To My Grand mum

Acknowledgement

There are many people who have helped me directly or indirectly with this thesis work. I am particularly grateful to my supervisor, Prof. Mamadou Kaba Traore for introducing me to this subject and many instructions on this subject. I appreciate his guidance, encouragement, and support during the period I worked on this thesis.

I owe deep gratitude to Dr. Boubou Cisse and Prof. Charles Chidume for their invaluable assistance. I would also like to thank Nafisa Abdullahi and Tracey Odigie for their logistical support. Thanks also to Dr. Ekpe Okorafor and Dr. Guy Degla.

My colleagues at African University of Science and Technology have been very wonderful. Special thanks to Segun for several discussions on my thesis work and for his excellent suggestions. Thanks also to Olamide, Tabot, Ngolah, Hamzat, Larry, Ignace, Patrick, Godfred, Mohammed and Kehinde.

I would also like to thank my dear friends, Aliu and Bright who have impacted my life in tremendous ways. The last ten years of my life have been remarkable because of their support and guidance.

My great family has been very supportive. Thanks to my dad, mum, brothers and sister for their financial assistance, prayers and encouragement.

My greatest thanks go to the Almighty God, to whom I owe my life, for His benevolence, mercy, and love.

This thesis work was funded by the African University of Science and Technology.

Abstract

DEVS models are implemented in various programming languages using various strategies. There is a scientific need to couple these models for larger simulations. Therefore there is a need for interoperability between these software components. Several attempts have been made to achieve this through the use of web services and middleware technologies. This work aims at studying DEVS model interoperability through the use of Virtual Machines. In computing, Virtual Machines have been used to provide software portability and a high level abstraction of computing resources. Our goal is to leverage these benefits to provide a simulation environment for heterogeneous models.

Table of Contents

LIST OF FIGURES

Chapter 1. Introduction

1.1. Motivation

One of the ways in which science has advanced over time has been by experimenting with systems and observing their behavior under certain conditions. This behavior leads to greater understanding of the system and its structural composition. Although this process appears straight forward, it has become much more expensive and impractical with the levels of complexity of the system. For example, consider predicting the behavior of a wild fire under certain conditions. Following the usual approach, the fire will have to be recreated before a battery of tests may begin. This is impractical and so a more subtle and pragmatic approach has to be taken which is Modeling and Simulation (M&S).

A model is a simplified representation of a system. It is a set of rules, instructions and equations used to abstract the details of a system. These instructions are then executed by a computational system to generate behavior. This computational system is termed a Simulator. It employs algorithms or mathematical techniques to execute the model's instructions.

How we go about modeling a system depends on the class of system modeling formalism adopted. Continuous variable dynamic systems can be defined using Differential Equation System Specification (DESS). This involves expressing the system as a series of differential equations. Discrete time dynamic systems are usually described using Discrete Time System Specification (DTSS) and discrete event dynamic systems can be described using Discrete Event System Specification.

Various modeling methodologies exist such as Cellular Automata, Finite Element method and Finite State Machines but the DEVS formalism has emerged as the preferred methodology due its support for other system formalisms. DESS through quantization can have an equivalent DEVS model. DTSS could also be simulated with DEVS by constraining the time advance to be constant. DEVS also allows the modeling of complex systems by hierarchically organize the models.

DEVS was first introduced to the public by Ziegler in 1976 along with the formal specification and simulation algorithms. These algorithms as presented by Ziegler [1] for both atomic and coupled models have been implemented in various programming languages as software packages some of which include: CD++ [2], DEVSJAVA [13] and ADEVS [5]. The diversity of these software components makes model reuse impractical. Model reuse helps to avoid the repetitious development of models in different DEVS domains. The goal of the DEVS community is to have a platform for the parallel and distributed simulation of heterogeneous models. This will promote collaboration and resource sharing within modeling and simulation. Hence the goal is Simulation Interoperability.

A recent proposal by a working group of the Simulation Interoperability Standardization Organization (SISO) [33] aims to achieve this through interface standardization. The approach specifies:

- A standard DEVS model interface
- A standard DEVS simulator interface
- A standard protocol that operates between the two

DEVSML[24] and DEVS Markup language[25] attempt to standardize the model by using Extensible Markup Language (XML) to define the model. The challenge comes with DEVS implementations that define their models with a textual format.

Attempts at providing simulator interoperability have involved the use of well known system integration techniques and middleware concepts to integrate various simulators. DEVS/SOA [26] is a SOAP [34] based web service environment that provides distributed simulation and integration of simulators from various DEVS implementations. However, it only has support for JAVA and .NET frameworks. RESTFUL-CD++ [1] leverages REST principles to provide distributed simulation but is built around the CD++ modeling and simulation environment. Our goal is to provide a modeling and simulation environment where models and simulators in several programming languages can be executed.

1.2. Structure of the Work

In the next chapter, we review the theory of DEVS and more specifically its variants. In chapter 3, we review several tools that have been used to define models based on DEVS. In chapter 4, we go deeper into DEVS standardization and discuss various approaches at achieving simulation interoperability. In chapter 5, we introduce Virtualization and the various techniques and tools.. In chapter 6, we present an architecture that aims to incorporate DEVS, simulation interoperability and Virtualization.

Chapter 2. Discrete Event System Specification (DEVS)

Discrete Event System Specification (DEVS) is a system formalism used in the modeling and simulation of Discrete Event Dynamic Systems (DEDS). It defines system behavior as well as system structure. DEVS involves model composition that leads to hierarchical modeling of the system i.e. higher level components are made up of lower level components. This way, complex systems can be built or aggregated from simple ones.

According to systems theory, a system is a natural or conceptual entity of which we are interested in like the solar system or a car. A system can be classified according to the variables that define its behavior and how these variables change over time. Thus we have: Continuous variable Dynamic systems; Discrete Time Dynamic systems; Discrete Event Dynamic systems and Discrete Dynamic systems. DEVS is currently applied to describing Discrete Event Dynamic systems.

To experiment on a system, a model of that system is developed. A model is a simple representation of a system. It is a set of instructions, rules and mathematical equations that is used to describe the system. The behavioral data obtained from the system is used to define the model's behavior. The model is then executed and its input/output trajectories are observed over an extended period of time.

To execute the model, a simulator which is a device or set of algorithms capable of executing the model's instructions is developed. Thus we can define Simulation as the process of reproducing the dynamic behavior of a system using a model. Unlike other system formalisms, DEVS focuses on events. When an event occurs, there is a change in the state of the system resulting in a different output. Hence, a series of events completely describes the input/output behavior of the system.

The formal specification of DEVS was established by Ziegler in 1976 and involves the separation of the model from the simulator. This coupled with the hierarchical modeling technique of DEVS have enabled formal proofs to be carried out on different entities under study. These proofs include the proof of correctness of the simulation algorithms i.e. the ability of the algorithms to correctly execute the instructions of the model.

2.1. The DEVS Formalism

2.1.1. Classic DEVS

According to Classical DEVS theory, an atomic model can be defined as:

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

Where

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* is the set of input ports and X_p is the set of values for the input ports;

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

S is the set of sequential *states*;

δ_{ext}: $Q \times X \rightarrow S$ is the *external* state transition function,

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

δ_{int}: $S \rightarrow S$ 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.

Every DEVS model is in a state $s \in S$. In the absence of external events, it remains in that state for a lifetime defined by $ta(s)$. When $ta(s)$ expires, the model outputs the value $\lambda(s)$ through a port $y \in Y$, and it then changes to a new state given by $\delta_{int}(s)$. This change of state is called an **Internal Transition**. When an external event occurs, the external transition function determines the new state, given by $\delta_{ext}(s, e, x)$, where s is the current state, e is the elapsed time since the last transition, and x is the external event that has been received.

A DEVS Coupled model which is composed of atomic models is defined as

$CM = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, select \rangle$

Where

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* is the set of input ports and X_p is the set of values for the input ports;

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

D is the set of the component names and for each $d \in D$;

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

EIC is the set of external input couplings;

EOC is the set of external output couplings;

IC is the set of internal couplings;

Select: $2^D \rightarrow D$ is the tiebreaker function

2.1.2. Parallel DEVS

Parallel DEVS was introduced after Classic DEVS and helped to address some of its limitations. It allows parallel and distributed computing in modeling and simulation. According to Classical DEVS when two or more models are imminent at same time, the Select function breaks the tie and selects a particular model to be executed first. With Parallel DEVS, all imminent models are allowed to be activated at same time and to send their outputs to other models. There is also an additional function called the *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. Parallel DEVS defines an Atomic model as:

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

Where

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* is the set of input ports and X_p is the set of values for the input ports;

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

S is the set of sequential *states*;

$\boldsymbol{\delta_{ext}: Q \times X^b \rightarrow}$ S is the external transition function;

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

$\boldsymbol{\delta_{con}: S \times X^b \rightarrow}$ S is the confluent transition function;

$\lambda: S \rightarrow Y^b$ is the *output* function; and

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

Parallel DEVS models use bags of events for receiving and collecting outputs (X^b, Y^b). this way, multiple events can be handled at same time. For example, if one or more external events $X^b = \{x_1 \ldots x_n / x_i \in X\}$ occur 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, $\delta_{ext}(s, e, X^b)$. We also have a scenario of when the internal transition and external transition are scheduled to occur at same time. The Confluent transition function can intervene by determining the next state of the model. More formally, if the external events X^b are received when $e = ta(s)$, the new state of the model will be given by the confluent function (δ_{con}).

