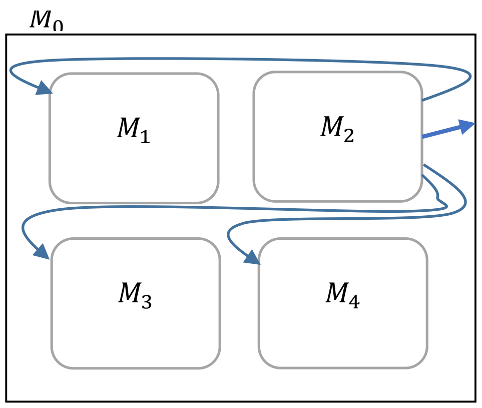


Figure 3: All connections of Model 2 at time t

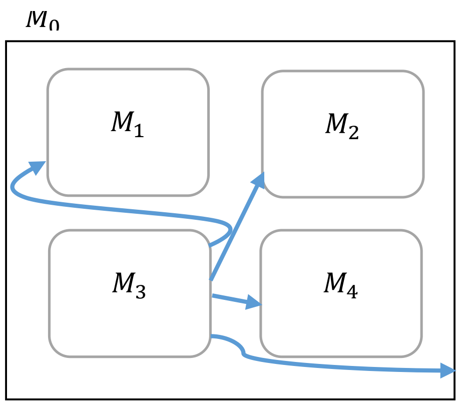


Figure 4: All connections of Model 3 at time t

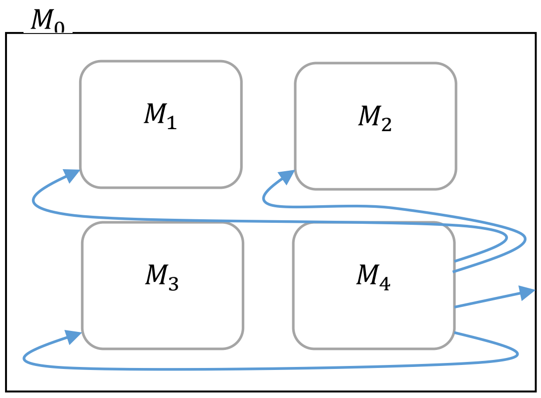


Figure 5: All connections of Model 4 at time t

Each of the individual component model of the composite model except from Model 0 has output linking all others except itself while Model 0 has External Input Coupling with all other individual models. The theoretical simulation of this composite model will be analyzed using sequence diagram of the UML (Unified Modeling Language) to illustrate its behavior whenever events occur.

We will then perform experimental analysis on the algorithms by implementing them using different software engineering approaches on the JAVA programming language and using toy cases to benchmark performance index.

1.5 ORGANIZATION OF DOCUMENT

The plan of this dissertation is as follows: Chapter 2 presents a survey of P-DEVS based M&S tools related to our work presented in subsequent chapters. In chapter 3, we review the Classic DEVS (C-DEVS) and P-DEVS formalisms, as well as behavior of selected P-DEVS simulator algorithms that exist in literature. Chapter 4 introduces the methodology of how the abstract simulators implemented was carried out. Chapter 5 presents a case study illustrating complex model specification and realization using our simulator. While Chapter 6 expatiates on the toy cases used and the performance benchmark results of the implementations. Then in Chapter 7, we conclude the work done with a summary of contributions, limitations and a discussion of future work and perspectives.

CHAPTER 2

2 RELATED WORK

This chapter presents an overview of work related to this research. We organize this synopsis of related work along the lines of tools that already implemented P-DEVS algorithms, tools that employed software engineering approaches to implement DEVS algorithms and tools that approach M&S simulation with closely related ideas.

2.1 OBJECT ORIENTED APPROACH P-DEVS IMPLEMENTATIONS

Some of the existing DEVS M&S toolkits which implements one or more P-DEVS algorithms as their main focus are listed below:

- The CD++ tool[5] is a general toolkit written in C++, which has been developed following the specifications of DEVS and Cell-DEVS. 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. The CD++ Builder[6] is also an Eclipse plugin that integrates varied applications and utility that aids in creating CD++ DEVS models, simulating and analyzing models.
- ADEVS[7], developed by Jim Nutaro (University of Arizona, U.S.A.) provides a C++ library based on Parallel DEVS and Dynamic DEVS (DynDEVS) formalisms, which developers can use to build their own models, and supports integration with other simulation environments. It includes support for standard, sequential simulation and conservative, parallel simulation on shared memory machines with POSIX threads.
- DEVS-C++[8] is a DEVS-based M&S environment written in C++ based on the parallel DEVS formalism, which implements parallel execution and supports large-scale systems. It is a modular hierarchical discrete event simulation environment implemented in the object-oriented C++ language.

- DEVSJAVA[9] is a DEVS-based M&S environment written in Java and supports parallel execution on a uni-processor. It supports higher-level, application specific modeling. It provides classes for the users to implement their own DEVS models.
- DEVS-Suite[10] is a new generation of DEVS simulator which combines the capabilities of DEVSJAVA and DEVS Tracking Environment. DEVS-Suite is part of the Component-Based System Modeling and Simulation (CoSMoS)[11] and thus enables both modeling and simulation of Parallel DEVS models.
- DEVS#[12] is C# Open Source Library of DEVS Formalism for simulation and verification analysis. It is an open source library written in C# language.
- SIMSTUDIO[13] is the virtual machine still under development in the LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes).
- DEVSim++[14] is an object-oriented software to simulate DEVS models, which was implemented in C++. The tool defines basic classes that can be extended by users to define their own atomic and coupled DEVS components.
- JAMES[15] implements DEVS theory to model and simulate agent systems. The toolkit supports software-in-the-loop simulation to test agents in virtual environments.
- JDEVS[16] is a DEVS M&S environment written in Java. It allows general purpose, component-based, object-oriented, visual simulation of models.
- PyDEVS[17] uses the ATOM3 tool to construct DEVS models and to create the code to be executed. Models are represented as a state graph used to generate Python code and then interpreted by PyDEVS.
- SimBeams[18] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, result output analysis and visualization using DEVS. It is for Modeling and Simulation: Simulation development by modular, hierarchical composition of components that adhere to a framework.
- GALATEA[19] is a simulation platform that offers a language to model multi-agent systems using an object-oriented architecture. The tool describes a real system as a set of interacting agents.

2.2 ADAPTATIONS OF M&S WORLD VIEWS TO DEVS

While the majority of toolkits mentioned above based the approach to implementation solely on constraints of the chosen programming language and software engineering techniques on DEVS formalism. This research also look at the possibilities of other M&S world view influencing the implementation of DEVS algorithms.

Examples of such in literature includes DESLib[20], a free modellica library, which has been designed and implemented to facilitate the description of discrete-event models using the Parallel DEVS formalism (using DEVSLib), and to facilitate the process-oriented modeling (in line with the process world view) of logistic systems (using SIMANLib and ARENALib). SIMANLib and ARENALib models are designed as DEVS models, and implemented using DEVSLib.

Also, SmallDEVS[21] is a DEVS based modeling and simulation tool for the programming language Smalltalk, which supports class-based as well as prototype-based object-oriented model construction and in line with the event based world view. It has been designed mainly for experiments with evolving, self-modifying models and with interactive modeling under simulation. Its meta-object protocol allows the models to be constructed from scratch and inspected and edited during run-time. SmallDEVS is different from other tools of its category, because of its openness and reflective features.

2.3 OTHER RELATED WORKS

The research group in charge of JAMES developed a set of algorithms for executing PDEVS models sequentially, in addition to the traditional abstract threaded variant, they developed a thread less variant, a basic version of a flattened algorithm, and its recursive version. They were more concerned with the performances of these various implementations, so the focus was more on the development of the approaches on a specific algorithm; such as threaded, threadless, flat-sequential and recursive flat-sequential rather than categorization of all algorithms in literature and implementation approaches, and how they perform generally [22]. Figures 6 to 8 shows the high level representations of the various implementations.

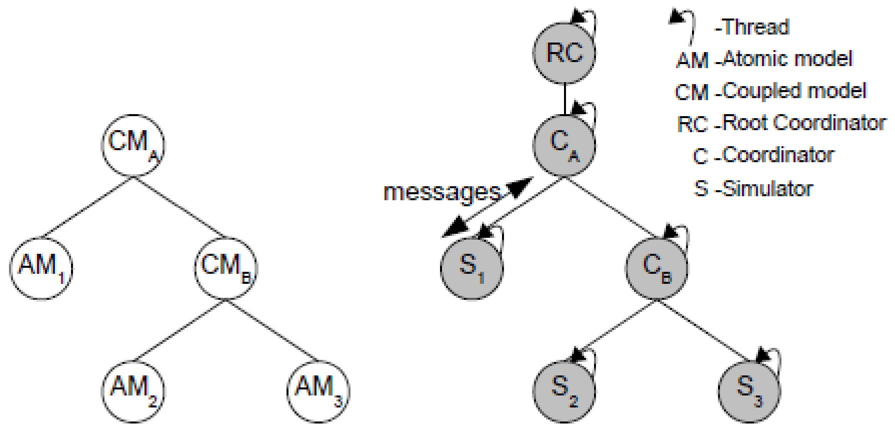


Figure 6: Threaded abstract sequential simulator

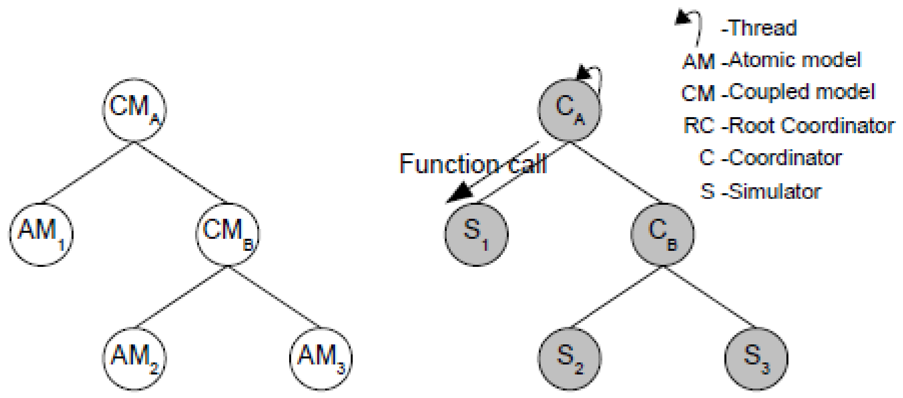


Figure 7: Threadless abstract sequential simulator

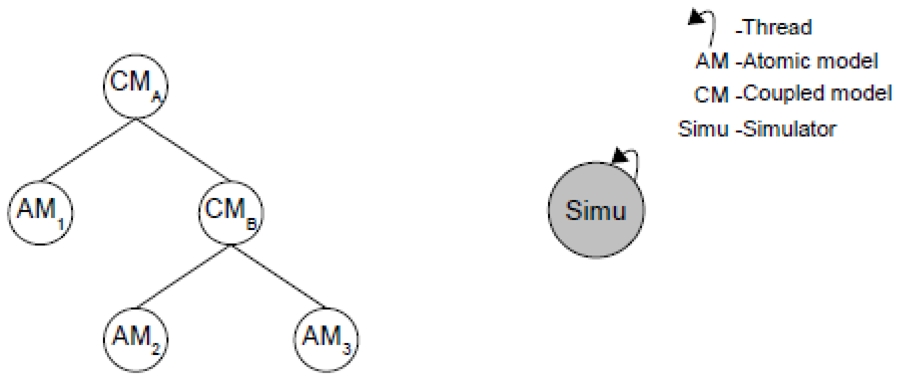


Figure 8: Flat abstract sequential simulator

CHAPTER 3

3 STATE OF THE ART

3.1 DEVS

Discrete Events systems Specification is a formalism for modeling and simulating dynamic event discrete systems (DEDS). It defines 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. It allows hierarchical decomposition of the model by defining a way to couple existing DEVS models. It is a unified framework for developing real-time software systems in which logical analysis, performance evaluation, and implementation can be performed, all based on the DEVS formalism[23]. The DEVS framework is based on systems engineering principles and is used for M&S in many application domains. This framework supports a number of important features, such as component-based hierarchical simulation model development, scalability, reusability, and distributed simulation.

The conceptual framework underlying the *DEVS formalism* is shown in Figure
9. The modeling and simulation enterprise concerns three basic objects:
The

- ***Real system,*** in existence or proposed, which is regarded as fundamentally a source of data
- ***Model***, which is a set of instructions for generating data comparable to that observable in the real system. The *structure* of the model is its set of instructions. The *behavior* of the model is the set of all possible data that can be generated by faithfully executing the model instructions.
- ***Simulator***, which exercises the model's instructions to actually generate its behavior.
- ***Experimental frame,*** which captures how the modeler's objectives impact on model construction, experimentation and validation.

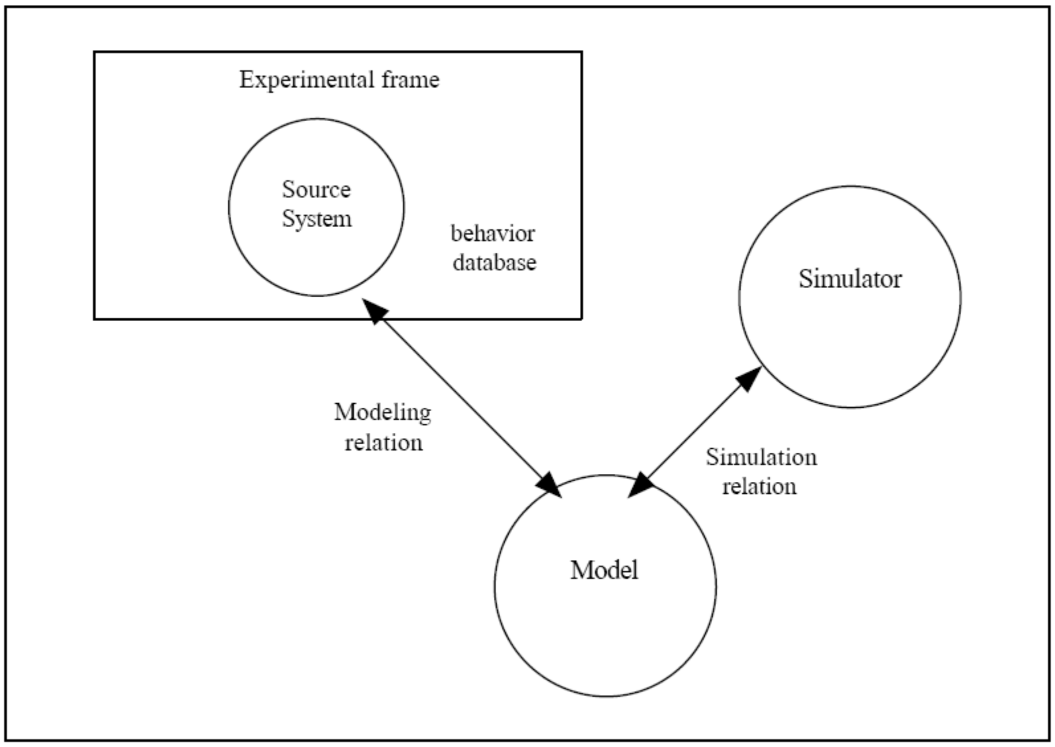


Figure 9: Entities and their relations in DEVS

A DEVS model processes an input event trajectory and, according to that trajectory and its own initial conditions, provokes an output event trajectory. This input/output behavior is depicted in Figure 10

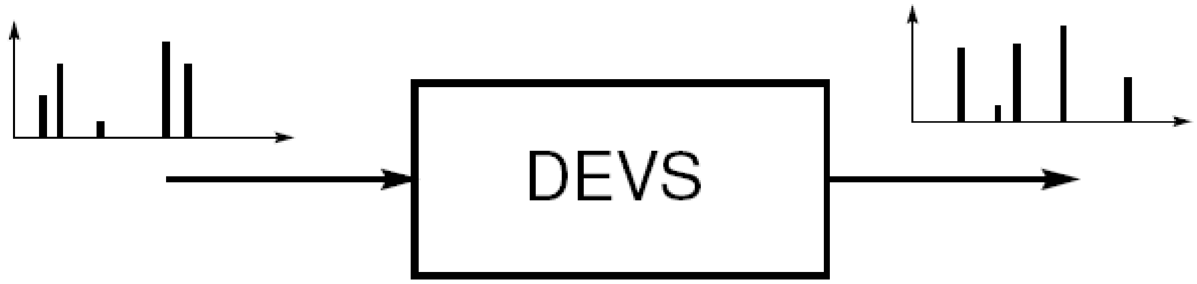


Figure 10: Input/output behavior of a DEVS model

The behavior of a DEVS model is expressed in a way that is quite common in automata theory. This kind of representation consists in enumerating some sets and functions that define the system dynamics in accordance with certain rules. Since the rules are always the same in a given formalism, they are not mentioned in each model.