A coupled model in PDEVS is defined as

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

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* is the set of input ports and X_p is the set of values for the input ports;

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

D is the set of the component names and for each $d \in D$;

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

EIC is the set of external input couplings;

EOC is the set of external output couplings;

IC is the set of internal couplings;

2.1.3. DEVS simulation protocol

As identified earlier, a simulator is a set of algorithms for executing the instructions of a model. The algorithms for simulating atomic and coupled models were proven to correctly execute models that adhere strictly to the DEVS formalism.

In other to simulate complex models (models composed of sub-models), the abstract simulation technique was developed were by simulators are hierarchically organized to match the structure of a coupled model. This concept is described in Figure 1. Simulators are matched to their corresponding atomic models. Coordinators are associated with coupled models.

The structure of DEVS is as follows:

- Every atomic model is an instance of an atomic class
- Every coupled model is an instance of a coupled class
- The atomic class has a simulator class
- An instance of this simulator class is responsible for simulating an atomic model

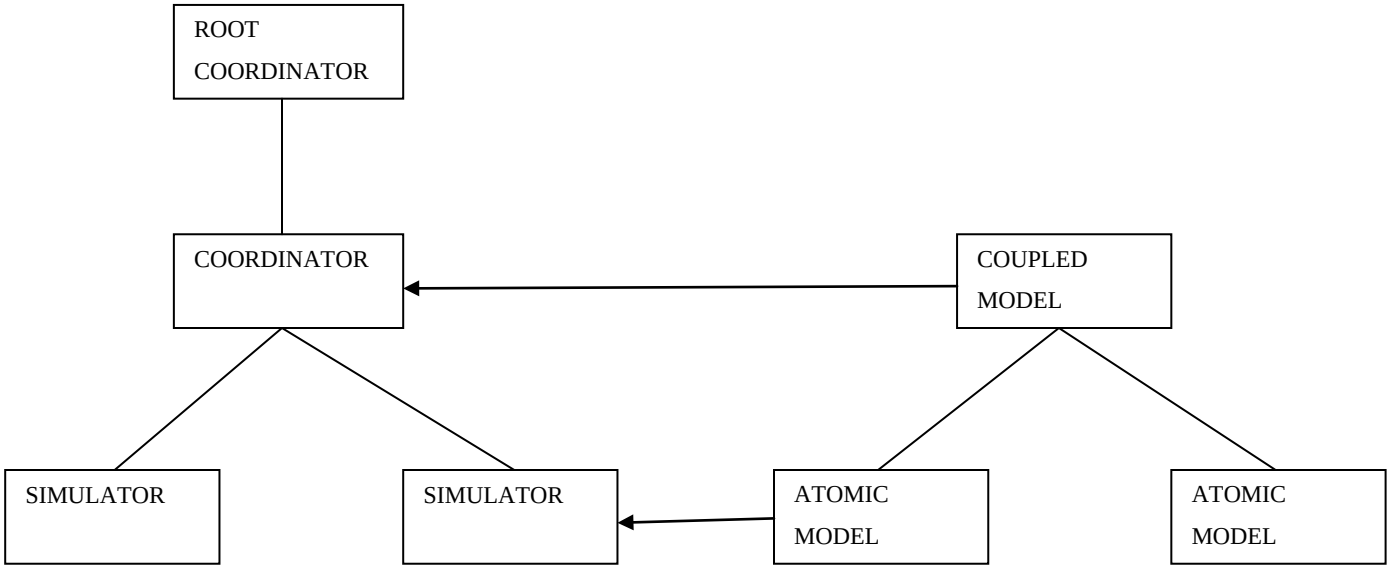

Figure 1. Relationship between models and simulators

- The coupled class has its own simulator called the coordinator class
- Each coupled model is handled by a coordinator during simulation
- There is a standard messaging protocol for message exchange during simulation. The types of messages are:

Initialization Message (i,t): It is sent from the coordinator to all its subordinates;

Input Message (x, t): Sent from the coordinator to the simulators and cause an external transition in the models;

Internal State Transition Message (*,t): It is sent by the coordinator to the simulators to cause an internal state transition in the models;

Output Message (y,t): It is sent by the simulators to their coordinators to notify them of output events;

Done Message: It carries scheduling information for future events, including that a model has finished with its current task;

For details of the simulation algorithms for DEVS (simulator, coordinator and root) and for PDEVS and efficient realizations of these algorithms, refer to [1].

2.2. DEVS Model Example

Figure 2 shows a Traffic light system described with DDML[22] (a graphical notation for modeling dynamic systems based on DEVS). It is composed of two atomic models (Generator and Tlight). The select flag shows the priority of the models. The Tlight is randomly switched ON/OFF by the Generator. The Generator has an output port (outputGen) with domain {0,1} defined. The output port of the Generator is connected to the input port of the Tlight (inputTlight) which has a domain of {0,1}.

The Tlight model described here has been used as a "Hello World" example in DEVS. It is shown in Figure 3. The model consists of seven states GREEN, YELLOWAR, YELLOWBR , BFBLACK, BLACK, AFBLACK and RED. Once simulation begins, the model is initialized to the GREEN state. After ten seconds, an internal transition occurs and the state of the model is changed to the YELLOWBR state. Just before the change of state, the output function is triggered and an output of integer value 0 is placed on the output port of the model. The model then changes state after a second to the RED state outputting a value of 1 and then to the YELLOWAR state after five seconds outputting a value of 2.

The BLACK state is used to represent the situation when the traffic light has no output. This could be as a result of power failure. The Generator is responsible for generating the failure signals at random. Once an integer value of 0 is received by the Tlight model, it undergoes a deltaext transition and changes to the BFBLACK state. It remains in this state for approximately 0.0 seconds after which it changes to the BLACK state. It remains in this state until it receives an input of integer value 1 from the Generator. After which it changes to the AFBLACK state and then to the RED state after 0.0 seconds.

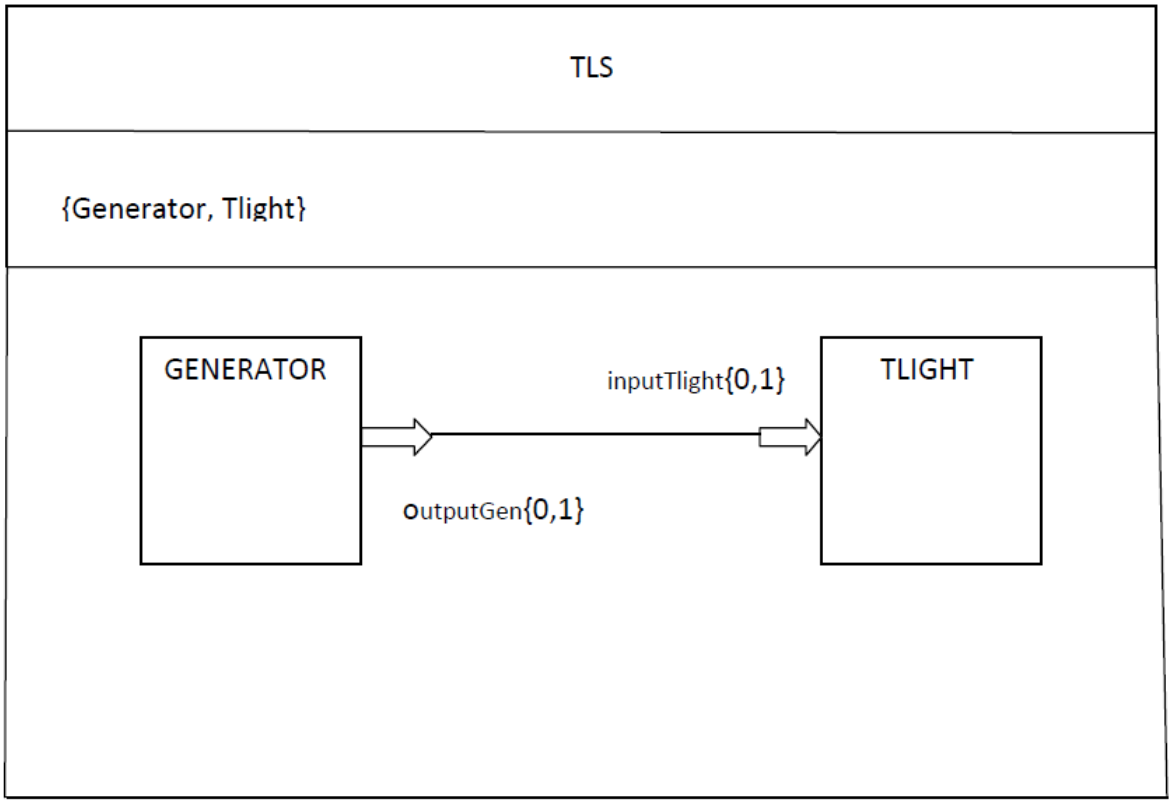

Figure 2. Traffic Light System (TLS Coupled Model)

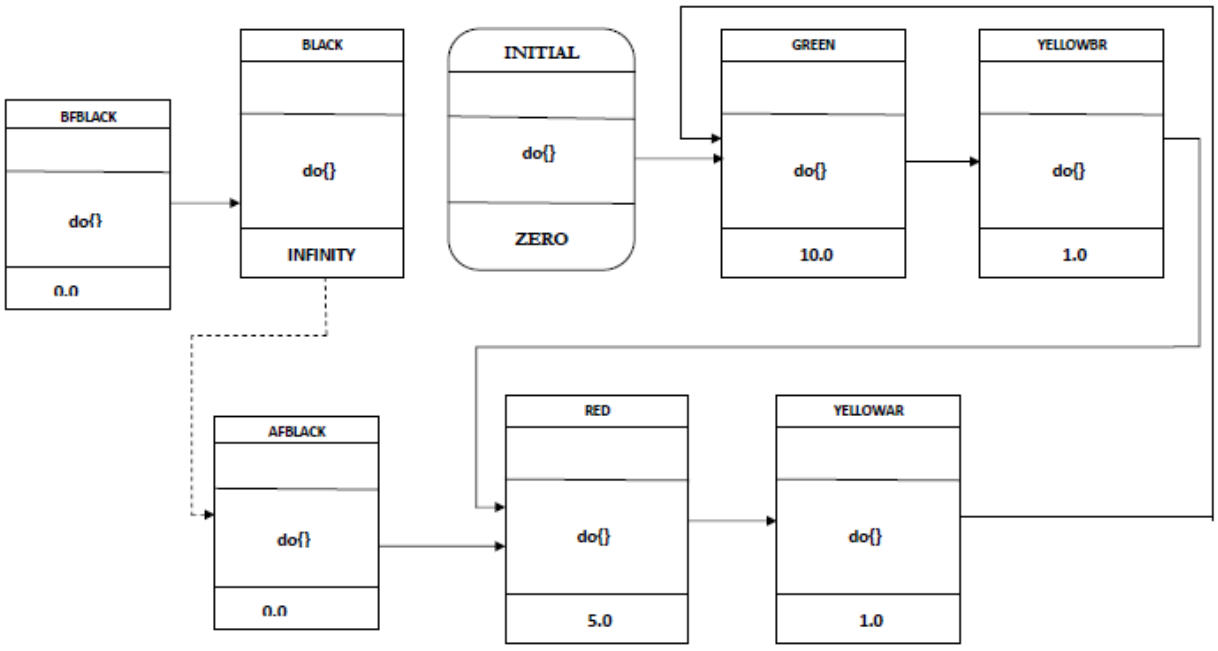

Figure 3. Tlight Atomic model

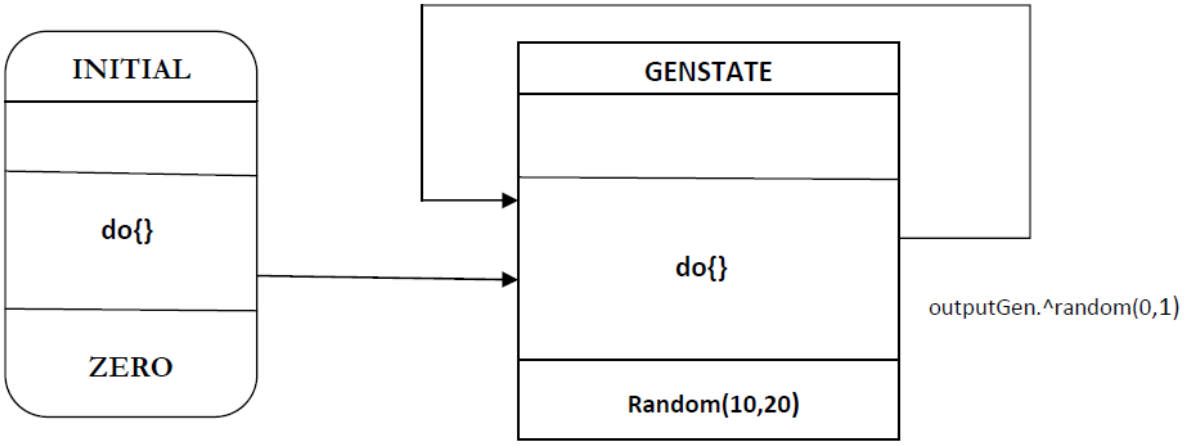

Figure 4. Generator Atomic model

Chapter 3. Survey of DEVS Implementations

The continued interest in DEVS has led to the development of many simulation tools implemented in several programming languages and adopting the object-oriented methodology of DEVS. The following is a list of these tools (although not exhaustive as research and development is still ongoing):

- CD++ Builder [2][3] is an Eclipse plug-in for developing CD++ and Cell-DEVS models for simulation and analysis. It is based on the C++ programming language.
- CD++ modeller [4] is a graphical user interface for developing CD++ and Cell-DEVS models.
- ADEVS [5] is a C++ library used for constructing discrete event simulations based on the parallel DEVS and dynamic DEVS formalisms.
- DEVS-Ada/TW[6] was the first attempt to combine DEVS and the Time Warp parallel simulation algorithm over a multiprocessor environment. DOHS, the distributed optimistic hierarchical simulation scheme, combine DEVS and Time Warp, implemented in DEVSim++. This alternative presents a more general approach for distributed optimistic execution of DEVS models, while addressing some restrictions introduced in DEVS-Ada/TW
- DEVS-C++ [7] is a DEVS-based modeling and simulation environment written in C++, which implements the Parallel DEVS formalism.
- DEVS++ is a C++ open-source library of DEVS.
- DEVS# [32] is a C# open source library of DEVS.
- DEVS-Scheme [8] [9] is a knowledge-based environment for modelling and simulation based on the Scheme functional language (a variation of Lisp).
- DEVS/HLA [10] [11] is based on the high-level architecture (HLA) [12]. It was used to demonstrate how an HLA-compliant DEVS environment could improve the performance of large-scale distributed modeling and simulation.
- DEVSJAVA [13] is a Modeling and simulation environment that supports Parallel DEVS models with real-time, variable structure, 2D/3D cellular automata and animation. It is based on JAVA programming language..
- DEVS-Suite [14] is a Parallel DEVS simulator that supports animating models and generating trajectories at run-time. It is based on Java programming language and combines the capabilities of DEVS tracking environment and DEVSJAVA
- DEVSim++ [15] is an object-oriented DEVS simulator implemented in C++. The tool defines basic classes that can be extended by users to define their own atomic and coupled DEVS components.
- GALATEA [16] is a simulation platform that offers a language to model multi-agent systems using an object-oriented architecture.
- JAMES [17] implements DEVS theory to model and simulate agent systems. The toolkit supports software-in-the-loop simulation to test agents in virtual environments.
- JDEVS [18] is a DEVS modeling and simulation environment written in Java. It allows general-purpose, component-based, object-oriented, visual simulation of models.

- PyDEVS [19] uses the ATOM3 tool [20] 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.
- SimStudio[20] is a modelling and simulation environment implemented in JAVA programming language that manages the communication between models, manages time and uses the specifications made by the modeller.
- SimBeams [21] 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.

With so many DEVS implementations, collaboration and resource sharing becomes a challenge. A modeller has to learn each DEVS implementation in other to use models developed in these domains.

3.1. Evaluation of Tools

We implemented the Traffic light system described in Chapter 2 with some of the tools highlighted earlier. These include SimStudio, Adevs, CD++, DEVS# and DEVS-Suite. The remainder of this section shows how these tools differ in simulating same DEVS model.