A real system modeled using DEVS can be described as a composition of *atomic* and *coupled* components. An *atomic* model is defined by:

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

Where:

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

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

S is the set of sequential states;

δ_{ext}: $Q \times X \to S$ is the external state transition function;

Where Q = $\{(s,e) / s \in S, e \in [0, ta(s)] \}$ and e is the elapsed time since the last state transition.

δ_{int}: $S \to S$ is the internal state transition function;

λ: $S \to$ Y is the output function;

ta: $S \to R_{0+} \cup \infty$ is the time advance function;

A DEVS model is in a state $s \in S$ at any given time. In the absence of external events, it remains in that state for a lifetime defined by *ta*(*s*). A transition that occurs due to the expiration of time indicated by *ta*(*s*) is called an internal transition. When *ta(s)* time expires, the system outputs the value $\lambda(s)$ and then changes to a new state given by $\delta_{int}(s)$. On the other hand, an external transition occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by $\delta_{ext}(s, e, x)$ where *s* is the current state, *e* is the time elapsed since the last transition and $x \in X$ is the external event that has been received.

The time advance function can take any real value between 0 and ∞. A state for which *ta(s) = 0* is called a transient state. In contrast, if the *ta(s)* = ∞, then *s* is said to be a passive state, in which the system will remain perpetually unless an external event is received.

A DEVS *coupled model* as illustrated in figure 11 is composed of several atomic or coupled sub-models. The later model can itself be employed as a component of larger model. This way the coupled models allow to construct models hierarchically. Coupled DEVS is defined as a structure. It is formally defined by:

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

Where:

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

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

D is the set of the component names, and the following constraints apply to the components, which are also DEVS models:

for each $d \in D$

$M_d = < X_d, Y_d, S, \delta_{int}, \delta_{ext}, \lambda, \Gamma a >$ is a component Atomic DEVS model,

External input couplings (EIC) connect external inputs to component inputs,

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

External output couplings (EOC) connect component outputs to external outputs,

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

Internal couplings (IC) connect component outputs to component inputs,

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

Direct feedback loops are not allowed, i.e., no output port of a component may be connected to an input port of the same component.

Select is the tie-breaker function, where select: subset of $D \rightarrow D$, such that for any non-empty subset E, select (E) ∈ E.

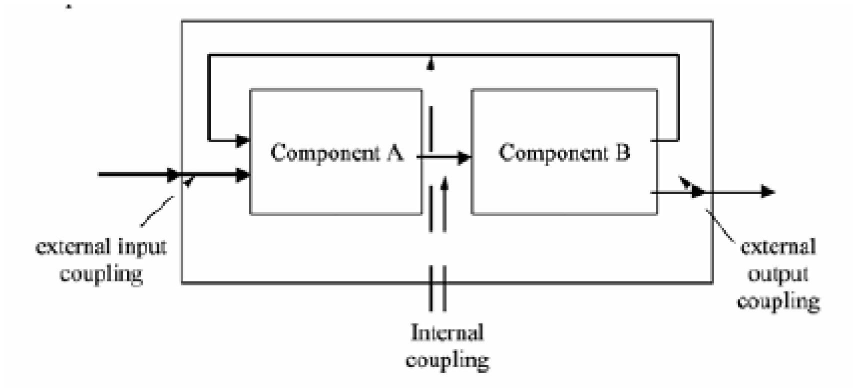


Figure 11: Schema of a Coupled Model

A coupled model groups several DEVS as shown in the figure above, into a compound model that can be regarded as a new DEVS model. The closure property guarantees that the coupling of several class instances results in a system of the same class [1]. This property allows hierarchical model construction.

In addition, each coupled model has its own input and output events, as defined by the X and Y sets. When external events are received, the coupled model has to redirect the inputs to one or more components. Similarly, when a component produces an output, it may have to map it as input to another component, or as an output of the coupled model itself. Mapping between ports is defined by the *Z* function.

Multiple components can be scheduled for an internal transition at the same time in a coupled component, and ambiguity may arise. If the first component to execute its internal transition produces an output that maps to an external event for another component that is already scheduled for an internal transition, then it is not clear which transition this second component should execute first. Two alternatives exist:

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

By the use of *select* function, the DEVS formalism enables a simple way to solve this ambiguity. The function defines an order over the components so that only one component of the group of imminent models is allowed to have *e = 0*. The other imminent models are divided in two groups: those that receive an external output from this model, and the rest. The former will execute their external transition functions with *e = ta(s)*, the latter will be imminent during the next simulation cycle which may require again the use of the *select* function to decide which model will execute first. This strategy for tiebreaking is rigid and, in addition, it introduces strict serialization in the execution of components. The serialization introduced by this approach becomes visible when the *select* function has to be used to determine the priority in which the components have to be executed. For example, the *select* function is used to determine which atomic component has priority over the rest to execute its internal transition function when many interconnected atomic models are imminent. The figure below shows a typical trajectory in DEVS models during simulation.

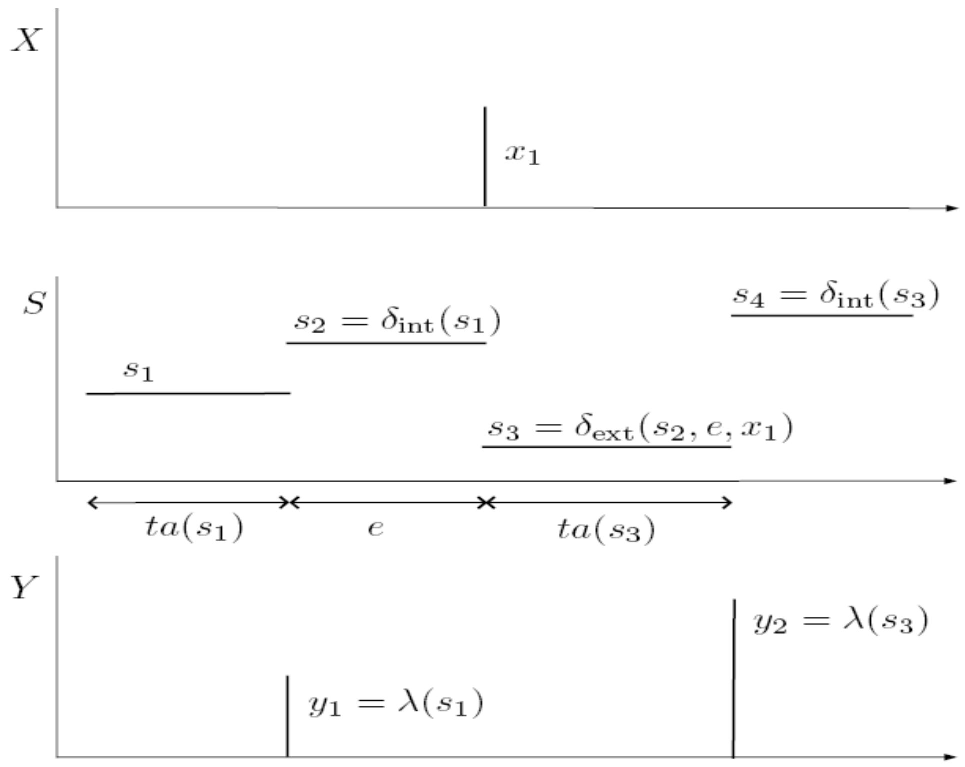

X
x_1
S
s_1
$s_2 = \delta_{\text{int}}(s_1)$
$s_3 = \delta_{\text{ext}}(s_2, e, x_1)$
$s_4 = \delta_{\text{int}}(s_3)$
$ta(s_1)$
e
$ta(s_3)$
Y
$y_1 = \lambda(s_1)$
$y_2 = \lambda(s_3)$

Figure 12: Behavior of DEVS models during simulation

3.2 Parallel DEVS

P-DEVS is an extension to DEVS that provides a more flexible way of dealing with these ambiguities. It removes the sequential management of events, allowing simultaneous occurrences of events. It also facilitates the description of the changes in the state during confluent events (i.e. simultaneous internal and external events) by defining the confluent transition function (δ_{conf})[24].Atomic models provide an additional confluent function to specify collision behavior for events that might be scheduled simultaneously. Since serialization constraints existing in the original DEVS formalism are now eliminated, P-DEVS permits increased degrees of parallelism that can be exploited in parallel and distributed environments. Consequently, Parallel DEVS was the formalism chosen as the foundation for this work.

P-DEVS models are described very much like DEVS models. An atomic Parallel DEVS model is defined as:

$$M_{TLC} \ = < X^b, Y^b, S, \delta_{int}, \delta_{ext}, \lambda, \Gamma a, \delta_{conf} \ >$$

Where

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

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

S is the set of sequential states;

δ_{ext}: Q x $X^b \rightarrow$ S is the external state transition function;

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

δ_{conf}: Q x $X^b \rightarrow$ 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 $\in$ S , 0 $\leq$ e $\leq$ ta(s)} the set of total states.

There are two main differences between a basic DEVS and a basic Parallel DEVS model. First, the external transition function uses a bag of events instead of a single event. This allows multiple events to be processed simultaneously. Since external events received by the component are added to a bag (X^b), external transition functions can combine the functionality of a number of external transitions into a single one. Second, the model specification includes a confluent transition function (δ_{conf}). When a collision between the internal and external functions occurs, the confluent function determines the new state of the model.

The semantics of P-DEVS are similar to those of DEVS. A basic model is in a state *s* at any given time. In the absence of external events, the model remains in that state for a lifetime period defined by ta(*s*). When that time expires, an internal transition takes place; the system outputs the value *l(s*) and then it changes to the state specified by *d_{int}(s)*. If one or more external events

$E = \{x_1 .. x_n / x \in X_M\}$ occurs before ta(*s*) expires, i.e., while the system is in total state (s, e) with *e* < ta(s), the new state will be given by the model's external transition function, *dext*(*s*,*e*,*E*). P-DEVS allows a better way to deal with collisions. External and internal transitions are in conflict when external events *E* are received when *e* = ta(*s*). In such cases, the new state of the model can be given by *dext*(*dint(s)*,*e*,*E*) or *dint(dext*(*s*,*e*,*E*)*)*. Hence, modelers have a flexible way of indicating the appropriate behavior for each model in the confluent function (δ_{conf}), which is triggered in case of collisions.

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

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

The definitions for the set of input and output events (*X* and *Y*), components (*D* and *Md*), and couplings (*EIC*, *EOC*, and *IC*) follow the specifications of DEVS coupled models presented earlier in this chapter.

If multiple components in a coupled model are imminent, all their outputs are first collected and mapped to their influencees. Then, the corresponding transition function is executed for every model.

A *simulator* object manages an associated atomic object, handling the execution of its d*int*, d*ext*, d*conf* and ta(s) functions. A *coordinator* object manages an associated coupled object. Only one *root coordinator* exists in a simulation. It manages global aspects of the simulation. It is involved with the topmost-coupled component, which has the highest level in the model hierarchy. Moreover, the *root coordinator* maintains the global time, and it starts and stops the simulation process. Lastly, it receives the output results that must be sent to the environment.

The simulation process is message driven; processors exchange messages to advance the execution of the model. Each message contains information to identify the *sender* and the *receiver*. A *time-stamp* for the message and an associated *value* are also included in the packet. Two main categories of messages exist: synchronization and content messages. These categories consist of several types of messages, examples include:

Synchronization messages:

- @ *Collect message*
- * *Internal transition message*
- done *Done message*

Content messages:

- x *External transition message*
- y *Output message*

Processors also have internal variables to keep the time of the simulation:

tL: *Time of last event*

tN: *Time of next event*

And a *bag* to store external messages. All the important times during simulation is described in figure 13 below.

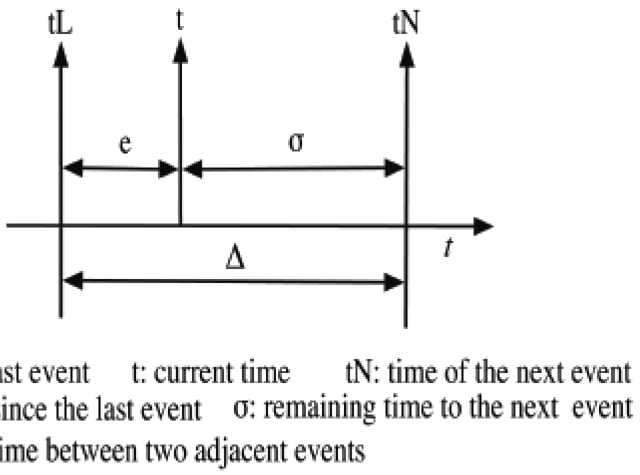

tL
t
tN
e
σ
Δ
t
tL: time of the last event
t: current time
tN: time of the next event
e: elapsed time since the last event
σ: remaining time to the next event
Δ: inter-arrival time between two adjacent events

Figure 13: time variables during DEVS simulation

3.3 ALGORITHMS FOR PARALLEL-DEVS SIMULATION

We describe the simulation mechanism more rigorously by presenting the behavior of each DEVS processor, namely *simulator*, *coordinator*, and *root coordinator* based on the different algorithms that exist in literature. We go on to analyze the behavior of this algorithms on the system presented in Figures 1 to 5 as a form of theoretical analysis of how they operate in simulations.

Four prominent algorithms have been identified and analyzed as shown below.

3.3.1 Ziegler and Chow's P-DEVS Algorithm

Ziegler and Chow's paper on revision of the hierarchical, modular Discrete Event System Specification (DEVS) modeling formalism[4] presented this algorithm for the abstract simulators(root coordinator, coordinator and simulator). The Hierarchical simulator uses the following types of messages:

- *Initialization messages (I,t)*
- *Collect message (@, t)*
- *Transition messages (*,t)*
- *Input content messages (q, t)*
- *Output content messages (y, t)*
- *Done messages (done, t)*

Root Coordinator

An initialization message (*i, t*) is transmitted at the beginning of the simulation to the topmost coordinator attached to the topmost coupled DEVS model.

A collect message (@, t) is transmitted and done message (done, t) is waited for in the first part of iteration to the topmost coordinator until the end of the simulation is reached.

An internal state transition message (**, t*) is transmitted and done message (done, tn) is waited for to update (t = tn) as the second part of the iteration to the topmost coordinator until the end of the simulation is reached.

Coordinator

Upon receipt of an initialization message (*i, t*) from parent component, another one is transmitted to all simulators and coordinators of the children component of a coupled DEVS. After that the time of the last event *tl* and the time of the next internal event *tn* of the coupled DEVS is calculated and updated and a done message (done, tn) is dispatched to the parent component.

If a collect message (@, t) is received from parent component, it is forwarded to all subordinate components that are imminent and done message (done, t) is expected back, to return a done message to the parent component.

If an internal state transition message (**, t*) is received, all events in *mail* are transmitted to their receivers and to the superior coordinator. A star message will be forwarded to all components of a coupled DEVS which are member of the set *imminent* (IMM). Finally, the event times *tl* and *tn* are updated.

Any input message (*x, t*) is forwarded according to the coupling relations *Zi,d* to other simulators or coordinators that will eventually cache the message in the appropriate simulator's bag.

All incoming output messages (*y, t*) from inferior simulators and coordinators are saved together with the name of the originator inside an internal list *mail*. After all members in the set IMM of the coordinator have answered with an output message, a done message is issued to prepare to receive an internal transition message (*, t).

<u>Simulator</u>

An initialization message (*i, t*) implies the calculation of the time of the next internal event *tn* and the time of the last event *tl* of the considered atomic DEVS and also a corresponding done message (done, tn) reflecting the new time of next event.

A collect message (*@, t*) indicates an output event. Therefore the output function $y = \lambda(s)$ of the attached atomic DEVS is calculated and the output events are transmitted with an output message ($\lambda(s)$*, t*) to the superior coordinator. A done message is also dispatched to signify the end of output event.

An internal state transition message (**, t*) indicates an internal event. Therefore the simulator decides whether the attached atomic DEVS executes $d_{int}(s)$ (internal transition) or $d_{ext}(s,e,X^b)$ (external transition) or $d_{conf}(s,X^b)$ (confluent transition) depending on the imminence of the atomic DEVS and the presence of contents in the bag (X^b).

- ・ The internal transition function $\delta int(s)$, if the simulation time $t = tn$ and the input event bag X^b contains an empty set.

- • The confluent function *δconf(s, X^b)*, if the simulation time *t = tn* and the input event bag X^b contains a non-empty bag of events.
- • The external transition function *δext(s, e, X^b)*, if *tl ≤ t < tn* and the input event bag X^b 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 and a corresponding done message (done, tn) reflecting the new time of next event is sent to the superior coordinator. Any input message (*x, t*) cached in the input event bag of the simulator.

3.3.2 Analysis of Ziegler and Chow's PDEVS Algorithm

The behavior of the simulation approach is investigated with message sequence diagrams for the figures 1 to 5 system example. This shows how messages are transmitted to realize simulation. Figure 14 shows the object tree of the system while figure 15 illustrates the system behavior.