3.1.1. SimStudio

The Traffic light system was implemented with SimStudio. Figure 7 shows the Tlight implementation, Figure 8 shows the Generator implementation and the Simulation code is shown in Figure 5. The TLS coupled model is shown in Figure 6. A FileManager class has been implemented for observation purposes. It enables the results of the simulation to be written to a file.

```
import exception.DEVS_Exception;
import simulator.RootCoordinator;

public class Simulation {

    public static void main(String [] args) throws DEVS_Exception{
        TLS myLight = new TLS("AbujaCRd");
        RootCoordinator root = new RootCoordinator(myLight.getSimulator());
        root.init(0.0);
        root.run(100.0);
    }

}
```

Figure 5. Simulation code (SimStudio)

```
import java.util.ArrayList;

import model.CoupledModel;
import model.Model;



public class TLS extends CoupledModel{
	Model Tlight;
	Model generator;


	public TLS(String name) {
		super(name, "TLS Coupled Model");
		Tlight = new Tlight("TLight");
		generator = new Generator("Generator");

		addSubModel(generator);
		addSubModel(Tlight);

		addIC(generator.getOutputPortStructure("GenOutput"),
Tlight.getInputPortStructure("InputLight"));

	}

	@Override
	public Model select(ArrayList<Model> arraylist) {
		for(Model model : arraylist){
			if(model instanceof Generator)return model;
		}
		if(arraylist.get(0)instanceof Tlight)return arraylist.get(0);
		else return arraylist.get(1);
	}


}
```

Figure 6. TLS coupled model (SimStudio)

```
import java.io.FileWriter;
import java.io.IOException;

import types.DEVS_Integer;
import types.DEVS_Real;
import exception.DEVS_Exception;
import model.AtomicModel;


public class Tlight extends AtomicModel {
	int colour;                                    //0 for Green
                                                   //1 for Yellow Before Red
                                                   //2 for red
                                                   //3 for Yellow After Red
                                                   //4 for Before Black
                                                   //5 for Black
                                                   //6 for After Black
	Double time, sigma;
	String output;
	FileManager fileManager = new FileManager();

	public Tlight(String name) {
		super(name, "Input/Output-free Traffic Light");
		addInputPortStructure(new DEVS_Integer(), "InputLight", "Input
Port");
		colour = 0;
		sigma = 0.0;
	}

	public void deltaExt(double e) throws DEVS_Exception {
		int n;
		n = ((DEVS_Integer)getInputPortData("InputLight")).getInteger();
		switch(n){
		case 0: {colour = 4; sigma = 0.0 ; break;}
		case 1:
				if(colour == 5){colour = 6;sigma = 0.0;}
				else sigma = sigma - e;
				break;
		default: System.out.println("Error");
		}
	}

	@Override
	public void deltaInt() {
		switch(colour){
		case 0: colour = 1;sigma = 1.0; break;
		case 1: colour = 2;sigma = 5.0; break;
		case 2: colour = 3;sigma = 1.0; break;
		case 3: colour = 0;sigma = 10.0; break;
		case 4: colour = 5;sigma = DEVS_Real.POSITIVE_INFINITY; break;
		case 6: colour = 2;sigma = 5.0; break;
	}

	}

	public void lambda() throws DEVS_Exception{
		time = getSimulator().getTL();
```

```
            switch(colour){
            case 0:
                  output = (time.toString()+ " : " + "Yellow");
                  try {
                        fileManager.saveToFile(output, "text.txt");
                  } catch (IOException e) {}break;
            case 1:
                  output = (time.toString()+ " : " + "Red");
                  try {
                        fileManager.saveToFile(output, "text.txt");
                  } catch (IOException e) {}break;
            case 2:
                  output = (time.toString()+ " : " + "Yellow");
                  try {
                        fileManager.saveToFile(output, "text.txt");
                  } catch (IOException e) {}break;
            case 3:
                  output = (time.toString()+ " : " + "Green");
                  try {
                        fileManager.saveToFile(output, "text.txt");
                  } catch (IOException e) {}break;
            case 4:
                  output = (time.toString()+ " : " + "Black");
                  try {
                        fileManager.saveToFile(output, "text.txt");
                  } catch (IOException e) {}break;
            case 6:
                  output = (time.toString()+ " : " + "Red");
                  try {
                        fileManager.saveToFile(output, "text.txt");
                  } catch (IOException e) {}break;
            }
      }

      public double ta() {
            return sigma;
      }

}
```

Figure 7. Traffic light implemented with SimStudio

```
import java.util.Random;

import types.DEVS_Integer;
import exception.DEVS_Exception;
import model.AtomicModel;


public class Generator extends AtomicModel{
	Random rand ;

	public Generator(String name) {
		super(name, "Generator for Traffic Light");

		addOutputPortStructure(new DEVS_Integer(), "GenOutput",
"Generator Function");

	}

	public void deltaExt(double e) throws DEVS_Exception {

	}

	public void deltaInt() {}

	public void lambda() throws DEVS_Exception {
		Random rand = new Random();
		int outcode = rand.nextInt(2);
		System.out.println(outcode);
		setOutputPortData("GenOutput", new DEVS_Integer(outcode));
	}

	public double ta() {
		Random rand = new Random();

		return ((rand.nextInt(20)) + 10);
	}

}
```

Figure 8. Generator(SimStudio)

To simulate the model, a root coordinator is created and the init method is called at time 0.0 to initialize the simulation. The run method is then called to start and run the simulation for 100 secs.

3.1.2. DEVS-Suite

DEVS-Suite is a modeling and simulation environment that enables visual design of experiments and introduces simulation data visualization. It incorporates both DEVSJAVA and DEVS Tracking environment. It provides a graphical interface for simulation thus; the user is only required to develop the DEVSJAVA model. To simulate, the graphical simulator is started and the model selected as shown in Figure 9. The Traffic light model was implemented with DEVS-Suite and is shown below in Figure 10. The simulator also gives the user the option to inject test values for initialization. The simulation can be run in steps or till a specified time.

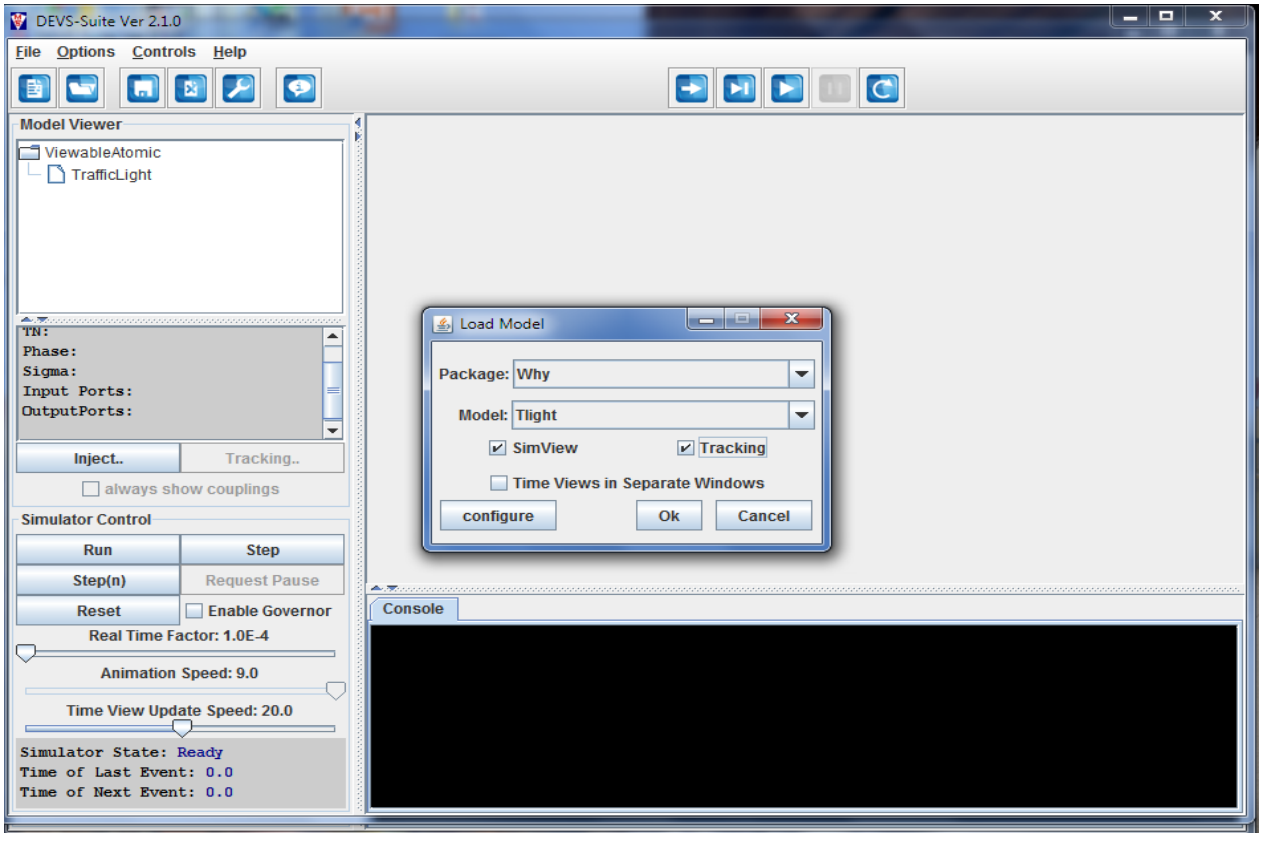

Figure 9. DEVS-Suite Graphical Interface

```
import crossroad.Car;
import GenCol.entity;
import GenCol.intEnt;

import model.modeling.content;
import model.modeling.message;
import view.modeling.ViewableAtomic;

public class Tlight extends ViewableAtomic{
	intEnt output;				//0 for Green
							//1 for Yellow
							//2 for Red


	public Tlight(){
		super("TrafficLight");
		addInport("tlightInput");
		addOutport("tlightOutput");
		initialize();

	}

	public void initialize(){
		super.initialize();
		sigma = 0.0;
		this.showstate();
		this.holdIn("GREEN", 10.0);
	}

	public void deltext(double e, message x){
		Continue(e);
		for(int i = 0; i<x.getLength(); i++){
			if(this.messageOnPort(x, "tlightInput", i)){
				intEnt genIn = (intEnt) x.getValOnPort("tlightInput",
i);
				if(genIn.getv() == 0){
					this.holdIn("BFBLACK", 0.0);
				}else {
					if(this.phaseIs("BLACK")){
						this.holdIn("AFBLACK", 0.0);
					}
				}
			}
		}
	}

	public void deltint(){
		if(this.phaseIs("GREEN")){
			this.holdIn("YELLOWBR",1.0);
		}else{
			if(this.phaseIs("YELLOWBR")){
				this.holdIn("RED", 5.0);
			}else{
				if(this.phaseIs("RED")){
					this.holdIn("YELLOWAR", 1.0);
				}else{
```

```
                                    if(this.phaseIs("YELLOWAR")){
                                          this.holdIn("GREEN", 10.0);
                                    }else{
                                          if(this.phaseIs("BFBLACK")){
                                                this.passivateIn("BLACK");

                                          }else{
                                                this.holdIn("RED", 5.0);
                                          }
                                    }
                              }
                        }
                  }
            }

            public message out(){
                  message m = new message();
                  if(this.phaseIs("GREEN")||this.phaseIs("RED")){
                        content con1 = new content("tlightOutput", new
entity("Yellow") );
                        m.add(con1);
                  }else{
                        if(this.phaseIs("YELLOWBR")||this.phaseIs("AFBLACK")){
                              content con1 = new content("tlightOutput", new
entity("Red")  );
                              m.add(con1);
                        }else{
                              if(this.phaseIs("YELLOWAR")){
                                    content con1 = new content("tlightOutput", new
entity("Green")  );
                                    m.add(con1);
                              }else{
                                    if(this.phaseIs("BFBLACK")){
                                          content con1 = new
content("tlightOutput", new entity("Black")  );
                                          m.add(con1);
                                    }
                              }
                        }
                  }

                  return m;

            }
            public double ta(){
                  return sigma;
            }

            public void showstate(){
                  super.showState();
            }

}
```

Figure 10. Traffic light implemented with DEVS-Suite

```
import java.util.Random;

import model.modeling.content;
import model.modeling.message;
import GenCol.intEnt;
import view.modeling.ViewableAtomic;

public class Generator extends ViewableAtomic{
	Random rand;

	public Generator(){
		super("Gen");
		this.addOutport("genOut");
		rand = new Random();
	}

	public void initialize(){
		super.initialize();
		this.holdIn("generating", ((rand.nextInt(20))+10.0));
	}
	public void deltext(double e, message x){
		//Do nothing

	}

	public void deltint(){
		this.holdIn("generating", ((rand.nextInt(20))+10.0));
	}

	public message out(){
		message m = new message();
		intEnt output = new intEnt(rand.nextInt(2));
			content con1 = new content("genOut", output );
			m.add(con1);
		return m;

	}
	public double ta(){

		return sigma;
	}

	public void showstate(){
		super.showState();
	}

}
```

Figure 11. Generator(DEVS-Suite)

```
import view.modeling.ViewableAtomic;
import view.modeling.ViewableDigraph;

public class TLS extends ViewableDigraph{

      public TLS() {
            super("TLS");
            // TODO Auto-generated constructor stub
            ViewableAtomic gen = new Generator();
            ViewableAtomic tlight = new Tlight();

            this.add(gen);
            this.add(tlight);
            this.addCoupling(gen, "genOut", tlight, "tlightInput");
      }



    /**
     * Automatically generated by the SimView program.
     * Do not edit this manually, as such changes will get overwritten.
     */
    public void layoutForSimView()
    {
        preferredSize = new Dimension(591, 332);
        ((ViewableComponent)withName("Gen")).setPreferredLocation(new
Point(11, 34));

((ViewableComponent)withName("TrafficLight")).setPreferredLocation(new
Point(266, 37));
    }
}
```

Figure 12. TLS coupled model(DEVS-Suite)

3.1.3. CD++

The traffic light model in CD++ is shown in Figure 14.A new atomic model is created as a new class object that inherits from the class Atomic and implements the methods as shown. To tell CD++ that a new atomic definition has been added, the model must be registered with the main simulator.

MainSimulator.registerNewAtomic();

This method is located in the file register.cpp. Coupled models are defined using a specification language specifically defined for this purpose. These definitions consist of port connections, atomic models and user-defined parameters for initialization. The definitions are saved in a file with a .ma extension.

With CD++ builder, CD++ has been integrated with Eclipse Integrated Development Environment as a plug-in. This gives the user the option to use a graphical interface or the

command prompt to simulate the model. To simulate the model, the source files are compiled. The graphical interface is opened and the parameters specified including the location of the file definitions. This is shown in Figure 13. CD++ builder also gives you the option to save the output events generated by the simulator as well as all the messages exchanged between the simulators.