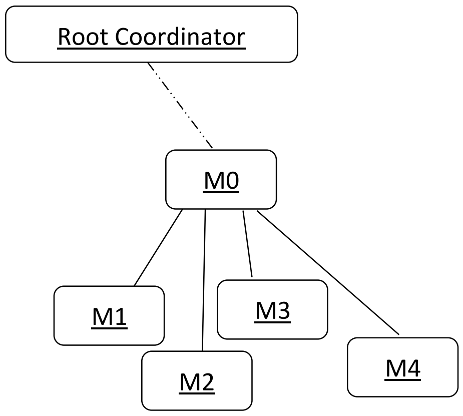


Figure 14: Object tree of example system

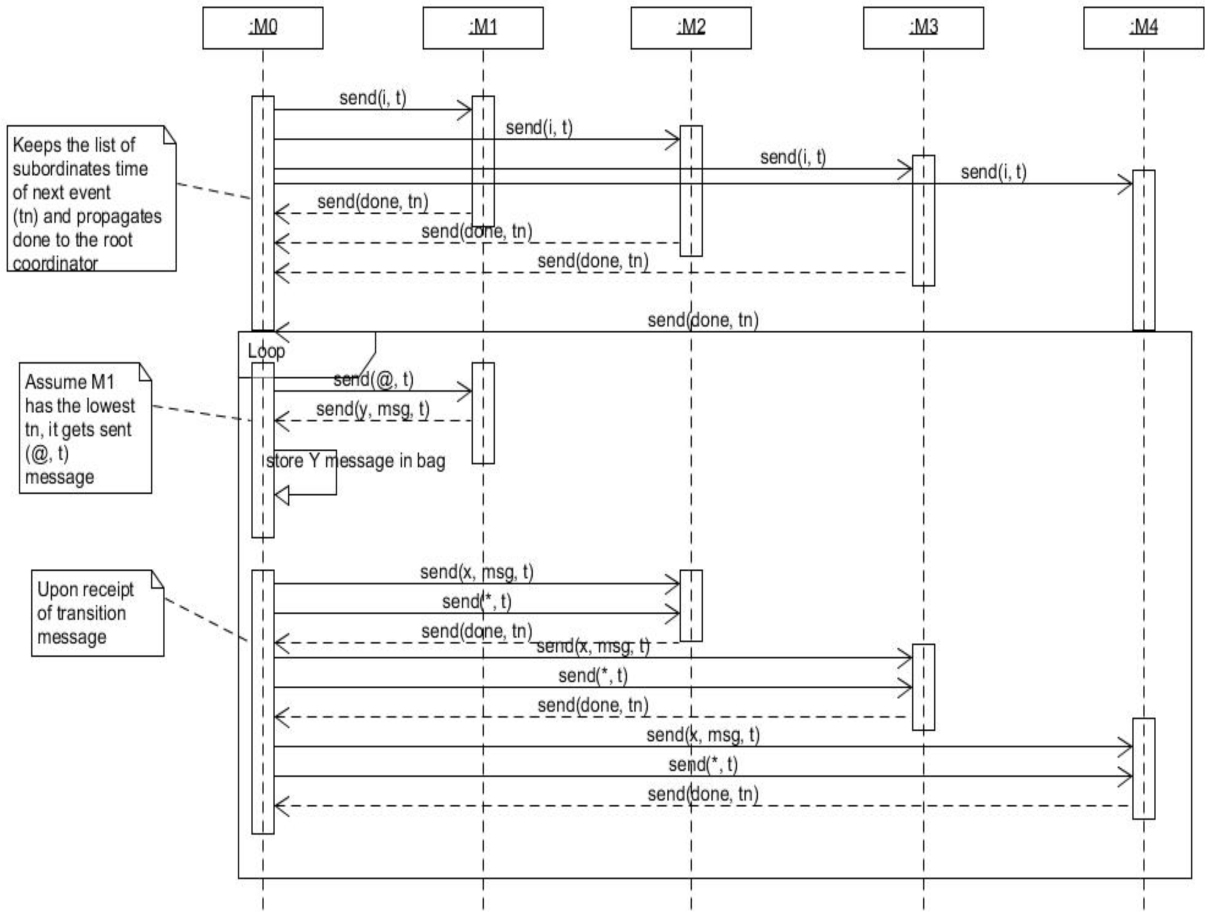


Figure 15: Sequence diagram for Ziegler and Chow algorithm

3.3.3 Chow's PDEVS Algorithm

Chow's paper on Parallel DEVS: A parallel, hierarchical, modeling formalism and its distributed simulator[25] presents another algorithm that easily support distributed systems. The Hierarchical simulator uses the following types of messages:

- initialization message (*i, t*)
- internal state transition message (**, x_count, t*)
- output/input message (*#, content, t*)
- ending message (*Done, tn*)

Root-coordinator

An initialization message (i, t) is transmitted at the beginning of the simulation to the topmost coordinator attached to the topmost coupled DEVS model, which propagates it to all components in the hierarchy to determine the respective time of next event (tn) and time of last event (tl).

An internal state transition message (*, 0, t) is transmitted to the topmost coordinator. If an ending message (Done, tn) is received from the subordinate DEVS coordinator the simulation time t will be advanced to tn and a new internal state transition message (*, 0, t) is sent to the topmost coordinator. This is done in a loop till the end of the simulation

Coordinator

An initialization message (i, t) is transmitted to all simulators and coordinators of the components of a coupled DEVS. After that the time of the next internal event tn of the considered coupled DEVS is calculated and an ending message (Done, tn) is transmitted to the superior coordinator.

If an internal state transition message (*, x_count, t) is received it will be forwarded as (*, i_count, t) message to all components of a coupled DEVS which are member of the sets imminent (IMM) or receivers (INF). Thereby, the variable x_count contains the amount of influencer of the coupled DEVS itself and the variable i_count contains the total amount of influencer of a component defined inside a coordinator. The internal variable *semaphore_count* of the coupled DEVS will be increased each time if any internal state transition message is transmitted to another inferior DEVS component. If *semaphore_count* reaches zero an ending message (*Done*, *tn*) will be transmitted to the superior coordinator.

Output/input-messages (*#, content, t*) are forwarded according to the coupling relations *Zi,d* to other simulators or coordinators.If any ending message (*Done*, *tn*) is received the next internal event time *tn* of the originator is saved inside an event list. Moreover, the value of *semaphore_count* is decreased each time by one.

Simulator

An initialization message (*i, t*) leads to the calculation of the time of the next internal event *tn* of the considered atomic DEVS. After that an ending message (*Done*, *tn*) will be transmitted to the superior coordinator.

An internal state transition message (**, x_count, t*) indicates an external or internal event. At first the internal variable *semaphore_count* is set to *x_count*.

If the simulation time *t* is equal to the time of the next internal event *tn* the output function *y = λ(s)* of the attached atomic DEVS is calculated and the output events are transmitted with an output message (*#, λ(s), t*) to the superior coordinator.

- If *semaphore_count = 0* only an internal event takes place and consequently the internal transition function *δint(s)* of the attached atomic DEVS will be carried out.
- If *semaphore_count ≠ 0* the external and internal event should take place at the same time. Therefore, the confluent function *δconf(s, e, x)* of the attached atomic DEVS will be carried out when *semaphore_count* is set to zero during execution.

If the simulation time *t* is not equal to the time of the next internal event *tn* only an external event takes place. Therefore the external transition function *δext(s, e, x)* of the attached atomic DEVS will be carried out when *semaphore_count* is set to zero during execution.

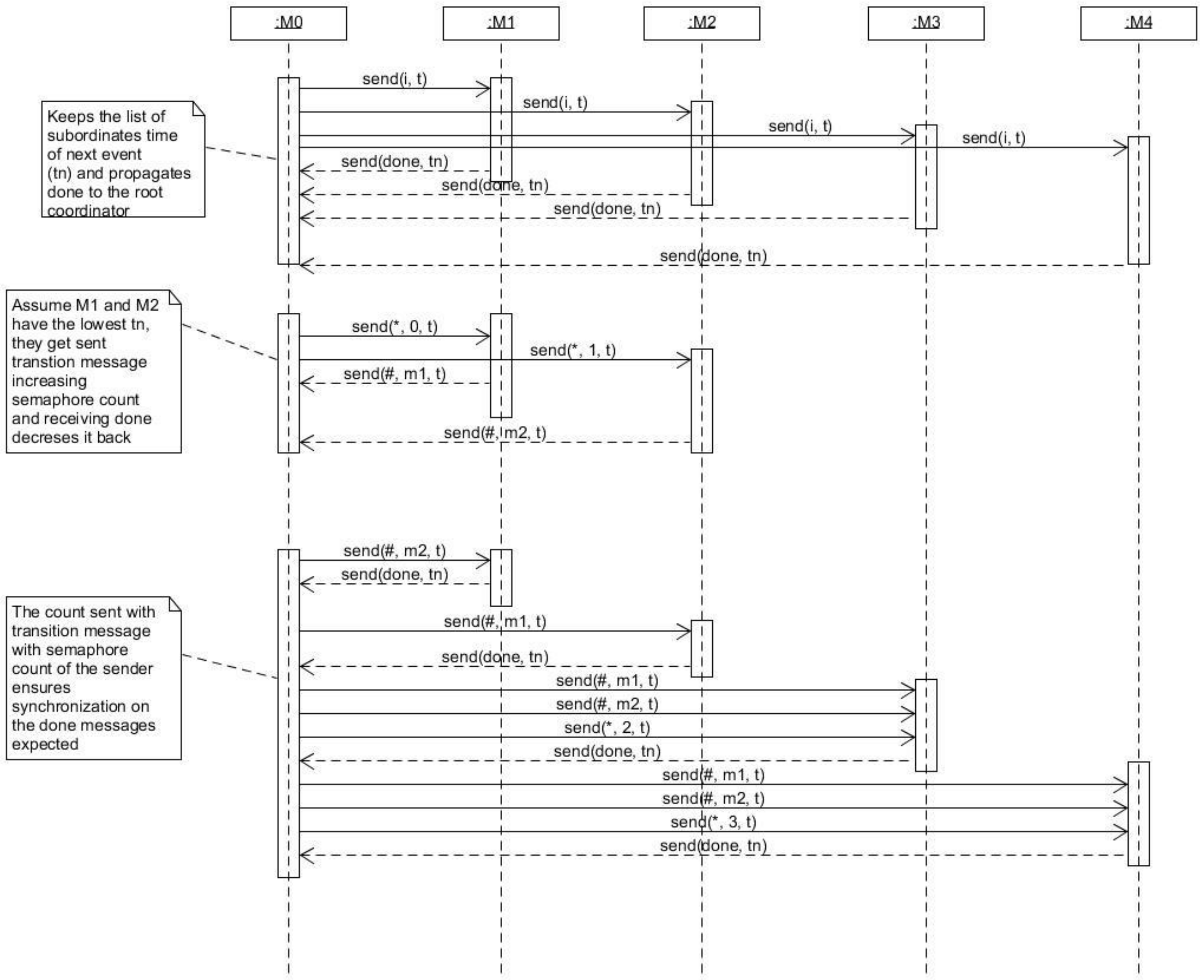
When *semaphore_count* is set to zero the time of the next event *tn* is calculated and an ending message (*Done*, *tn*) is transmitted to the superior coordinator.

Any content of an input message (*#, content, t*) is saved in a vector *xb* (event bag) and the value of *semaphore_count* is decreased by one. The internal variable *semaphore_count* reaches zero if *all input events at the simulation time t* have been saved inside the event bag X^b.

It is important to belabor on the variable *x_count* in the internal state transition message of this algorithm. This variable tells every simulator of an atomic DEVS, ***how many input events must be arrive until any transition function is carried out***. Of course the amount of input events must be calculated before by every coordinator.

3.3.4 Analysis of Chow's PDEVS Algorithm

The behavior of this other simulation approach is also investigated with message sequence diagrams for the figures 1 to 5 system example as shown in figure 16.



:M0
:M1
:M2
:M3
:M4
Keeps the list of subordinates time of next event (tn) and propagates done to the root coordinator
send(i, t)
send(i, t)
send(i, t)
send(i, t)
send(done, tn)
send(done, tn)
send(done, tn)
send(done, tn)
Assume M1 and M2 have the lowest tn, they get sent transtion message increasing semaphore count and receiving done decreses it back
send(*, 0, t)
send(*, 1, t)
send(#, m1, t)
send(#, m2, t)
send(#, m2, t)
send(done, tn)
The count sent with transition message with semaphore count of the sender ensures synchronization on the done messages expected
send(#, m1, t)
send(done, tn)
send(#, m1, t)
send(#, m2, t)
send(*, 2, t)
send(done, tn)
send(#, m1, t)
send(#, m2, t)
send(*, 3, t)
send(done, tn)

Figure 16: Sequence diagram for Chow algorithm

3.3.5 Schwatinski and Pawletta PDEVS Algorithm

Tobias Schwatinski and Thorsten Pawletta from the University of Applied Sciences Wismar, Germany proposed another algorithm for PDEVS[26] in which the message concepts are extended. The abstract simulator uses the following types of messages:

- initialization message (*i, t*)
- internal state transition message (**, t*)
- input message (*x, t*)
- output messages:
    - ordinary output message (*y, t*)
    - interpellation message (*'Y', t*)

The first 4 messages have the same meanings as Zeigler and Chow's algorithm messages but the *interpellation message* is introduced newly. It can only be transmitted from a coordinator to subordinate simulator or coordinator components. Any simulator reacts on an interpellation message (*'Y', t*) with the execution of the output function *y = λ(s)* just like the collect message of Zeigler and Chow's algorithm, from its associated atomic DEVS system and transmits the output events *y* back to the parent coordinator with an ordinary output message (*y, t*).
Any coordinator forwards the interpellation message (*'Y', t*) to all components in the set IMM. The real difference lies at an internal variable *layer* that is introduced in the coordinator algorithm. It influences the behavior of the coordinator if an internal state transition message (**, t*) or an ordinary output message (*y, t*) occur.

Root-coordinator

An initialization message (*i, t*) is transmitted at the beginning of the simulation to the topmost coordinator attached to the topmost coupled DEVS model.

An internal state transition message (**, t*) with *t = tn* of the topmost coordinator is transmitted in a loop to the topmost coordinator till the end of the simulation is reached.

Coordinator

An initialization message (*i, t*) is transmitted to all simulators and coordinators of the components of a coupled DEVS. After that the time of the last event *tl*, the time of the next internal event *tn* and the local eventlist are updated. Furthermore, *layer* is set to *passive* and an internal list *mail* which is used for saving output events is set to *Ø*.

If an internal state transition message (**, t*) is received and

- If *mail = Ø,* then *layer* is set to *active* and an interpellation message (*'Y', t*) is transmitted to all subordinate simulators/coordinators which are member of the set *imminent* (IMM). After that the program execution is blocked at the test condition "if *mail ≠ Ø*" till *layer* is set to *passive*.
- If *mail ≠ Ø* than all events in *mail* with *yi ≠ Ø* are transmitted with an input message (*x, t*) to their receivers. Moreover, the internal state transition message (**, t*) is sent to all members of the set IMM which have actually no input events in *mail*. The variable *layer* is set to *passive* and the internal list *mail* is set to *Ø*, i.e. event entries with *yi = Ø* are deleted without any processing. Finally, both event times *tl* and *tn* and the local event-list are updated.

If an input message (x, t) is received and

- if the set IMM = *Ø* than any input message (*x, t*) is forwarded according to the coupling relations *Zi,d* to other simulators or coordinators and later the event times *tl* and *tn* and the local event-list are updated.
- if the set IMM ≠ *Ø* than the input event *x* from the input message (*x, t*) is saved together with the name of the originator inside the internal list *mail* and an internal state transition message (**, t*) is transmitted to the coordinator itself. In doing so, the coordinator executes two successive computation tasks. At first it treats the received input message (*x, t*) and afterwards it immediately processes its self-sent internal state transition message (**, t*).

All incoming output messages (*y, t*) from subordinate simulators and coordinators are saved together with the name of the originator in the internal list *mail*. After all members in the set IMM of the coordinator have answered with an output message (*y, t*) and

- If *layer* = *passive*, than all messages inside *mail*, which are addressed to the superior coordinator are sent to it. If there is no message in *mail* the empty output message (*Ø, t*) must be sent.
- If *layer* = *active*, than *layer* is set to *passive* and the blocked internal state transition message is unblocked and executed further.

Any interpellation message (*'Y', t*) is forwarded to all members of the set IMM.

Simulator

An initialization message (*i, t*) leads to the_calculation of the time of the next internal event *tn* and the time of the last event *tl* of the associated atomic DEVS.

An internal state transition message (**, t*) indicates an internal event. The internal transition function *δint(s)* of associated atomic DEVS is called.

Upon receiving an interpellation message (*'Y', t*) the output function *y = λ(s)* of the associated atomic DEVS is called and the output events are transmitted with an output message (*λ(s), t*) to the parent coordinator.

An input message (*x, t*) indicates an internal or external event and the appropriate transition function is carried out:

- The confluent function *δconf(s, e, x)*, if the simulation time *t = tn* and the input event bag contains a non-empty bag of input events.
- The external transition funct. *δext(s, e, x)*, if *tl ≤ t < tn* and the input event bag contains a non-empty bag of events.

After calling any transition function the last event time *tl* is set to the current simulation time *t* and the time of the next event *tn* is calculated.