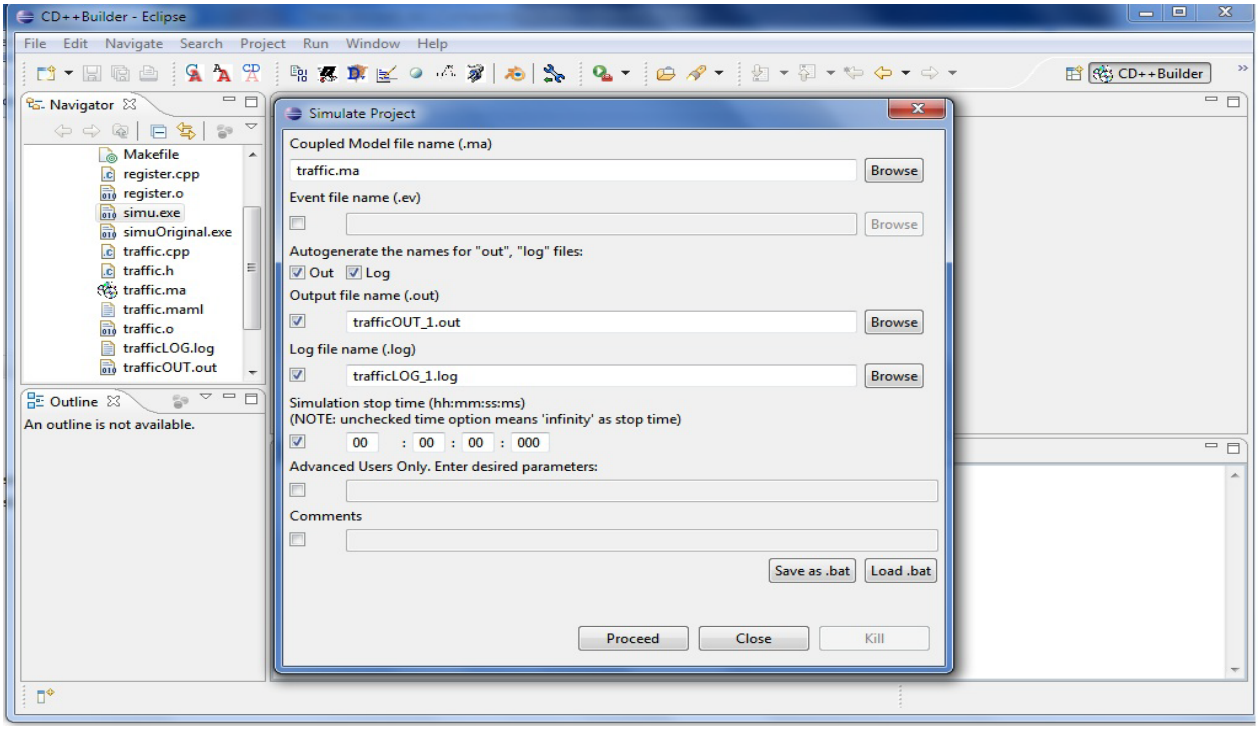

Figure 13. CD++ Builder

```
#include "traffic.h"
#include "model.h"
#include "mainsimu.h"
#include "message.h"


TLight::TLight(const string &name)
: Atomic(name)
, red(addOutputPort("red"))
, green(addOutputPort("green"))
, yellow(addOutputPort("yellow"))
, black(addOutputPort("black"))
, in1(addInputPort("in1"))
, in0(addInputPort("in0"))

{
	value = 0.0;
	gtime = Time(0,0,10,0);
	rtime = Time(0,0,5,0);
	ytime = Time(0,0,1,0);
	aftime = Time(0,0,0,0);
	bftime = Time(0,0,0,0);
}

Model &TLight::initFunction()
{
	this->holdIn(greenState, gtime);
	value = 0.0;
	return *this;
}


Model &TLight::internalFunction( const InternalMessage & )
{
	if(state()== greenState){
		this->holdIn(yellowbrState, ytime);
		value = 2.0;
	}else{
		if(state()== yellowbrState){
			this->holdIn(redState, rtime);
			value = 0.0;
		}else{
			if(state()== redState){
				this->holdIn(yellowarState, ytime);
				value = 2.0;
			}else{
				if(state()== yellowarState){
					this->holdIn(greenState, gtime);
					value = 1.0;
				}else{
					if(state()== bfblack){
						this->holdIn(blackState, Time::Inf);
						value = 3.0;
					}else{
						this->holdIn(redState, rtime);
						value = 0.0;
					}
```

```
                        }
                    }
                }
            }
            return *this;
}
Model &TLight::externalFunction( const ExternalMessage &msg )
{
            if(msg.port() == in0 ){
                this->holdIn(bfblack, bftime);
            }else{
                if((state() == blackState)){
                    this->holdIn(afblack, aftime);
                }
            }
            return *this;
}


Model &TLight::outputFunction( const InternalMessage &msg)
{
            if(state()== greenState){
                this->sendOutput( msg.time(), yellow, value);
            }else{
                if((state() == yellowbrState)){
                    this->sendOutput(msg.time(), red, value);
                }else{if(state() == yellowarState){
                    this->sendOutput(msg.time(), green, value);
                    }else{
                        if(state() == redState){
                            this->sendOutput(msg.time(), yellow, value);
                        }else{
                            if(state()== bfblack){
                                this->sendOutput(msg.time(), black,
value);
                            }else{
                                this->sendOutput(msg.time(), red, value);
                            }
                        }
                    }
                }
            }
            return *this;
}
```

Figure 14. Traffic light CD++

+

```
#include "gen.h"
#include "model.h"
#include "mainsimu.h"
#include "message.h"
#include <cstdlib>
#include <mcd++/time.h>



Gen::Gen(const string &name)
: Atomic(name),
  genOut1(addOutputPort("genOut1")),
genOut0(addOutputPort("genOut0"))
{
      value = 0.0;
      gtime = Time(0,0,0,0);
      randNo = 0;
}

Model &Gen::initFunction(){
      srand(time(NULL));
      randNo = (rand() % 20)+ 10 ;
      gtime = Time(0,0,randNo,0);
      this -> holdIn(active,gtime );
      return *this;
}

Model &Gen::internalFunction( const InternalMessage & ){
      randNo = (rand() % 20)+ 10 ;
      gtime.seconds(randNo);
      randNo = (rand() %2);
      if(state() == active){
            this->holdIn(active, gtime);
            value = randNo;
      }
      return *this;
}


Model &Gen::externalFunction( const ExternalMessage & ){

      return *this;
}


Model &Gen::outputFunction( const InternalMessage &msg){

      if(value == 0.0){
            this->sendOutput( msg.time(), genOut0, value);
      }else{
            this->sendOutput( msg.time(), genOut1, value);
      }
      return *this;
}
```

Figure 15. Generator (CD++)

```
[top]
components : tlight@TLight gen@Gen
out : red green yellow

Link : red@TLight red
Link : yellow@TLight yellow
Link : green@TLight green
Link : genOut0@gen in0@tlight
Link : genOut1@gen in1@tlight
```

Figure 16. Coupling of Generator and Tlight models (CD++)

```
#include "modeladm.h"
#include "mainsimu.h"
#include "traffic.h"     // class Server
#include "gen.h"



void MainSimulator::registerNewAtomics()
{

SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<TLight>() ,
"TLight" ) ;
SingleModelAdm::Instance().registerAtomic( NewAtomicFunction<Gen>() , "Gen" )
;

}
```

Figure 17. Register.cpp (CD++)

3.1.4. ADEVS

ADEVS is a C++ library based on DEVS. The Traffic light model is shown in Figure 18. An observer class has been created for purpose of saving the simulation results to a file. To observe the behaviour of the Traffic light model, it is coupled with the Observer atomic model. A simulator is created and the coupled model is supplied to the constructor as a parameter. This is shown in Figure 19.

Simulation begins when the program execution gets to the while block within the simulation code. Within the while block, the simulation runs until the time specified by the user.

```
#include "traffic.h"
#include "adevs.h"
#include "iostream.h"
#include "string.h"



Traffic::Traffic():
sig(10.0), simTime(0.0), state(0.0), inp(0),output_strm("debug.txt")
{
}

void Traffic::delta_ext(double e, const adevs::Bag<adevs::PortValue<Light> > &xb){
        simTime += e;

    adevs::Bag<adevs::PortValue<Light> >::iterator iter;
            for (iter = xb.begin(); iter != xb.end(); iter++)
            {
                if ((*iter).port == inputGen)
                {
                    output_strm << ((*iter).value).input << "         "<<endl;
                    inp = ((*iter).value).input;
                    if((inp == 0)&&(state != 4.0)){
                        state = 4.0; sig = 0.0;
                    }else{
                        if((state == 5.0)){
                            state = 6.0; sig = 0.0;
                        }else{
                            sig -= e;
                        }
                    }
                }
            }
}

void Traffic::delta int(){
    if(state == 0.0){
        state = 1.0; sig = 1.0;
    }else{
        if(state == 1.0){
            state = 2.0; sig = 5.0;
        }else{
            if(state == 2.0){
                state = 3.0;    sig = 1.0;
            }else{
                if(state == 3.0){
                    state = 0.0; sig = 10.0;
                }else{
                    if(state == 4.0){
                        state = 5.0; sig = DBL_MAX;
                    }else{
                        if(state == 6.0){
                            state = 2.0; sig = 5.0;
                        }
                    }
                }
            }
        }

    }
}

void Traffic::delta_conf(const adevs::Bag<adevs::PortValue<Light> > &xb){
    // Discard the old job
            delta_int();
```

```
            // Process the incoming job
            delta_ext(0.0,xb);
}

void Traffic::output_func(adevs::Bag<adevs::PortValue<Light> > &yb){
    Light light;
    char green[10] = "Green    ";
    char yellow[10] = "Yellow   ";
    char red[10] = "Red      ";
    char black[10] = "Black    ";
    if(state == 0.0){
        light.leaving = (simTime = simTime + sig);
        light.state = yellow;
    }else{
        if(state == 1.0){
            light.leaving = (simTime = simTime + sig);
            light.state = red;
        }else{
            if(state == 2.0){
                light.leaving = (simTime = simTime + sig);
                light.state = yellow;
            }else{
                if(state == 3.0) {
                    light.leaving = (simTime = simTime + sig);
                    light.state = green;
                }else{
                    if(state == 4.0){
                        light.leaving = (simTime = simTime + sig);
                        light.state = black;
                    }else{
                        if(state == 6.0){
                            light.leaving = (simTime = simTime + sig);
                            light.state = red;
                        }
                    }
                }
            }
        }
    }
    adevs::PortValue<Light> y(depart, light);
    yb.insert(y);
}

void Traffic::gc_output(adevs::Bag<adevs::PortValue<Light> >& g)
{
    //do nothing;
}

double Traffic::ta(){
    return sig;
}

Traffic::~Traffic(){
    output_strm.close();
    // destroy
}
```

Figure 18. Traffic light ADEVS

```
#include "traffic.h"
#include "Observer.h"
#include "gen.h"
#include "light.h"
#include <iostream>
using namespace std;

int main()
{

    // Create a digraph model whose components use PortValue<Customer*>
    // objects as input and output objects.
    adevs::Digraph<Light> store;
    // Create and add the component models
    Traffic* traf = new Traffic();
    Observer* obsrv = new Observer("output.txt");
    Gen* gen = new Gen();

    store.add(gen);
    store.add(obsrv);
    // Couple the components
    store.couple(traf,traf->depart,obsrv,obsrv->departed);
    store.couple(gen, gen->outGen, traf,traf->inputGen);
    // Create a simulator and run until its done
    adevs::Simulator<adevs::PortValue<Light> > sim(&store);
    while (sim.nextEventTime() < 100)
    {
        sim.execNextEvent();
    }
    // Done, component models are deleted when the Digraph is
    // deleted.
    return 0;
}
```

Figure 19. Simulation Code ADEVS

```
#include "gen.h"
#include "adevs.h"
#include "iostream.h"
#include "string.h"
#include <cstdlib>
#include "time.h"



Gen::Gen():
sig(0.0), simTime(0.0), output(0)
{
    srand(time(NULL));
    sig =( (rand() % 20) + 10.0 );
}

void Gen::delta_ext(double e, const adevs::Bag<adevs::PortValue<Light> >&xb){
 // do nothing
}

void Gen::delta int(){
    sig =( (rand() % 20) + 10 );
    output = ((rand() % 2));
}

void Gen::delta_conf(const adevs::Bag<adevs::PortValue<Light> >&xb){
    //do nothing
}

void Gen::output_func(adevs::Bag<adevs::PortValue<Light> >&yb){

    Light light;
    light.input = output;
    adevs::PortValue<Light> y(outGen, light);
    yb.insert(y);
}

void Gen::gc_output(adevs::Bag<adevs::PortValue<Light> >& g)
{
    //do nothing;
}

double Gen::ta(){
    return sig;
}

Gen::~Gen(){
    // destroy
}
```

Figure 20. Generator(ADEVS)

3.1.5. DEVS#

DEVS# is a C# open source library for DEVS. The traffic light model is shown in Figure 22. The code for simulation is shown in Figure 21. An execution engine is created and the atomic model is passed along with the duration of the simulation as arguments to its constructor. The execution

engine provides a console menu which the user can use to control the simulation. The user can specify how the simulation should be run (in steps or till the specified time), settings for log files and inject test values for the simulation.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using DEVSsharp;

namespace Traffic
{
    class Program
    {
        static Devs generate(string name)
        {
            Coupled TLS = new Coupled(name);
            Tlight tlight = new Tlight("tlight");
            Gen gen = new Gen("Gen");
            tlight.CollectStatistics(true);
            gen.CollectStatistics(true);
            TLS.AddModel(tlight);
            TLS.AddModel(gen);
            TLS.AddCP(gen.OutputGen, tlight.InputGen);

            return TLS;
        }
        static void Main(string[] args)
        {
            Devs cp = generate("TLS");
            SRTEngine engine = new SRTEngine(cp, 100, null);
            engine.RunConsoleMenu();
        }
    }
}
```

Figure 21. Simulation code(DEVS#)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DEVSsharp;

namespace Traffic
{
    class Tlight : Atomic
    {
        private enum State { GREEN, YELLOWBR, RED, YELLOWAR, BFBLACK, AFBLACK, BLACK };
        private State state;
        private double sigma;
        private int outputValue;
        const string GREEN = "GREEN";
        const string YELLOWBR = "YELLOWBR";
        const string YELLOWAR = "YELLOWAR";
        const string RED = "RED";
        const string BLACK = "BLACK";
        const string BFBLACK = "BFBLACK";
        const string AFBLACK = "AFBLACK";
        const string EMPTY = "";
        public OutputPort OutputTlight;
        public InputPort InputGen;

        public Tlight(string name)
            : base(name, TimeUnit.Sec)
        {
            state = State.GREEN;
            this.OutputTlight = this.AddOP("OutputTlight");
            this.InputGen = this.AddIP("InputGen");

        }
        public override void init()
        {
            sigma = 10.0;
            outputValue = 0;
        }
        public override double tau()
        {
            if (state == State.GREEN)
            {
                sigma = 10.0;
            }
            else
            {
                if ((state == State.YELLOWBR) || (state == State.YELLOWAR))
                {
                    sigma = 1.0;
                }
                else
                {
                    if (state == State.RED)
                    {
                        sigma = 5.0;
                    }
                    else
                    {
                        if ((state == State.AFBLACK) || (state == State.BFBLACK))
                        {
                            sigma = 0.0;
                        }
                        else
                        {
                            sigma = double.PositiveInfinity;
                        }
                    }
```

```
        }
    }
    return sigma;
}
public override bool delta_x(PortValue x)
{
    int inp = (int)x.value;
    if (inp == 0)
    {
        state = State.BFBLACK;
        return true;
    }
    else
    {
        if (state == State.BLACK)
        {
            state = State.AFBLACK;
            return true;
        }
    }
    return false;
}
public override void delta_y(ref PortValue y)
{
    if (state == State.GREEN)
    {
        state = State.YELLOWBR;
        outputValue = 2;
    }
    else
    {
        if ((state == State.YELLOWBR))
        {
            state = State.RED;
            outputValue = 1;
        }
        else
        {
            if (state == State.RED)
            {
                state = State.YELLOWAR;
                outputValue = 2;
            }else{
                if (state == State.YELLOWAR)
                {
                    state = State.GREEN;
                    outputValue = 0;
                }
                else
                {
                    if (state == State.BFBLACK)
                    {
                        state = State.BLACK;
                        outputValue = 3;
                    }
                    else
                    {
                        if (state == State.AFBLACK)
                        {
                            state = State.RED;
                            outputValue = 1;
                        }
                    }
                }
            }
        }
    }
```

```
            y.Set(OutputTlight, outputValue);
        }
        public override string Get_s()
        {
            switch (state)
            {
                case State.GREEN: return GREEN;
                case State.YELLOWBR: return YELLOWBR;
                case State.YELLOWAR: return YELLOWAR;
                case State.RED: return RED;
                case State.BFBLACK: return BFBLACK;
                case State.AFBLACK: return AFBLACK;
                case State.BLACK: return BLACK;
                default: return EMPTY;
            }
        }

    }
}
```

Figure 22. Traffic light implemented with DEVS#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DEVSsharp;

namespace Traffic
{
    class Gen: Atomic
    {
        private enum State { Generating };
        private State state;
        private string gname;
        public OutputPort OutputGen;
        const string GEN = "Generating";
        private Random rand;
        private double sigma;
        private int output;

        public Gen(string name)
            : base(name, TimeUnit.Sec)
        {
            OutputGen = this.AddOP("OutputGen");
            gname = name;
            state = State.Generating;
            init();

        }
        public override void init()
        {
            sigma = 0.0;
            rand = new Random();
            output = rand.Next(2);
        }

        public override double tau()
        {
            sigma = rand.Next(20) + 10;
            return sigma;
        }

        public override bool delta_x(PortValue x)
        {
            return false;
        }

        public override void delta_y(ref PortValue y)
        {
            if (state == State.Generating)
            {
                y.Set(OutputGen, output);
                state = State.Generating;
                output = rand.Next(2);
            }
        }
        public override string Get_s()
        {

            return GEN;

        }
    }
}
```

Figure 23. Generator(DEVS#)

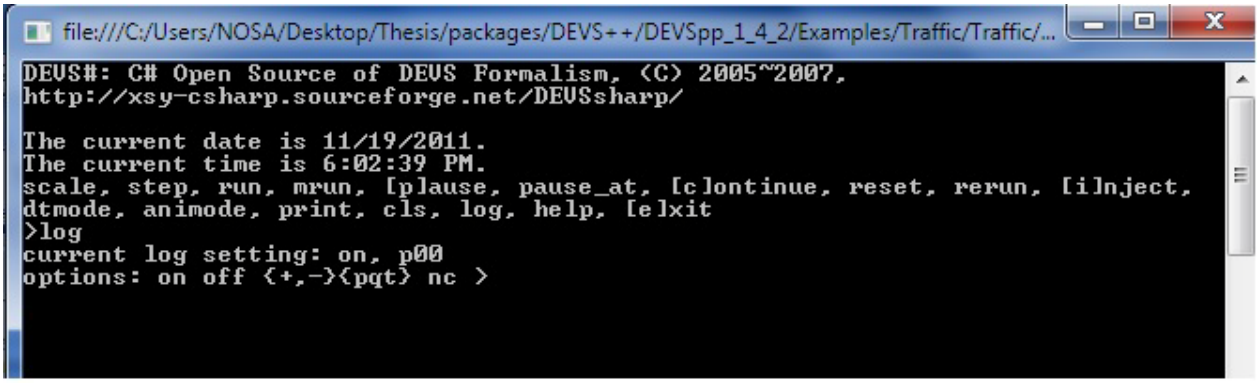

Figure 24. DEVS# console menu

3.2. Results and Discussion

3.2.1. CD++

The results obtained from the simulation are shown in Figure 25. CD++ gives you the option of saving the outputs and the messages exchanged during the simulation. The first column indicates the time of next event and the next column indicates the output port and the value on that port at that time.

```
00:00:10:000 yellow 0
00:00:11:000 red 2
00:00:16:000 yellow 0
00:00:17:000 green 2
00:00:27:000 yellow 1
00:00:28:000 red 2
00:00:41:000 red 3
00:00:46:000 yellow 0
00:00:47:000 green 2
00:00:57:000 yellow 1
00:00:58:000 red 2
00:01:03:000 yellow 0
00:01:04:000 green 2
00:01:14:000 yellow 1
00:01:15:000 red 2
00:01:20:000 yellow 0
00:01:21:000 green 2
```

Figure 25. Results (CD++)

From the results shown, a failure occurs at simulation time 28 secs. There is no output for another 13 secs after which the traffic light resumes operations.

3.2.2. ADEVS

The results of the simulation are shown in Figure 26. The first column indicates the Time of next event while the second column indicates the Output just before that time from the Traffic light model