3.3.6 Analysis of the Schwatinski and Pawletta PDEVS Algorithm

Figure 17 is used to illustrate the system dynamics of the system depicted in Figures 1 to 5 as shown below.

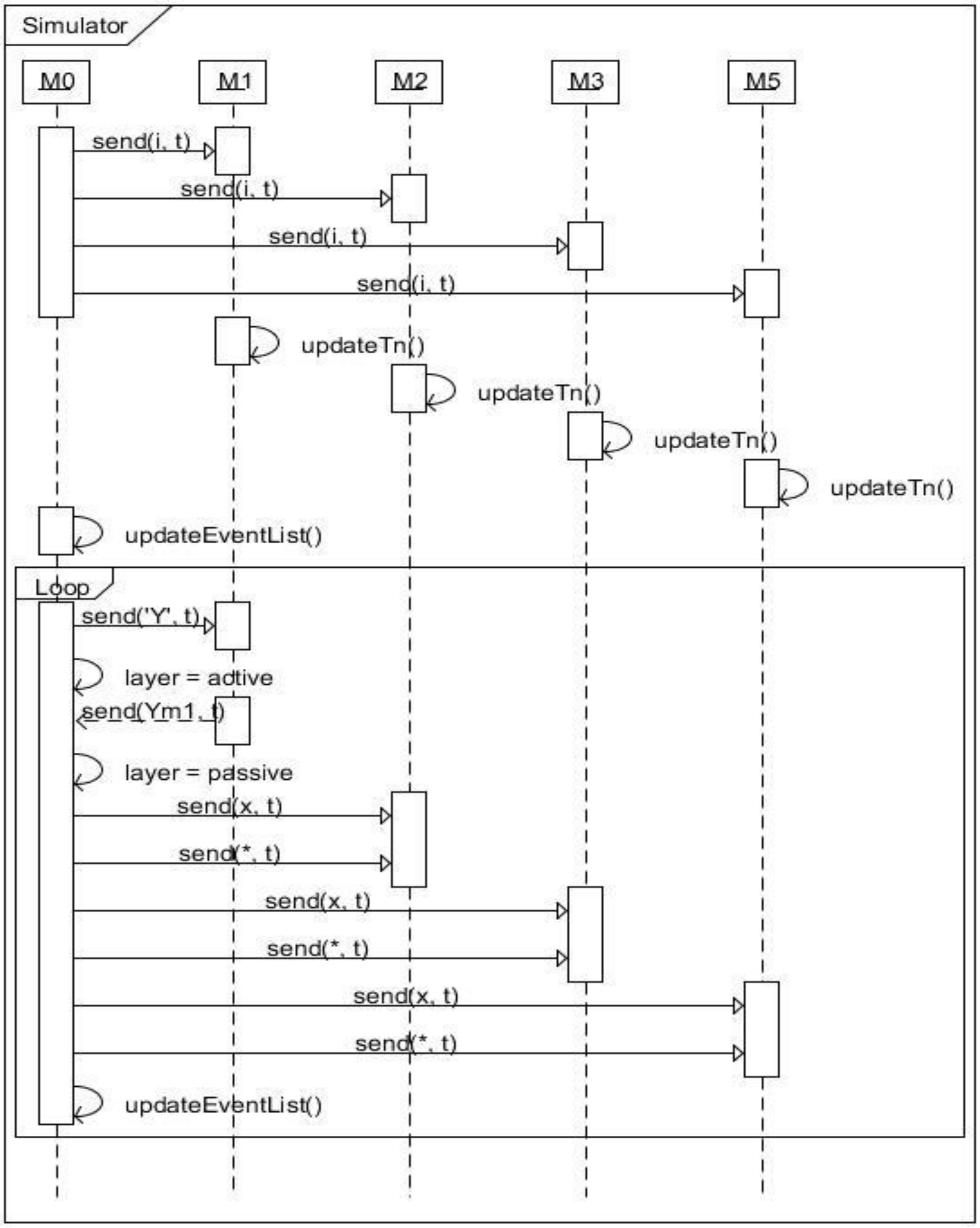


Figure 17: Sequence diagram for Schwatinski algorithm

3.3.7 Ziegler's original algorithm

In Ziegler's book on Theory of Modeling and Simulation (First Edition)[2] lies the original proposed PDEVS algorithm which makes use of the following messages:

- *Initialization messages (I,t)*
- *Transition messages (*,t)*
- *Input messages (q, t)*
- *Output messages (y, t)*
- *Done messages (done, t)*

Root Coordinator

An initialization message (*i, t*) is transmitted at the beginning of the simulation to the topmost coordinator attached to the topmost coupled DEVS model.

An internal state transition message (**, t*) with *t = tn* of the topmost coordinator is transmitted in a loop to the topmost coordinator till the end of the simulation is reached.

Coordinator

Upon receipt of an initialization message (*i, t*) from parent component, it is transmitted to all simulators and coordinators of the children component of a coupled DEVS. After that the time of the last event *tl* and the time of the next internal event *tn* of the coupled DEVS is calculated and updated and a done message (done, tn) is dispatched to the parent component.
If an internal state transition message (**, t*) is received, it will be forwarded to all components of a coupled DEVS which are member of the set *imminent* (IMM).
Any input message (*x, t*) is forwarded according to the coupling relations *Zi,d* to other simulators or coordinators. Moreover, the empty input message (*Ø, t*) is transmitted to all members of the set IMM which have currently no input event in *x*. Finally the event times *tl* and *tn* are updated.
All incoming output messages (*y, t*) from inferior simulators and coordinators are saved together with the name of the originator inside an internal list *mail*. After all members in the set IMM of the coordinator have answered with an output message (*y, t*) all events in *mail* are transmitted to their receivers and to the superior coordinator.
The empty input message (*Ø, t*) is also transmitted to all members of the set IMM which have currently no input event in *mail*. Finally, the event times *tl* and *tn* are updated.

Simulator

An initialization message (*i, t*) implies the calculation of the time of the next internal event *tn* and the time of the last event *tl* of the considered atomic DEVS and also a corresponding done message (done, tn) reflecting the new time of next event.

An internal state transition message (**, t*) indicates an internal event. Therefore the output function *y = λ(s)* of the attached atomic DEVS is calculated and the output events are transmitted with an output message (*λ(s), t*) to the superior coordinator.

An input message (*x, t*) indicates an internal and/or external event and the appropriate transition function is carried out, which can be:

- ・The internal transition function *δint(s)*, if the simulation time $t = tn$ and the input event *x* contains an empty set.
- ・The confluent function *δconf(s, e,* X^b*)*, if the simulation time $t = tn$ and the input event *x* contains a non-empty bag of events.
- ・The external transition function *δext(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.

3.3.8 Analysis of Ziegler's PDEVS Algorithm

The behavior of the simulation approach is investigated with message sequence diagrams for the figures 1 to 5 system example. This shows how messages are transmitted to realize simulation. Figure 18 illustrates the behavior of the system.

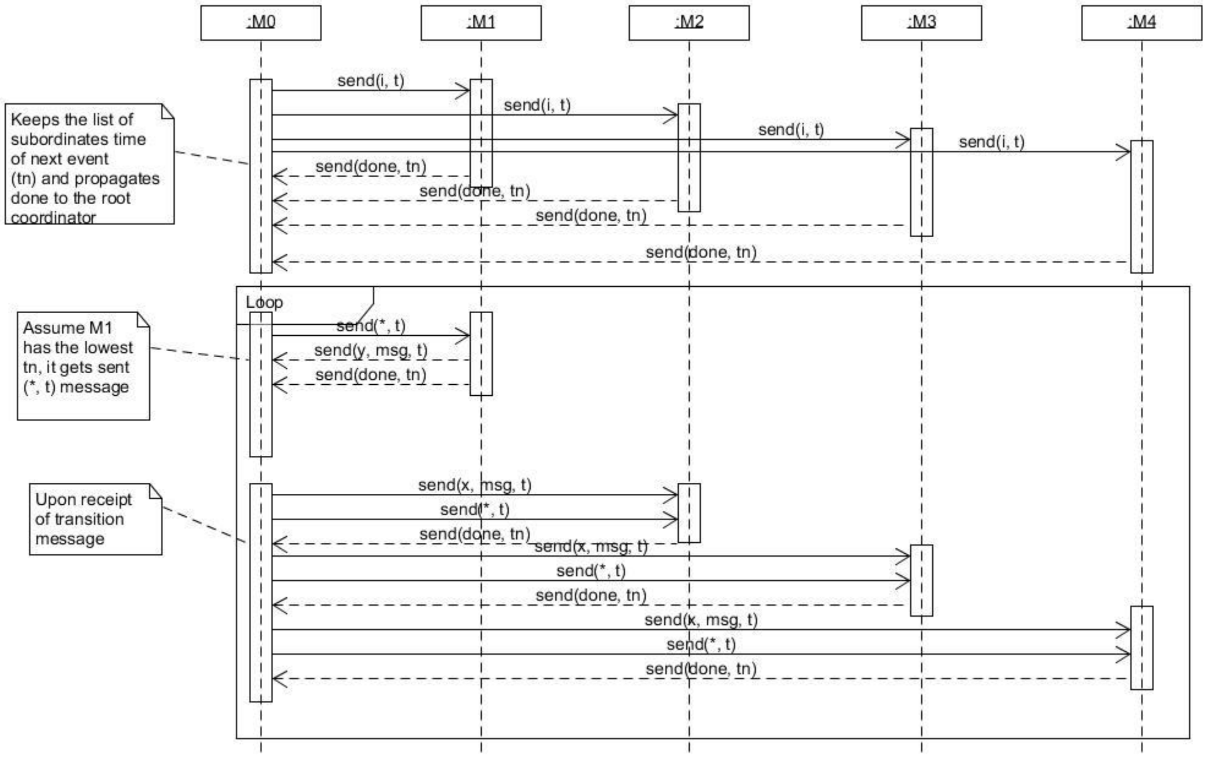


Figure 18: Sequence diagram for Zeigler's original algorithm

CHAPTER 4

4 PROPOSAL

This chapter provides the proposed methodology of various implementations that has been built and the rationale behind their creation. We also discuss details about the simulation engine that drives the execution and hierarchical construction of these models since separation of concerns between the simulation engine and models constructed is natural to DEVS.

4.1 SIMSTUDIO

SimStudio[13] is an operational framework for building a community-based environment to serve as a virtual laboratory for the study and experimentation of advanced concepts in modeling and simulation by the integration and exploration of tools (for example modeling, code generation, simulator, traces visualization plug-ins, etc.). It has a multi-layer structure with each layer serving as a container for modeling and simulation tools and integrates perfectly with the tools in the neighboring layer(s) by means of some MDA techniques to support or complement their outputs and/or activities[27]. Figure 19 shows a description of the architecture.

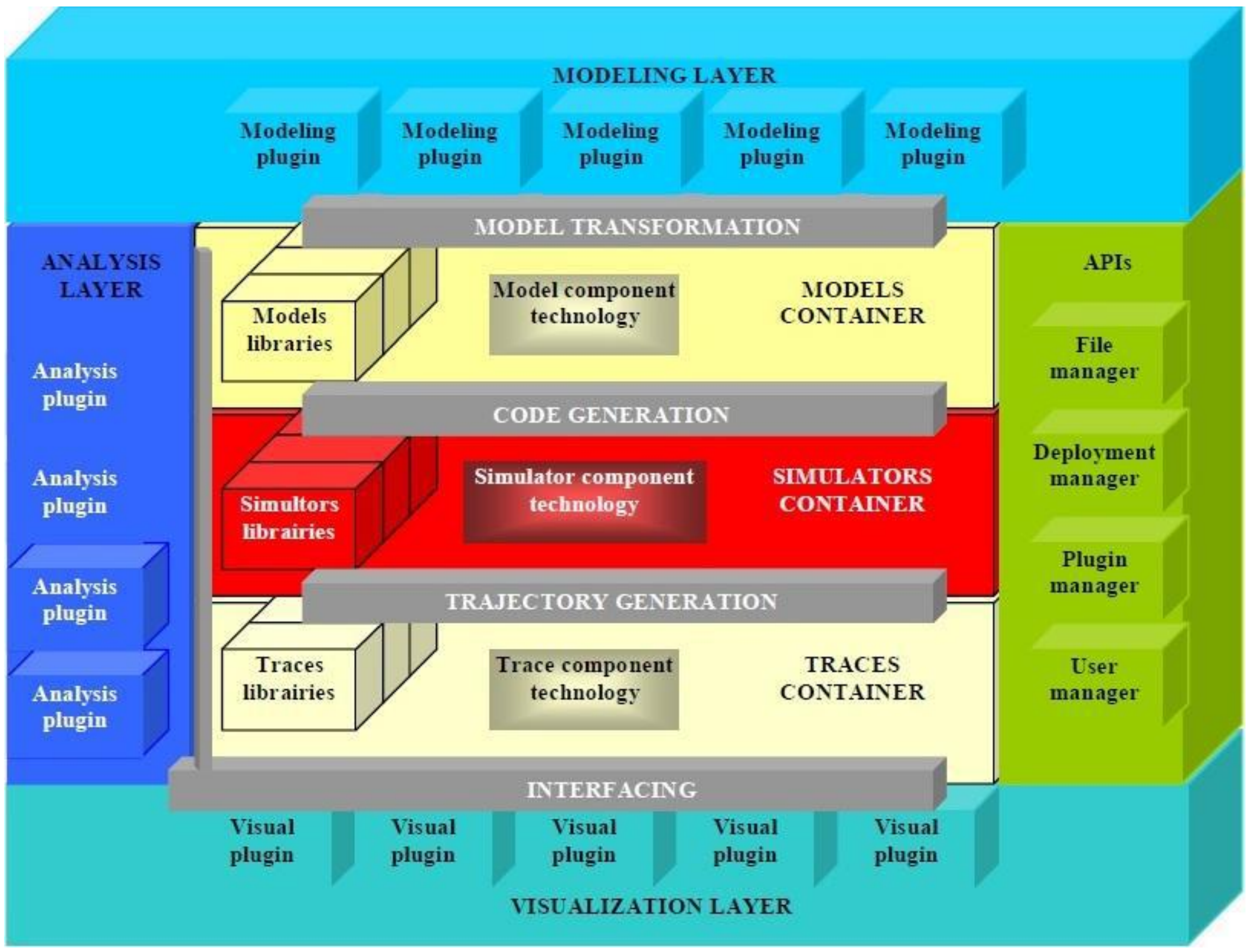


Figure 19: SimStudio framework architecture for modeling and simulation

The code generation layer require different simulator libraries to transform a model in its persistent form to the implementation code appropriate for a chosen simulator. We will implement simulator libraries that the framework can use to realize code generation. We also hope to provide different implementations to identify possible affinity in terms of performance of different models to different implementations which will be belabored upon subsequently.

4.2 DOMAIN OF ALGORITHMS

We propose a classification of PDEVS algorithms based on a global holistic view of the category each algorithm in literature falls under and the possible implementation approaches. All identified algorithms with a difference in the concept of approach is classified based on different implementation approaches possible which we believe represent the basis of all known implementations till date. For example, the algorithms proposed for testing in the JAMES toolkit by Jan Himmelspach and Adelinde M. Uhrmacher[22] falls under the Object-oriented programming (OOP) approach for abstract sequential algorithm, process based approach for the abstract threaded simulator variant and hybrid approach for flat sequential simulator; all of the Ziegler's algorithm.

We consider the possibility of implementing similar approaches for other algorithms such as the Chow's algorithm and Schwatinski variant. Table 1 illustrates the global view we came up with to classify all PDEVS algorithms.

Table 1 Domain categories of PDEVS algorithms

Algorithm	*OOP based*	*Process based*	*Hybrid*
Ziegler & Chow's	V	V	X
Chow's	X	X	X
Schwatinski & Pawletta	X	X	X
Ziegler	X	X	X

4.3 Specification of PDEVS algorithms implemented

At the moment we provide the specific implementation for the Ziegler and Chow's algorithm using the OOP and the process based approach. The implementations will be described using some of the basic UML notation[28] diagrams. The class diagram will provide structural overview of the implementation and the sequence diagram will illustrate behavioral flow of messages and proceedings of the implementation. We will sometimes use the state diagram also, to further show the behavior of a less clear implementation.

4.3.1 Message Passing

All Implementations uses the message passing scheme as illustrated in the Ziegler and Chow's algorithm. The class Diagram is depicted in Figure 20. The use of messages also introduces the possibility of synchronization error in dealing with them, so figure 21 shows the exceptions class diagram also.

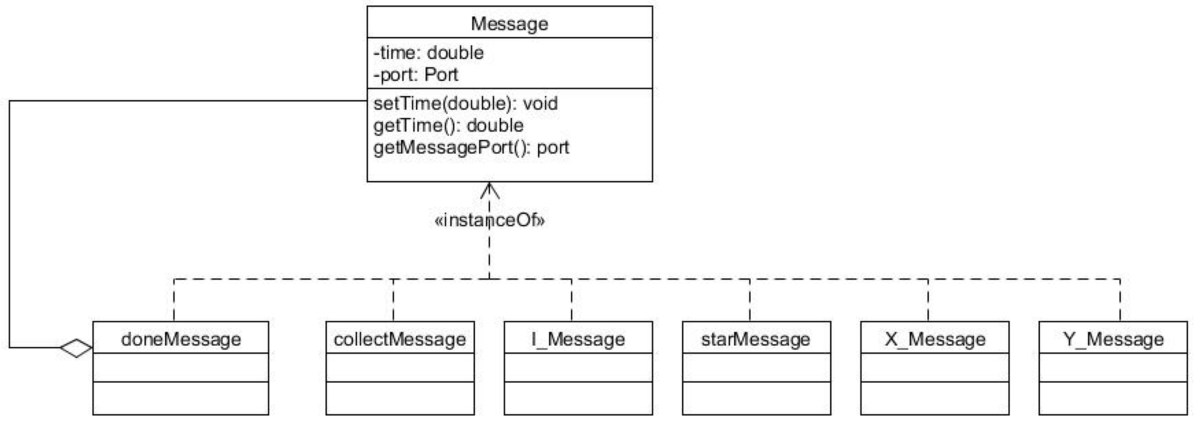


Figure 20: Class Diagram of Message passing scheme

The Invalidcoupling exception informs the modeler of possible error in connectin of models.

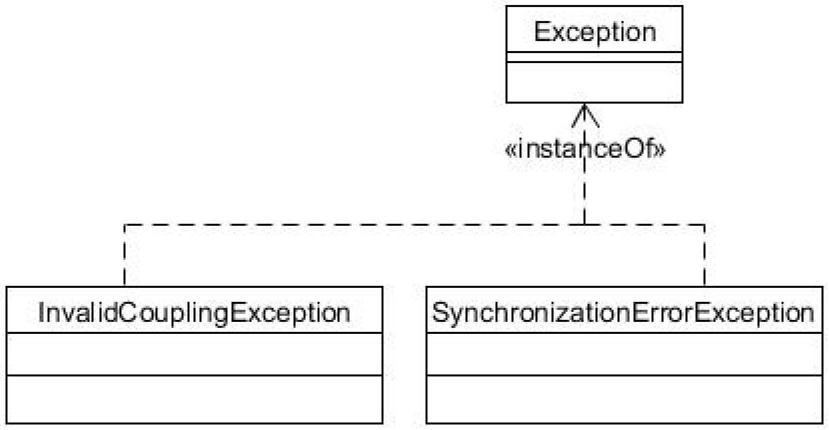


Figure 21: Class diagram for the exceptions hierarchy

4.3.2 OOP based Approach Specification

The class diagram for this implementation is shown in Figure 22. We will briefly explain the roles of the different classes in the implementation. We then also show the flow of messages and the behavior of the simulation using the sequence diagram in figure 23.

AbstractSimulator

This is the base simulation kernel that handles simulation controls. It is the superclass all DEVS processors inherit from, and it provides basic functionalities of the processor such as getting the time of events and the associated model it processes.

AtomicModel

This is the superclass for every atomic DEVS model. It Implements the basic control over the model to be provided by the modeler.

Bag

The collection for ports that extends the capability of models; Instead of receiving a single input or sending a single output, the models (atomic or coupled) can handle bags of ports which can be input or output.

BasicModel

This is the abstract base class for all user defined DEVS components, it provides basic functionalities such as getting the model name, creating and deleting ports, putting and removing values in ports common to all DEVS models in the user defined specification. It implicitly stores all created ports in either Inport Bag or Outport Bag based on the impetus of the modeler.

Coordinator

This processes a single CoupledModel unit and provides functionalities for passing messages to other DEVS components based on event list sorted based on time. The OOP approaches relies on using known components to execute their methods while the process based approach sends messages to the queue of the known components.

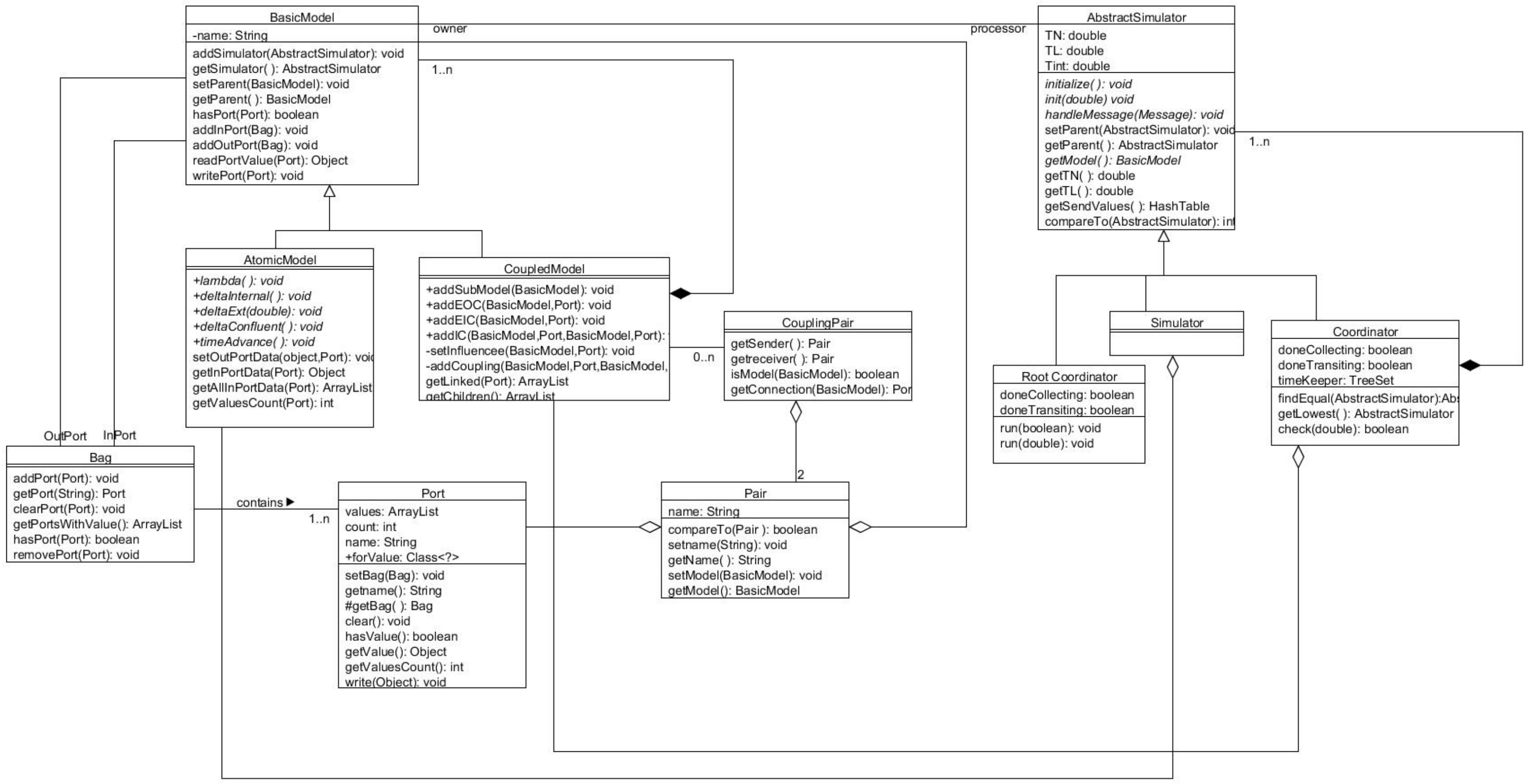


Figure 22: Class Diagram for the OOP based Simulation kernel for Ziegler and Chow's PDEVS Algorithm

CoupledModel

The superclass for every Coupled DEVS model. It Implements the basic component connections as specified by the modeler.

CouplingPair

This describes connection of pairs of model and port to realize a connected DEVS components. It provides functionalities of getting the sender and receiver in the connection. The Coordinator uses it to manage handling of messages received from subordinate components.

Pair

This is the bundle of a model and a specific port in its bag. Two pairs realizes a CouplingPair connection.

Port

The communication framework of models. This has been implemented as a list of message that will be delivered to the receiving model.

RootCoordinator

Directs and advances the simulation mechanism until the end.

Since all processors inherit from the AbstractSimulator class and it defines handleMessage method as a way of dealing with received messages. The initialize method is also implemented to establish all the processor hierarchy before the start of simulation.

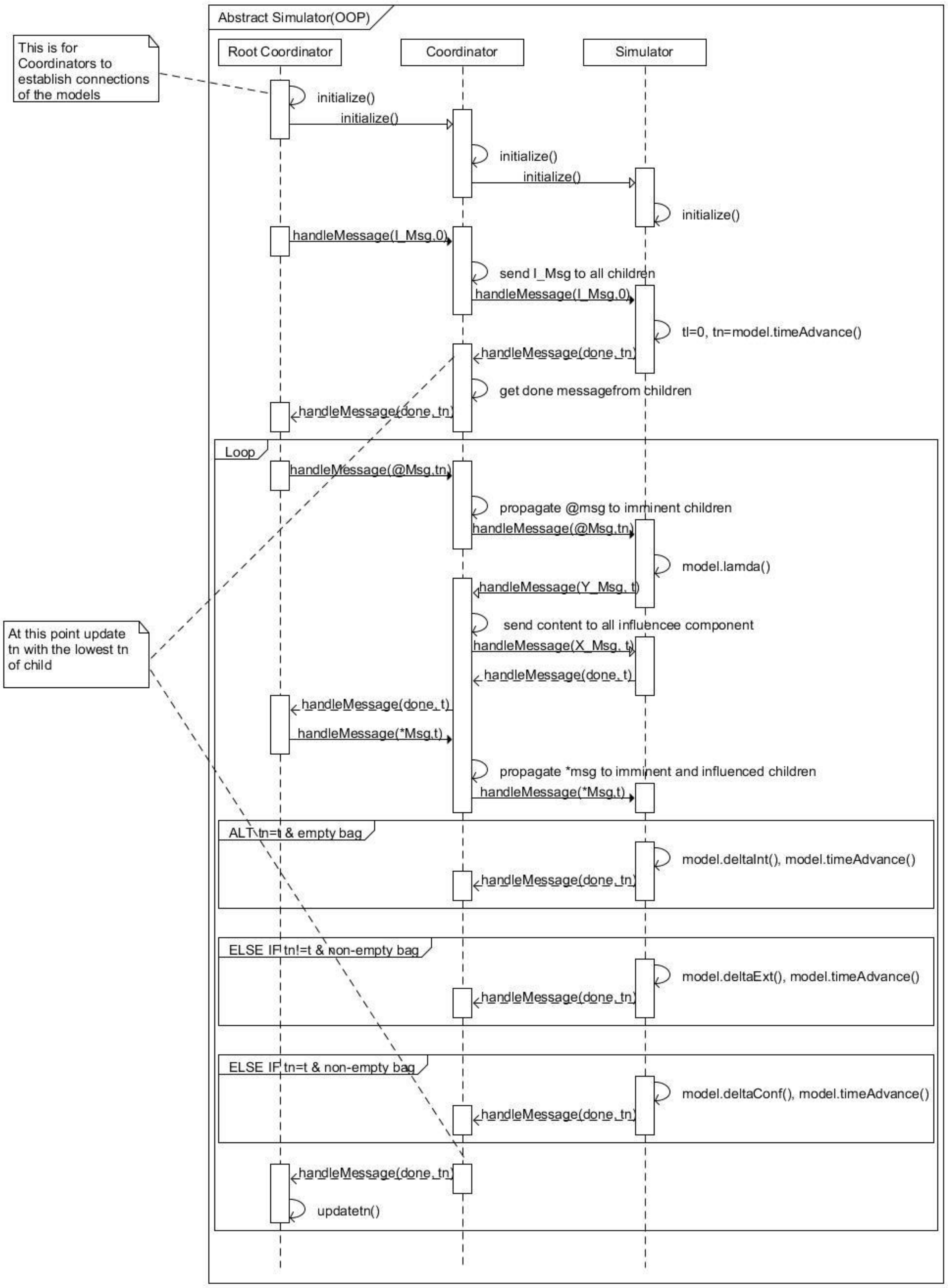


Figure 23: Sequence Diagram for OOP based Implementation

4.3.3 Process Based Approach Specification

The class diagram for this implementation is shown in Figure 24. This differs from the OOP approach in that threads are used by each DEVS processor to specify tasks that are executed. Processors implements runnable interface and possesses a thread-safe queue where messages a cached asynchronously, whereby the thread of the processor keeps polling for new messages and treats it, until the end of simulation. The state diagram illustrates this in figure 25. We then also show the flow of messages and the behavior of the simulation using the sequence diagram in figure 26.The class that is particular to this implementation is also briefly discussed.

Locker

While threads communicate using the queue of messages, the Locker is used to synchronize the expected number of done messages to be processed before signaling completion of the collection phase during the simulation.

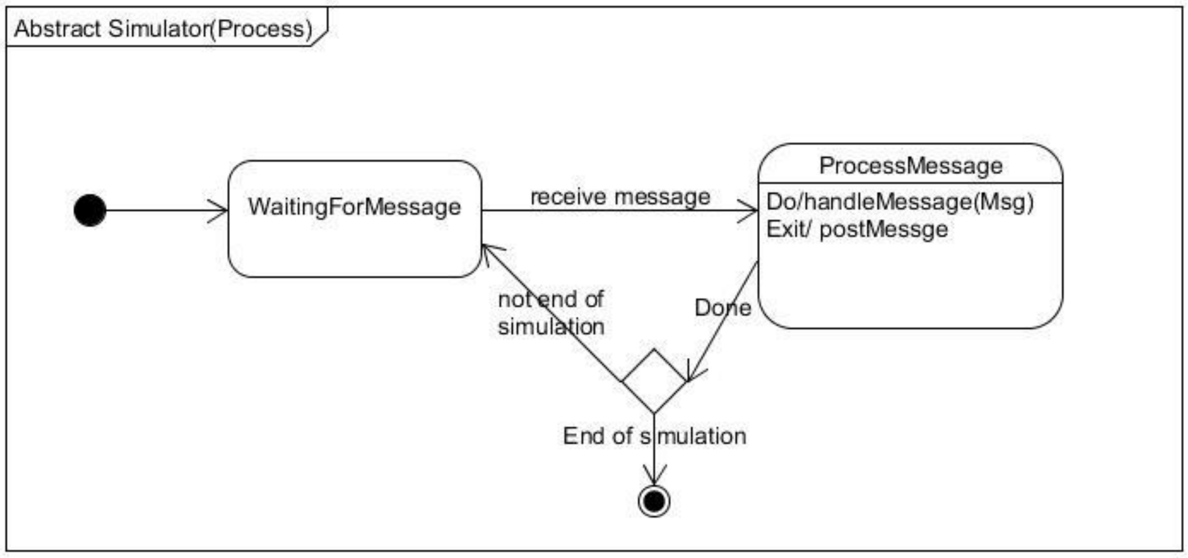


Figure 24: State Diagram for process based PDEVS algorithm Implementation

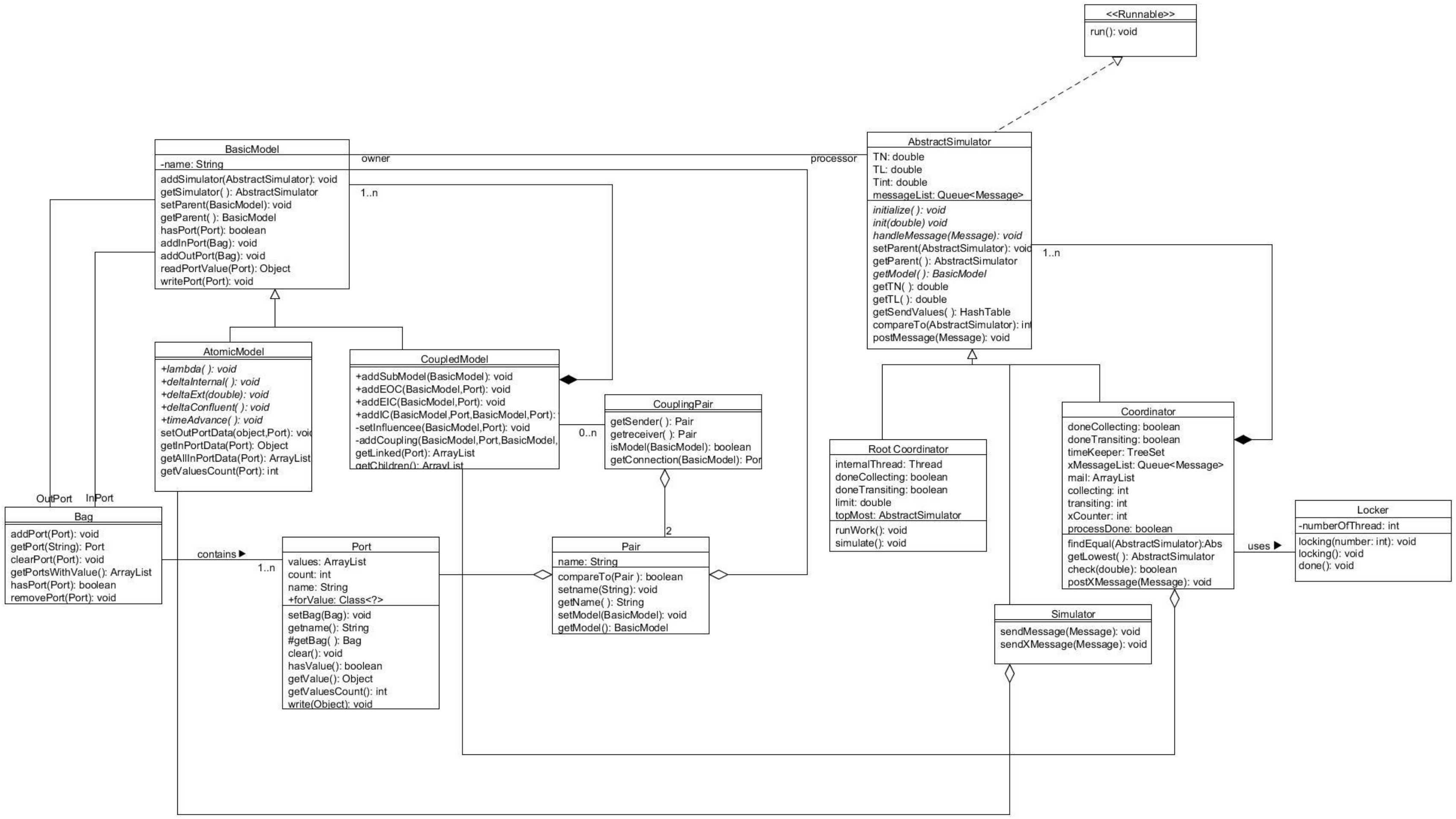


Figure 25: Class Diagram of Process based PDEVS algorithm Implementation

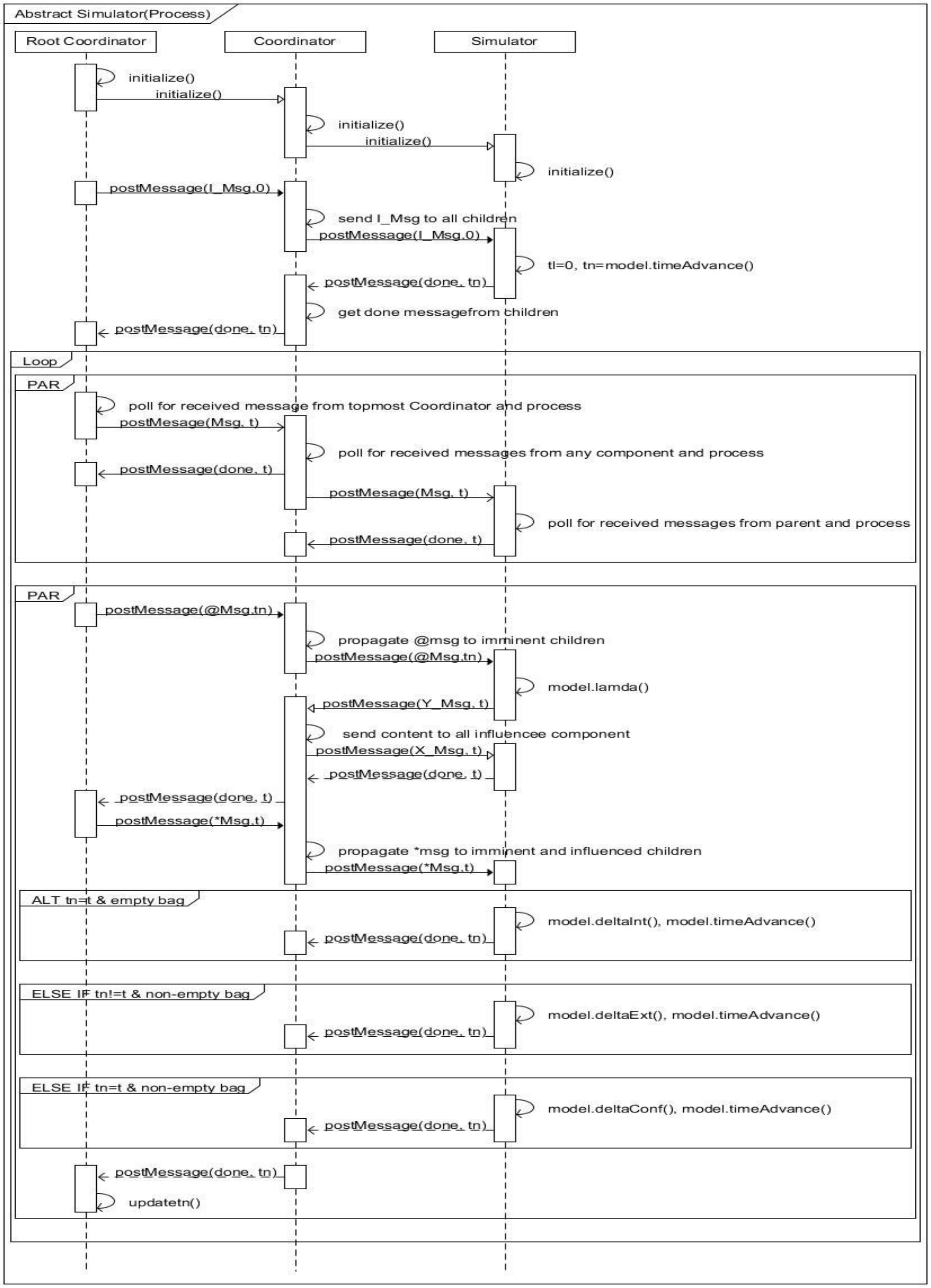

Abstract Simulator(Process)
Root Coordinator
Coordinator
Simulator
initialize()
initialize()
initialize()
initialize()
initialize()
postMessage(I_Msg,0)
send I_Msg to all children
postMessage(I_Msg,0)
tl=0, tn=model.timeAdvance()
postMessage(done, tn)
get done messagefrom children
postMessage(done, tn)
Loop
PAR
poll for received message from topmost Coordinator and process
postMesage(Msg, t)
poll for received messages from any component and process
postMessage(done, t)
postMesage(Msg, t)
poll for received messages from parent and process
postMessage(done, t)
PAR
postMessage(@Msg,tn)
propagate @msg to imminent children
postMessage(@Msg,tn)
model.lamda()
postMessage(Y_Msg, t)
send content to all influencee component
postMessage(X_Msg, t)
postMessage(done, t)
postMessage(done, t)
postMessage(*Msg,t)
propagate *msg to imminent and influenced children
postMessage(*Msg,t)
ALT tn=t & empty bag
model.deltaInt(), model.timeAdvance()
postMessage(done, tn)
ELSE IF tn!=t & non-empty bag
model.deltaExt(), model.timeAdvance()
postMessage(done, tn)
ELSE IF tn=t & non-empty bag
model.deltaConf(), model.timeAdvance()
postMessage(done, tn)
postMessage(done, tn)
updatetn()

Figure 26: Sequence Diagram for the Process based PDEVS algorithm Implementation

CHAPTER 5

5 APPLICATION / CASE STUDY

5.1 CASE STUDY OF CROSSROAD IN A ROAD NETWORK

The road network is a versatile part of the human transportation system which contains crossroads where vehicle path intersects, if not properly managed, chaos ensues, and the possible loss of life due to potential accidents are unacceptable. However, even when managed, this junctions easily lead to bottlenecks and traffic jams. Understanding the behavior of this system can provide a basis of planning to improve the network, but it is inefficient to do in real time, since a very long time, that may run into years is required for proper study. Simulation of the system can also be used as an alternative, hence we provided the simulation of the system using the PDEVS formalism.

5.2 DEVS SPECIFICATION OF THE ROAD NETWORK

We focus on a crossroad of two road lanes, each possessing a traffic light which controls the passage of cars. The traffic light of the first lane influences the traffic light of the other lane to ensure only one lane is passed at any point in time. The state diagram of the traffic lights is shown in figure 27.

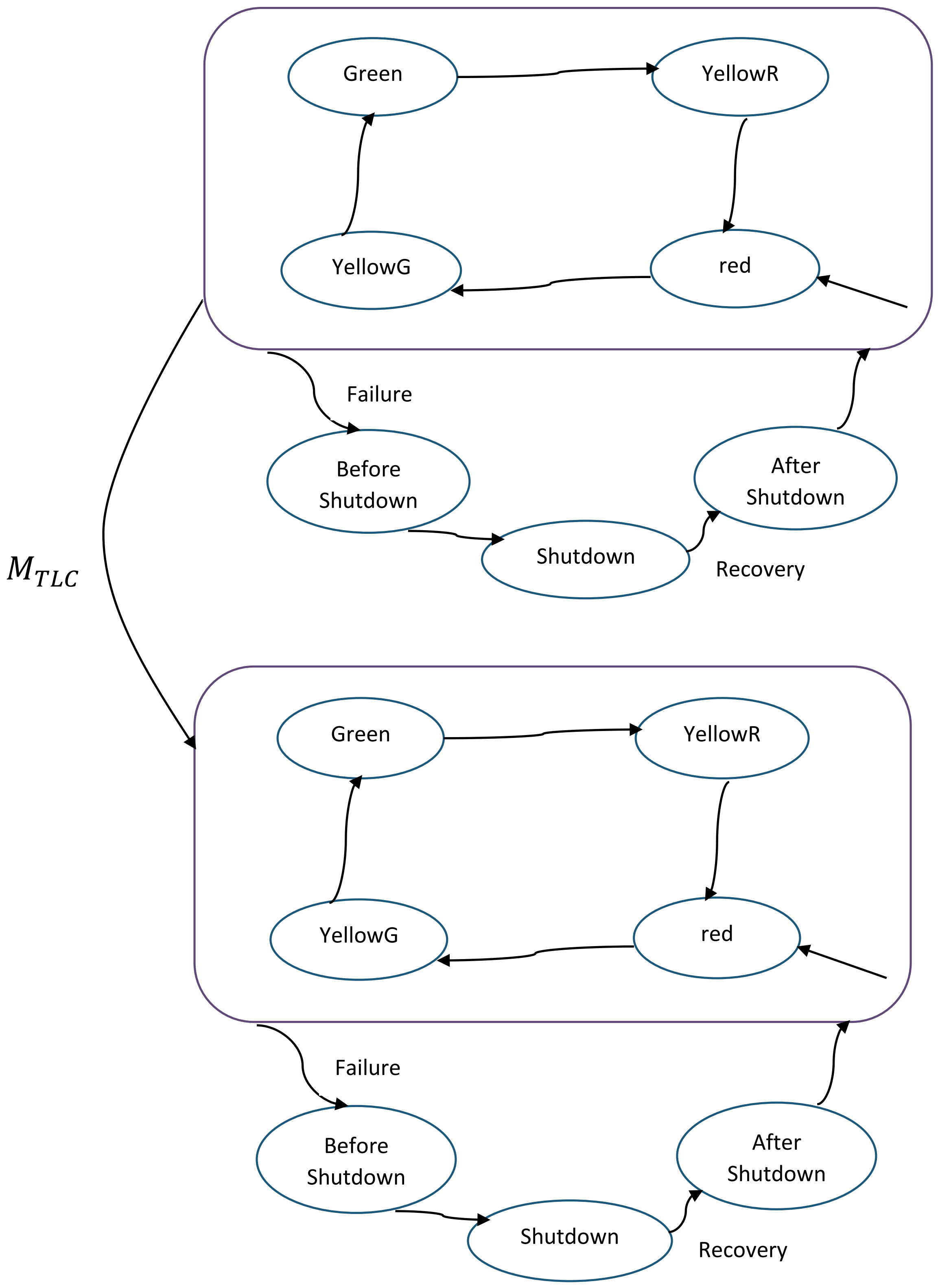


Figure 27: State Diagram of the Traffic Lights Specification

M_{TLR}

$$M_{TLC} = < X^b, Y^b, S, \delta_{int}, \delta_{ext}, \lambda, \Gamma a, \delta_{conf} >$$

$$X^b = \{Failure, Recovery\}$$

$$Y^b = \{green, yellow, red, black\}$$

$$S = \{R, Yr, G, Yg, Bs, Sd, As\}$$

Let,

$s \in S \; and$

$e \in [0, \Gamma a(s)]$

$\boldsymbol{\Gamma a : S \rightarrow \mathbb{R}_{+}^{0,\infty}}$	$\boldsymbol{\lambda : S \rightarrow Y}$	$\boldsymbol{\delta_{int} : S \rightarrow S}$	$\boldsymbol{\delta_{ext} : Qxx \rightarrow S}$	$\boldsymbol{\delta_{conf} : Qx^b \rightarrow S}$
$\boldsymbol{G \rightarrow 8.0}$	$G \rightarrow yellow$	$\mathrm{G} \rightarrow Yr$	$(s, e, Failure) \rightarrow Bs$	
$\boldsymbol{Yr \rightarrow 2.0}$	$Yr \rightarrow red$	$Yr \rightarrow R$	$(s, e, Failure) \rightarrow As$	
$\boldsymbol{R \rightarrow 5.0}$	$R \rightarrow yellow$	$R \rightarrow Yg$		
$\boldsymbol{Yg \rightarrow 2.0}$	$Yg \rightarrow green$	$Yg \rightarrow G$		
$\boldsymbol{Sd \rightarrow \infty}$	$Bs \rightarrow black$	$Bs \rightarrow Sd$		
$\boldsymbol{Bs \rightarrow 0.0}$	$As \rightarrow red$	$As \rightarrow R$		

$\boldsymbol{As \rightarrow\ 0.0}$				

$$M_{TLR}\ \ = < X^b, Y^b, S, \delta_{int}, \delta_{ext}, \lambda, \Gamma a, \delta_{conf}\ \ >$$

$$X^b\ = \{Failure, Recovery\} \times\ \{green, yellowR, yellowG, red, black\}$$

$$Y^b\ = \{green, yellow, red, black\}$$

$$S = \{R, Yr, G, Yg, Bs, Sd, As, hold\}$$

Let,

$y\ \in\ Y^b$

$s\ \in S\ and$

$e\ \in [0, \Gamma a(s)$

$\boldsymbol{\Gamma a}: \boldsymbol{S} \rightarrow \mathbb{R}_{+}^{\mathbf{0},\infty}$	$\boldsymbol{\lambda : S \rightarrow Y}$	$\boldsymbol{\delta_{int} : S \rightarrow S}$	$\boldsymbol{\delta_{ext} : Qxx \rightarrow S}$	$\boldsymbol{\delta_{conf} : Qx^b \rightarrow S}$
$\boldsymbol{G \rightarrow\ 0.0}$	$G \rightarrow\ yellow$	$\mathrm{G} \rightarrow hold$	$(s, e, Failure) \rightarrow Bs$	$(s, Failure, y) \rightarrow Bs$
$\boldsymbol{Yr \rightarrow\ 0.0}$	$Yr \rightarrow\ red$	$Yr \rightarrow hold$	$(s, e, Recovery) \rightarrow As$	$(s, Recovery, green) \rightarrow Yr$
$\boldsymbol{R \rightarrow\ 0.0}$	$R \rightarrow\ yellow$	$R \rightarrow hold$		$(s, Recovery, yellowR) \rightarrow R$
$\boldsymbol{Yg \rightarrow\ 0.0}$	$Yg \rightarrow\ green$	$Yg \rightarrow hold$		$(s, Recovery, yellowG) \rightarrow G$
$\boldsymbol{Sd \rightarrow\ \infty}$	$Bs \rightarrow\ black$	$Bs \rightarrow Sd$		$(s, Recovery, red) \rightarrow Yg$

$\boldsymbol{Bs \rightarrow\ 0.0}$	$As \rightarrow\ red$	$As \rightarrow hold$		$(s, Recovery, black) \rightarrow Yr$
$\boldsymbol{As \rightarrow\ 0.0}$				
$\boldsymbol{hold \rightarrow\ \infty}$				

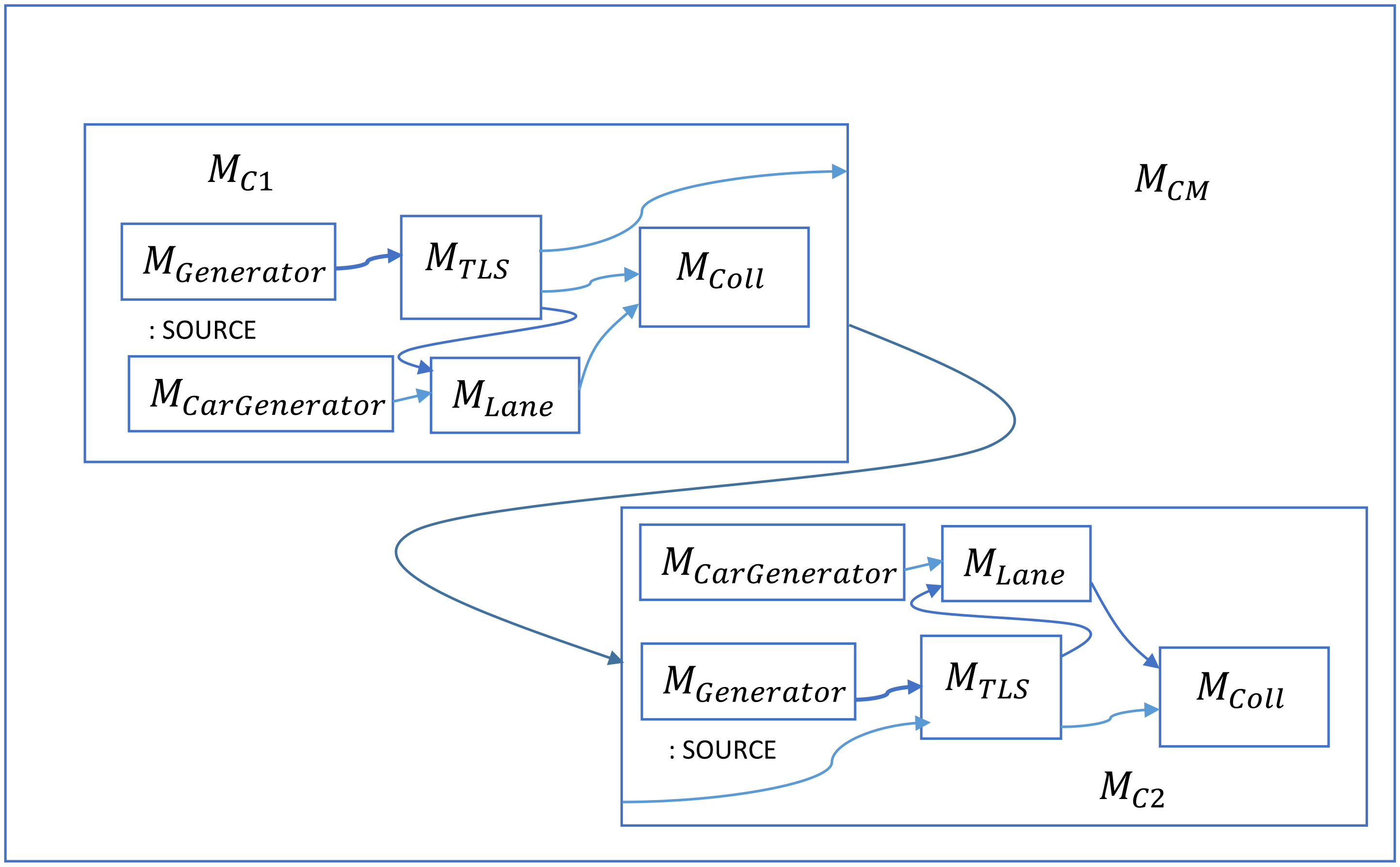


Figure 28: Coupled Model Diagram of the Crossroad Model

$$M_{CarGenerator} = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \Gamma a$$

$$X = \emptyset$$

$$Y = \{0,1\}$$

$$S = \{gen\}$$

$\boldsymbol{\Gamma a: S \rightarrow \mathbb{R}_{+}^{0,\infty}}$	$\boldsymbol{\lambda : S \rightarrow Y}$	$\boldsymbol{\delta_{int} : S \rightarrow S}$	$\boldsymbol{\delta_{ext} : Qxx \rightarrow S}$
$\boldsymbol{gen} \rightarrow [\boldsymbol{2,6}]$	$gen \rightarrow y$	$\boldsymbol{gen} \rightarrow gen$	

$where, y \in Y$

$$M_{Lane} = < X, Y, S, \delta_{int}, \delta_{ext}, \lambda, \Gamma a >$$

$$X = \{Car, NoCar, Go, Stop\}$$

$$Y = \{RoadState\}$$

$$S = \{A \mid A \in (a_1, a_2, a_3, a_4, a_5)\} \times \{Go, Stop\} \times \mathbb{R}_{+}^{\mathbf{0},\infty}$$

$$where, a_i = [0,1] \; for \; i = 0..5$$

$\boldsymbol{\Gamma a: S \rightarrow \mathbb{R}_{+}^{0,\infty}}$	$\boldsymbol{\lambda : S \rightarrow Y}$	$\boldsymbol{\delta_{int} : S \rightarrow S}$	$\boldsymbol{\delta_{ext} : Qxx \rightarrow S}$
$(\boldsymbol{A}, \boldsymbol{tr}, \boldsymbol{\sigma}) \rightarrow \boldsymbol{\sigma}$	$(A, tr, \sigma) \rightarrow A$	$(A, Go, \sigma) \rightarrow ((a_0, a_1, a_2, a_3, a_4), tr, [3,5])$	$(\boldsymbol{s}, e, Go) \rightarrow (A, Go)$
		$(A, Stop, \sigma) \rightarrow (A_{new}, tr, [3,5])$	$(\boldsymbol{s}, e, Stop) \rightarrow (A, Stop)$
			$(\boldsymbol{s}, e, Car) \rightarrow (A_c, tr)$
			$(\boldsymbol{s}, e, noCar) \rightarrow (A, tr)$

$where, s \in S$

$\sigma \in \mathbb{R}_{+}^{0,\infty}$,

$e \in [0, \Gamma a(s)]$

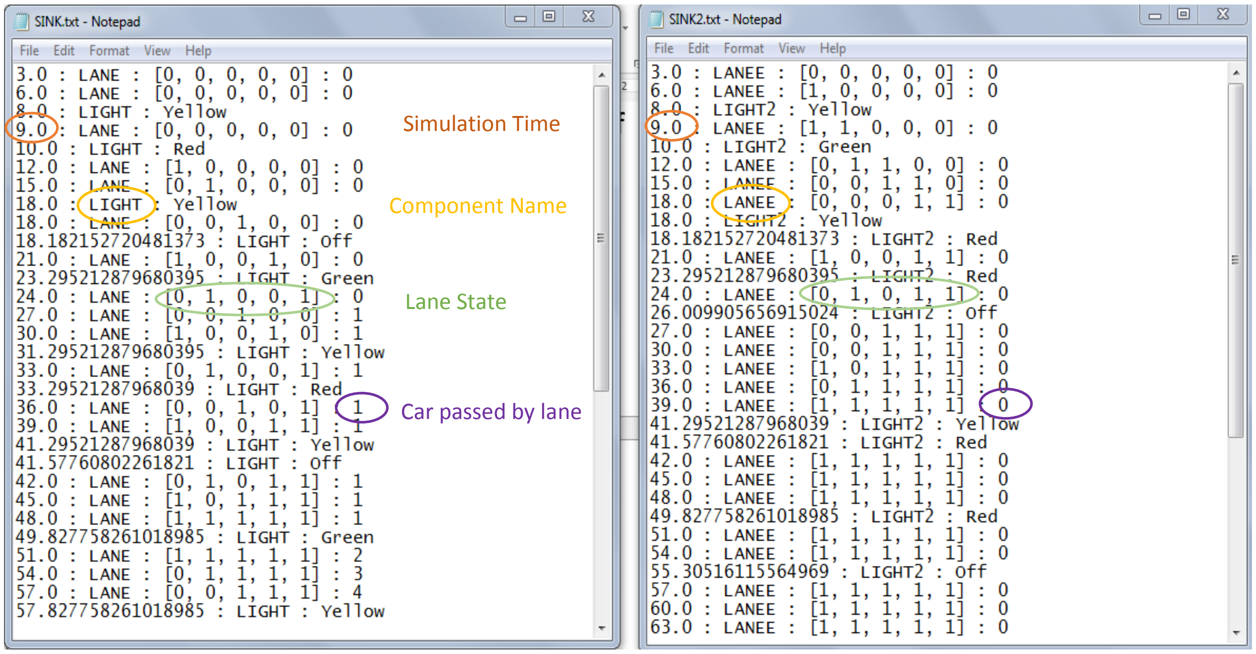
$tr = [Go, Stop]$

$A_{new} \in S \mid \forall \; a_i, \; a_i = a_{i-1}$ $and \; then \; a_{i-1} = 0$ if a_i is 0

$$A_c = A \mid a_1 = 1 \text{ If } a_1 \text{ is } 0$$

RESULTS:

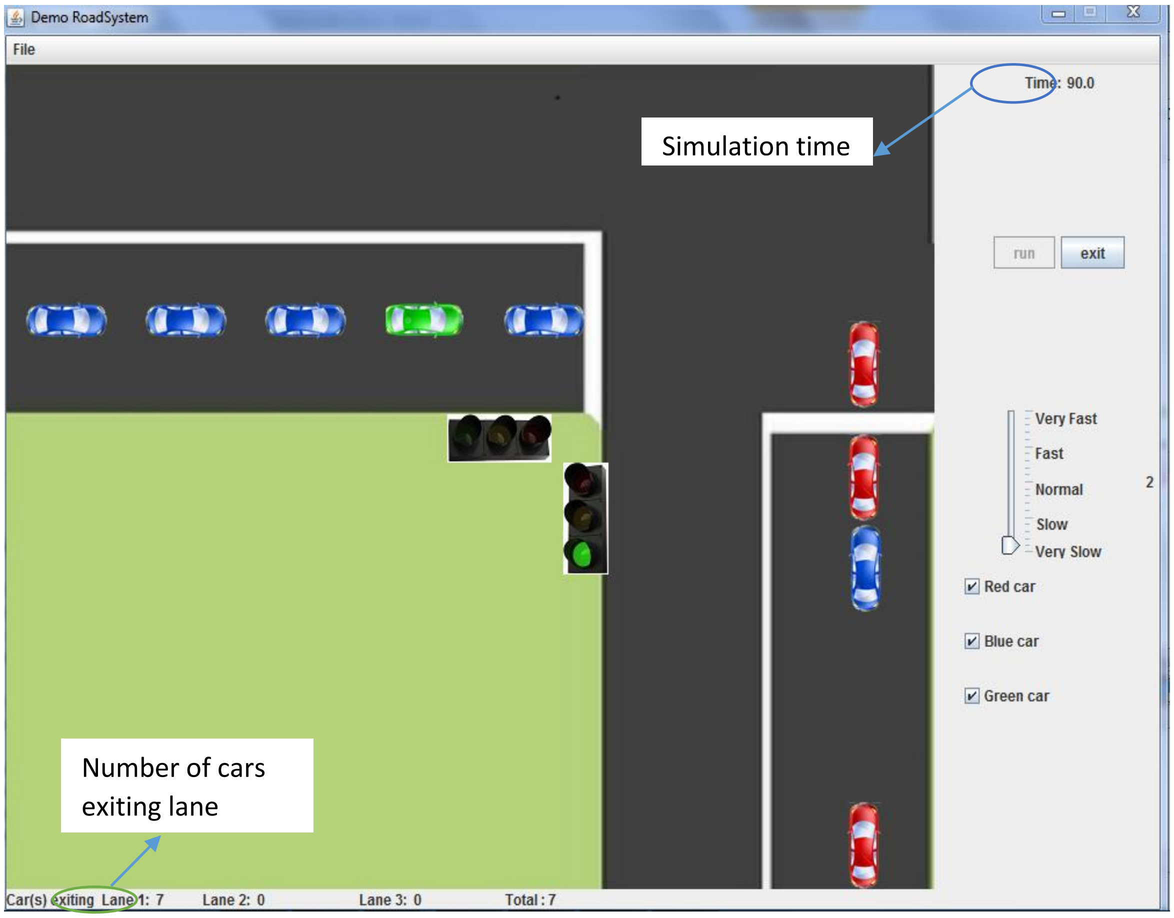
The System outputs the trace of a simulation in text file created by the collector atomic model with the format as illustrated below.



```
3.0 : LANE : [0, 0, 0, 0, 0] : 0
6.0 : LANE : [0, 0, 0, 0, 0] : 0
8.0 : LIGHT : Yellow
9.0 : LANE : [0, 0, 0, 0, 0] : 0
10.0 : LIGHT : Red
12.0 : LANE : [1, 0, 0, 0, 0] : 0
15.0 : LANE : [0, 1, 0, 0, 0] : 0
18.0 : LIGHT : Yellow
18.0 : LANE : [0, 0, 1, 0, 0] : 0
18.182152720481373 : LIGHT : Off
21.0 : LANE : [1, 0, 0, 1, 0] : 0
23.295212879680395 : LIGHT : Green
24.0 : LANE : [0, 1, 0, 0, 1] : 0
27.0 : LANE : [0, 0, 1, 0, 0] : 1
30.0 : LANE : [1, 0, 0, 1, 0] : 1
31.295212879680395 : LIGHT : Yellow
33.0 : LANE : [0, 1, 0, 0, 1] : 1
33.2952128796803 9 : LIGHT : Red
36.0 : LANE : [0, 0, 1, 0, 1] : 1
39.0 : LANE : [1, 0, 0, 1, 1] : 1
41.2952128796803 9 : LIGHT : Yellow
41.5776080226182 1 : LIGHT : Off
42.0 : LANE : [0, 1, 0, 1, 1] : 1
45.0 : LANE : [1, 0, 1, 1, 1] : 1
48.0 : LANE : [1, 1, 1, 1, 1] : 1
49.827758261018985 : LIGHT : Green
51.0 : LANE : [1, 1, 1, 1, 1] : 2
54.0 : LANE : [0, 1, 1, 1, 1] : 3
57.0 : LANE : [0, 0, 1, 1, 1] : 4
57.827758261018985 : LIGHT : Yellow
```

```
3.0 : LANEE : [0, 0, 0, 0, 0] : 0
6.0 : LANEE : [1, 0, 0, 0, 0] : 0
8.0 : LIGHT2 : Yellow
9.0 : LANEE : [1, 1, 0, 0, 0] : 0
10.0 : LIGHT2 : Green
12.0 : LANEE : [0, 1, 1, 0, 0] : 0
15.0 : LANEE : [0, 0, 1, 1, 0] : 0
18.0 : LANEE : [0, 0, 0, 1, 1] : 0
18.0 : LIGHT2 : Yellow
18.182152720481373 : LIGHT2 : Red
21.0 : LANEE : [1, 0, 0, 1, 1] : 0
23.295212879680395 : LIGHT2 : Red
24.0 : LANEE : [0, 1, 0, 1, 1] : 0
26.009905656915024 : LIGHT2 : Off
27.0 : LANEE : [0, 0, 1, 1, 1] : 0
30.0 : LANEE : [0, 0, 1, 1, 1] : 0
33.0 : LANEE : [1, 0, 1, 1, 1] : 0
36.0 : LANEE : [0, 1, 1, 1, 1] : 0
39.0 : LANEE : [1, 1, 1, 1, 1] : 0
41.2952128796803 9 : LIGHT2 : Yellow
41.5776080226182 1 : LIGHT2 : Red
42.0 : LANEE : [1, 1, 1, 1, 1] : 0
45.0 : LANEE : [1, 1, 1, 1, 1] : 0
48.0 : LANEE : [1, 1, 1, 1, 1] : 0
49.827758261018985 : LIGHT2 : Red
51.0 : LANEE : [1, 1, 1, 1, 1] : 0
54.0 : LANEE : [1, 1, 1, 1, 1] : 0
55.30516115564969 : LIGHT2 : Off
57.0 : LANEE : [1, 1, 1, 1, 1] : 0
60.0 : LANEE : [1, 1, 1, 1, 1] : 0
63.0 : LANEE : [1, 1, 1, 1, 1] : 0
```

Figure 29: Trajectory of Simulation of the road network using PDEVS



Car(s) exiting Lane 1: 7 Lane 2: 0 Lane 3: 0 Total : 7

Figure 30: Visualization of the Road Network System Simulation

CHAPTER 6

6 BENCHMARKING AND ANALYSIS

This chapter details some experiments to test the performance of implemented prototypes. Basic experiment of transferring messages between varying amounts of atomic models will be used to benchmark how long simulations takes in real time as the simulation end time increases, we also look at how the depth of the model affects the simulation. We go on to benchmark with a simple version of the traffic light system explained in the chapter above.

6.1 EXPERIMENTAL SETUP

We compare real time spent as the depth increases for the two implementations using two atomic models passing message from a generator to collector, while increasing the depth (number of intermediate coupled models) for a simulation period of 1000 time unit and constant time advance of 2 units. The result plot is shown in figure 30.

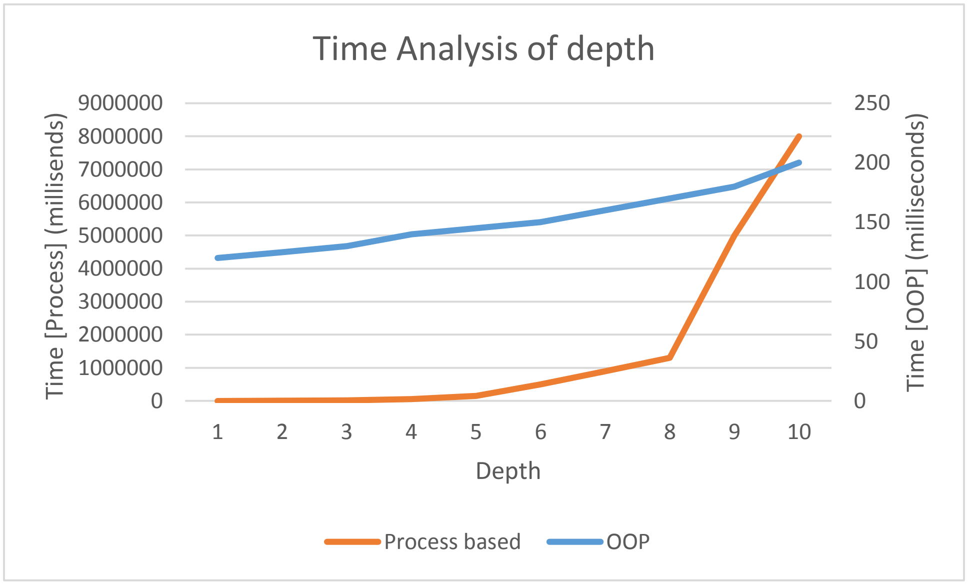


Figure 31: Depth plot of Implementations

The plot shows that the OOP implementation's time spent rises steadily as the depth increases but the process based time spent rises in a near exponential manner. This is due to increase in inter-thread communication overhead majorly and can only be worthy of practical uses when the model is sufficiently large enough and deployed on multiple processor in our humble opinion; since the OOP variant is incapable of utilizing multiple cores and as the model size increases, so does the real time spent on the simulation.

We also look at the time spent as the traffic light model simulation period is increased from a 100 to 1000 to illustrate constant depth, practical model analysis of the implementation. Figure 31 shows the plot result.

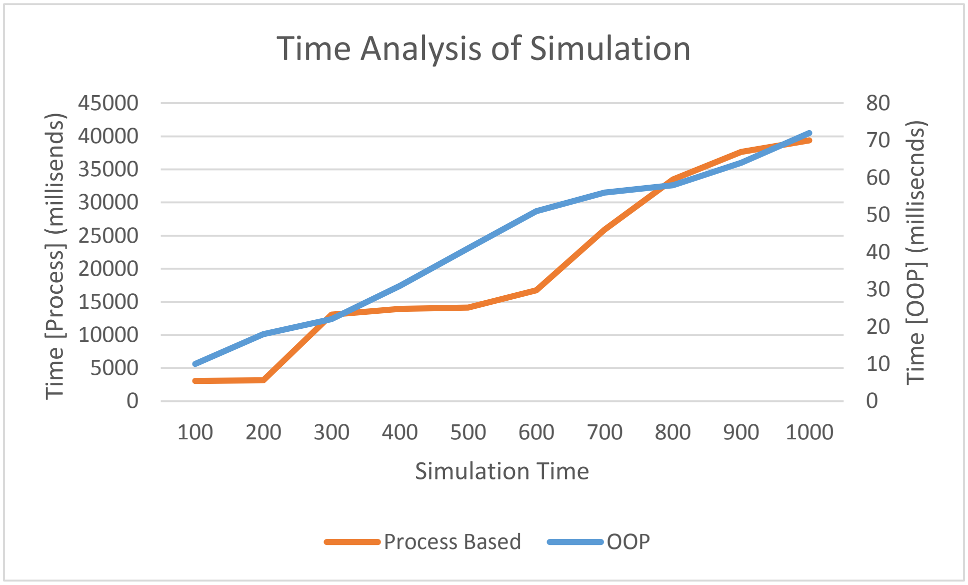


Figure 32: Simulation time plot of the Implementations

The simulation time plot shows a steady increase in real time spent on simulation for both implementations, just that, the starting point of real time spent, and the factor of increase per unit increase in simulation time for the process oriented implementation is quite greater than that of the OOP based implementation.

CHAPTER 7

7 CONCLUSION

This chapter draws some general conclusions and describes the outcomes of this work. The basic facts are summarized, and we describe possible future works.

7.1 PERSPECTIVE

In this thesis, we motivated the development of sequential PDEVS algorithms using different software engineering approaches in a bid to identify possible affinity or compatibility between simulation models and PDEVS algorithms and standardize the categories of all implementations of the algorithms. The context for development of multiple PDEVS algorithms and testing using appropriate benchmark model for theoretical analysis was motivated (Chapter 1). The different PDEVS algorithms found in literature are then analyzed using the UML sequence diagram on their execution of the benchmark model (Chapter 3).

We go on to implement the different categories of PDEVS algorithms; we started with Ziegler and Chow's PDEVS algorithm and completed the Object oriented approach and the process approach only; owing to time limitation. Chapter 4 details the implementations done. Finally, we provide experimental analysis using basic simulation models to investigate important parameters and how they affect the performance of simulation (Chapter 6).

7.2 CONTRIBUTIONS AND LIMITATIONS

While our exiguous implementations fails to completely analyze our proposed global view of implementing PDEVS algorithms. We do however, contribute to the functionality of the sim-studio by providing fully functional PDEVS simulators, and our proposal of PDEVS implementations classification still holds.

It is quite important to also note that our classification scheme has compressed a bevy of possible implementation variants in the hybrid such as OOP oriented flattened simulators and process oriented flattened simulators including any other un-wholesome method of developing simulators. This may be a case of over-generalization which will not clearly show possible varying behaviors of these different simulators.

7.3 Future works

In this thesis we have not completed the full scope of the implementations in the global scheme we proposed, even-though some have been implemented by other research groups. We hope implementation of the full scope is complemented with more elaborate test basis on which full analysis can be carried out.

Other future research includes:

- The development of a Graphics User Interface (GUI) oriented specifications of models and basic trajectories plot for the simulators. This would make it easier for the modeler to specify his models and analyze trajectories behavior.
- A tool can be dedicated to analyzing and comparing the performance of various DEVS implementations and can also include the verification and validation of the created models.
- Some more functionality could be added, mainly with distributed simulation. These additional features are not really that useful in the general sense due to their huge impact on performance. Examples of unsupported combinations are real-time distributed simulation and distributed dynamic structure DEVS.
- Provision of a more robust tracing mechanism for the implementations will also easy debugging for the modelers and provide better analysis of the simulation behavior.

8 REFERENCES

[1] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[2] B. P. Zeigler, *Theory of Modeling and Simulation, Wiley, N.Y.*, First Edition. 1976.

[3] W. C. Cave and R. E. Wassmer, "UNDERSTANDING INHERENT PARALLELISM." 2008.

[4] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: A parallel, hierarchical, modular, modeling formalism," in *Proceedings of the 26th conference on Winter simulation*, 1994, pp. 716–722.

[5] G. Wainer, "CD++: a toolkit to develop DEVS models," *Softw. Pract. Exp.*, vol. 32, no. 13, pp. 1261–1306, 2002.

[6] M. Bonaventura, G. A. Wainer, and R. Castro, "Advanced IDE for modeling and simulation of discrete event systems," in *Proceedings of the 2010 Spring Simulation Multiconference*, 2010, p. 125.

[7] J. Nutaro, "Adevs user manual and API documentation." [Online]. Available: http://web.ornl.gov/~1qn/adevs/adevs-docs/index.html. [Accessed: 28-Oct-2014].

[8] B. P. Zeigler, Y. Moon, D. Kim, and J. G. Kim, "DEVS-C++: A high performance modelling and simulation environment," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,* 1996, vol. 1, pp. 350–359.

[9] H. S. Sarjoughian and B. R. Zeigler, "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment," *Simul. Ser.*, vol. 30, pp. 29–36, 1998.

[10] S. Kim, H. S. Sarjoughian, and V. Elamvazhuthi, "DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring," in *Proceedings of the 2009 Spring Simulation Multiconference*, 2009, p. 161.

[11] H. S. Sarjoughian and V. Elamvazhuthi, "CoSMoS: a visual environment for component-based modeling, experimental design, and simulation," in *Proceedings of the 2nd international conference on simulation tools and techniques*, 2009, p. 59.

[12] M. H. Hwang, "DEVS#: C# Open Source Library of DEVS Formalism," *Modeling and Simulation using DEVS#.* [Online]. Available: http://xsy-csharp.sourceforge.net/DEVSsharp/. [Accessed: 29-Oct-2014].

[13] M. K. Traoré, "SimStudio: a next generation modeling and simulation framework," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, 2008, p. 67.

[14] T. G. Kim, "DEVSim++: C++ based simulation with hierarchical modular DEVS models," *User's Man. CORE Lab EE Dept KAIST Taejon Korea*, 1994.

[15] J. Himmelspach and A. M. Uhrmacher, “A component-based simulation layer for JAMES,” in *Proceedings of the eighteenth workshop on Parallel and distributed simulation*, 2004, pp. 115–122.

[16] J. B. Filippi, M. Delhom, and F. Bernardi, “The JDEVS modeling and simulation environment,” in *Integrated Assessment and Decision Support, Proceedings of the 1st Biennial Meeting of the International Environmental Modelling and Software Society*, 2002, vol. 3, pp. 283–288.

[17] J. De Lara and H. Vangheluwe, “AToM3: A Tool for Multi-formalism and Meta-modelling,” in *Fundamental approaches to software engineering*, Springer, 2002, pp. 174–188.

[18] H. Praehofer, J. Sametinger, and A. Stritzinger, “Concepts and architecture of a simulation framework based on the JavaBeans component model,” *Future Gener. Comput. Syst.*, vol. 17, no. 5, pp. 539–559, 2001.

[19] J. Dávila and M. Uzcágegui, “Galatea: A multi-agent simulation platform,” in *International Conference on Modeling, Simulation and Neural Networks MSNN*, 2000.

[20] V. Sanz, A. Urquia, and S. Dormido, “Parallel DEVS and process-oriented modeling in Modelica,” in *In Proceedings of the 7 th International Modelica Conference*, 2009.

[21] V. Janoušek and E. Kironský, “Exploratory modeling with SmallDEVS,” in *Proceedings of the 20th annual European Simulation and Modelling Conference*, 2006, pp. 122–126.

[22] J. Himmelspach and A. M. Uhrmacher, “Sequential processing of PDEVS models,” *Proc. 3rd EMSS*, pp. 239–244, 2006.

[23] H. S. Sarjoughian, F. E. Cellier, and B. P. Zeigler, *Discrete event modeling and simulation technologies: a tapestry of systems and AI-based theories and methodologies*. Springer, 2001.

[24] V. Sanz, A. Urquia, and S. Dormido, “Integrating Parallel DEVS and equation-based object-oriented modeling,” in *Proceedings of the 2010 Spring Simulation Multiconference*, 2010, p. 126.

[25] A. C. Chow, “Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator,” *Trans. Soc. Comput. Simul.*, vol. 13, no. 2, pp. 55–68, 1996.

[26] T. Schwatinski and T. Pawletta, “An advanced simulation approach for parallel DEVS with ports,” in *Proceedings of the 2010 Spring Simulation Multiconference*, 2010, p. 147.

[27] H. O. Aliyu, “AUTOMATED CODE SYNTHESIS FOR MODELING AND SIMULATION,” African University of Science and Technology, Abuja, Nigeria, 2011.

[28] “Unified Modeling Language (UML).” [Online]. Available: http://www.uml.org/. [Accessed: 16-Nov-2014].

APPENDIX

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

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

PDEVS SIMULATORS

For **Coupled Models** Figure I: Sample Model

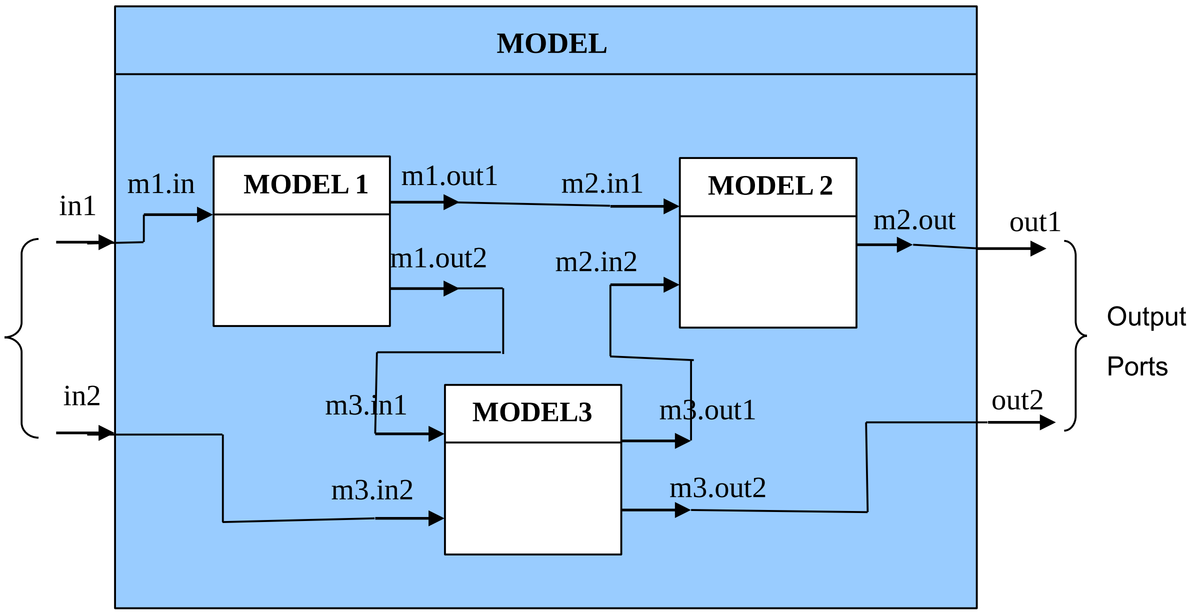


Figure 33: The Figure above will be used to describe the Coupled Model specification

PDEVS SIMULATOR (OOP):

Atomic Models can be defined as

//Import the necessary packages in the SimStudio.

import model.*;

//Give the name of the atomic model to be simulated and extend the AtomicModel //class

```
public class MySubModel extends AtomicModel {


//Declare state variables

        int sigma;

// Declare the ports to be used

        Port portVariable, portVariable2;


//Define the constructor:

//      1. Call the constructor of the super class

//      2. Define input and output ports

//      3. Initialize state variables

//      DataType of port values can be any data type permissible by java

        public MySubModel(String name) {

                super(name);

                sigma = 1;

                portVariable = new Port(new Integer(0));

        portVariable2 = new Port(“name”,new String());

                addInPort(portVariable);

        addOutPort(DEVS_Type type, String nam, String desc);

        }
```

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

        //Override the output function of the super class
        //Define the function
        //Set the output port(s) to receive an event
        @Override
        public void lambda() {
                setOutputPortData(DataType value, Port portVariable);
}
//Override the external transition function of the super class
//Define the function and set it to accept an argument
@Override
public void deltaExt(double e) {
                //
}
```

```
//Override the confluent  transition function of the super class

//Define the function

@Override

public void deltaConfluent() {

            //

}


//Override the time advance function of the super class

//Define the function

//Set it to return the time interval between two transitions

@Override

      public double timeAdvance() {

            return sigma;

      }

}
```

For **Coupled Models**

```
//Import the necessary packages in the SimStudio.

import model.*;


//Give the name of the coupled model and extend the CoupledModel class

public class MyModel extends CoupledModel {
```

```
//Declare atomic models
                mySubModel model;


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


                public MyModel(String name) {
                        super(name);
                        model = new mySubModel("ModelName");
                        addSubModel(model);
                        addIC(Model1,Model1.getOutputPortVariable,
                        Model2,Model2.getInputPortVariable);
                        or
                        addIC(Model1,Model1.getOutputPort("m1.out1"),
                        Model2,Model2.getInputPort("m2.in1"));
                        addEIC(Model1.getInputPortVariable);
                        addEOC(Model2.getOutputPortVariable);
                }
        }
```

START THE SIMULATOR

To start the simulator a new class has to be created

```
//Import the necessary packages in the SimStudio.

import processor.*;


//Give the name of the new class

public class Simulation {


//Create a Main method with an exception

//        1. Create and initialize an object of the Coupled Model

//        2. Create and initialize an object of RootCoordinator class with the simulator //
          associated to the coupled model

//        3. The initial time starts from zero always

//        4. Set the number of times the simulation should run


public static void main(String args[]) throws DEVS_Exception {

          CoupledModel coupledModel;

          RootCoordinator root = new RootCoordinator(coupledModel.getSimulator());

          root.run(simulationTime);

}
```

PDEVS SIMULATOR (Process):

This is basically done using the same format as the OOP variant, the only difference lies in the simulator as shown below:

START THE SIMULATOR

To start the simulator a new class has to be created

```
//Import the necessary packages in the SimStudio.

import processor.*;


//Give the name of the new class

public class Simulation {


//Create a Main method with an exception

//      1. Create and initialize an object of the Coupled Model

//      2. Create and initialize an object of RootCoordinator class with the simulator //
        associated to the coupled model

//      3. Set the number of times the simulation should run


public static void main(String args[]) throws DEVS_Exception {

        CoupledModel coupledModel;

        RootCoordinator root =

        new RootCoordinator(coupledModel.getSimulator(),simulationTime);

        root.simulate();

}
```