```
# Col 1: Time of simulation event
# Col 2: TrafficLight output
10        Yellow
11        Red
16        Yellow
17        Green
27        Yellow
28        Red
28        Black
52        Red
57        Yellow
58        Green
67        Black
77        Red
82        Yellow
83        Green
93        Yellow
94        Red
99        Yellow
```

Figure 26. Results (ADEVS)

The execution of the traffic light model is interrupted after 28 secs with an input of 0 from the generator. It changes to the Black state and remains in this state for another 24 secs. Another failure occurs at 67 secs and normal output resumes after anpther 9 secs.

3.2.3. DEVS-Suite

The results of the simulation are shown in Figure 27. From the information given, we can obtain the Time of last event, Time of next event and output. From the results, a failure occurs after 24 secs and the the Traffic light is resumed at 52 sec. Another failure occurs at 98 sec.

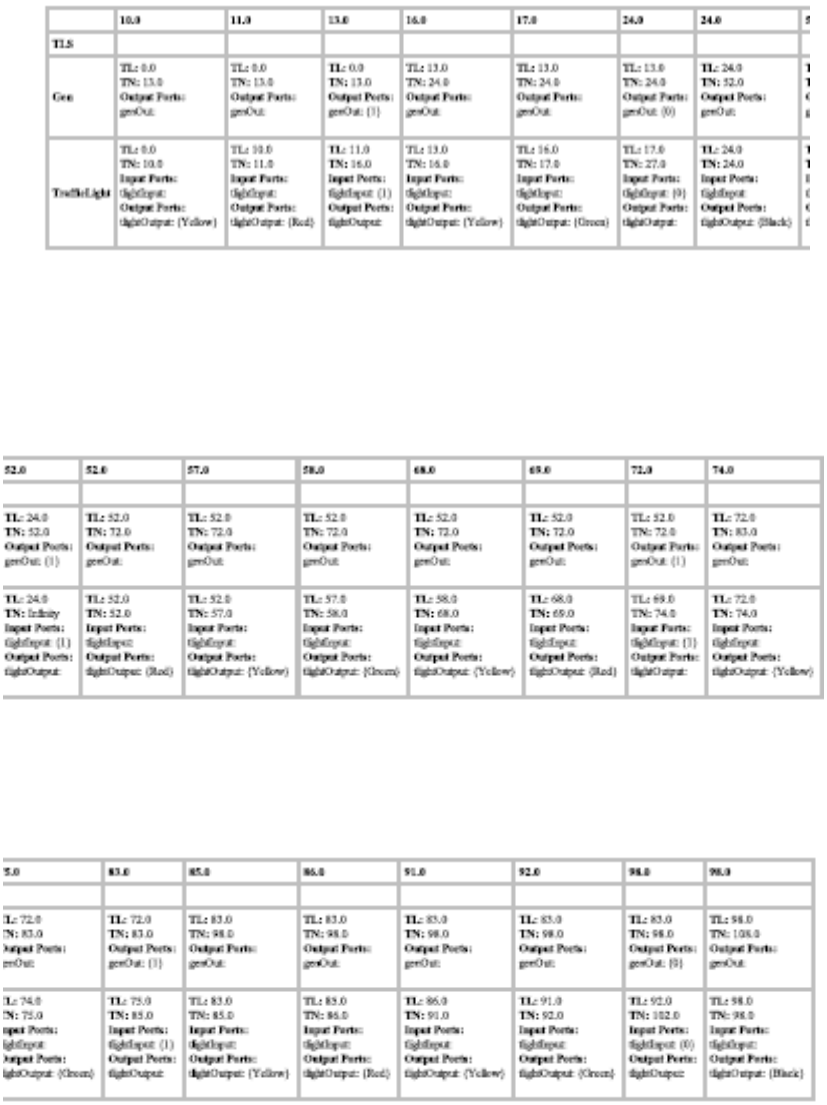

Figure 27. Results (DEVS-Suite)

3.2.4. SimStudio

The results are shown in Figure 28. The first column indicates the Time of next event while the second column indicates the Output just before that time from the Traffic light model. A failure didn't occur until the fifty sixth second. The traffic light was reset at 72 sec

0.0 : Yellow
1.0 : Red
6.0 : Yellow
7.0 : Green
17.0 : Yellow
18.0 : Red
23.0 : Yellow
24.0 : Green
34.0 : Yellow
35.0 : Red
40.0 : Yellow
41.0 : Green
51.0 : Yellow
52.0 : Red
56.0 : Black
72.0 : Red
77.0 : Yellow
78.0 : Green
88.0 : Yellow
89.0 : Red
94.0 : Yellow
95.0 : Green

Figure 28. Results (SimStudio)

3.2.5. DEVS#

The results obtained are shown in Figure 29. At each stage of the simulation, the first row indicates the current state just before the output function is called; the time taken by the model to remain in that state (Time advance) and the elapsed Time from the last event. The next row indicates the output just before the model changes state. The last row indicates the next state of the model; the time taken to remain in that state and the elapsed time within that state. There is also information about the performance indices such: as the CPU time for the simulation; the total time that the model was in a particular state as a ratio of the CPU time. For example, the model spent 60% (0.600) of the whole simulation time in the GREEN state. A failure occurs after 72 secs and the traffic light is reset at the 93 sec.

```
DEVS#: C# Open Source of DEVS Formalism, (C) 2005~2007,
http://xsy-csharp.sourceforge.net/DEVSsharp/

The current date is 11/16/2011.
The current time is 2:27:02 AM.
(TrafficLight:GREEN, t s=10.000, t e=10.000)
 --({!TrafficLight.OutputTlight:0}, t_c=10.000)-->
(TrafficLight:YELLOWBR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWBR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=11.000)-->
(TrafficLight:RED, t_s=5.000, t_e=0.000)

(TrafficLight:RED, t s=5.000, t e=5.000)
 --({!TrafficLight.OutputTlight:2}, t_c=16.000)-->
(TrafficLight:YELLOWAR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWAR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=17.000)-->
(TrafficLight:GREEN, t_s=10.000, t_e=0.000)

(TrafficLight:GREEN, t s=10.000, t e=10.000)
 --({!TrafficLight.OutputTlight:0}, t_c=27.000)-->
(TrafficLight:YELLOWBR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWBR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=28.000)-->
(TrafficLight:RED, t_s=5.000, t_e=0.000)

(TrafficLight:RED, t s=5.000, t e=5.000)
 --({!TrafficLight.OutputTlight:2}, t c=33.000)-->
(TrafficLight:YELLOWAR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWAR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=34.000)-->
(TrafficLight:GREEN, t_s=10.000, t_e=0.000)

(TrafficLight:GREEN, t s=10.000, t e=10.000)
 --({!TrafficLight.OutputTlight:0}, t c=44.000)-->
(TrafficLight:YELLOWBR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWBR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=45.000)-->
(TrafficLight:RED, t_s=5.000, t_e=0.000)

(TrafficLight:RED, t_s=5.000, t_e=5.000)
 --({!TrafficLight.OutputTlight:2}, t c=50.000)-->
(TrafficLight:YELLOWAR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWAR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=51.000)-->
(TrafficLight:GREEN, t_s=10.000, t_e=0.000)

(TrafficLight:GREEN, t_s=10.000, t_e=10.000)
 --({!TrafficLight.OutputTlight:0}, t c=61.000)-->
(TrafficLight:YELLOWBR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWBR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=62.000)-->
(TrafficLight:RED, t_s=5.000, t_e=0.000)

(TrafficLight:RED, t_s=5.000, t_e=5.000)
 --({!TrafficLight.OutputTlight:2}, t c=67.000)-->
(TrafficLight:YELLOWAR, t_s=1.000, t_e=0.000)

(TrafficLight:YELLOWAR, t_s=1.000, t_e=1.000)
 --({!TrafficLight.OutputTlight:1}, t_c=68.000)-->
```

```
DEVS#: C# Open Source of DEVS Formalism, (C) 2005~2007,
http://xsy-csharp.sourceforge.net/DEVSsharp/

The current date is 12/4/2011.
The current time is 7:05:08 AM.
(TLS:
   (tlight:GREEN, t_s=10.000, t_e=10.000),
   (Gen:Generating, t_s=16.000, t_e=10.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=10.000)-->
(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=16.000, t_e=10.000)), t_r=1.000


(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=1.000),
   (Gen:Generating, t_s=16.000, t_e=11.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:1}, t_c=11.000)-->
(TLS:
   (tlight:RED, t s=5.000, t e=0.000),
   (Gen:Generating, t_s=16.000, t_e=11.000)), t_r=5.000


(TLS:
   (tlight:RED, t_s=5.000, t_e=5.000),
   (Gen:Generating, t_s=16.000, t_e=16.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=16.000)-->
(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=16.000, t_e=16.000)), t_r=0.000


(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=16.000, t_e=16.000)), t_r=0.000

 --({!TLS.Gen.OutputGen:1,?TLS.tlight.InputGen:1}, t_c=16.000)-->
(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=15.000, t_e=0.000)), t_r=1.000


(TLS:
   (tlight:YELLOWAR, t s=1.000, t e=1.000),
   (Gen:Generating, t_s=15.000, t_e=1.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:0}, t_c=17.000)-->
(TLS:
   (tlight:GREEN, t_s=10.000, t_e=0.000),
   (Gen:Generating, t_s=15.000, t_e=1.000)), t_r=10.000


(TLS:
   (tlight:GREEN, t_s=10.000, t_e=10.000),
   (Gen:Generating, t_s=15.000, t_e=11.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=27.000)-->
(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=15.000, t_e=11.000)), t_r=1.000


(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=1.000),
```

```
   (Gen:Generating, t_s=15.000, t_e=12.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:1}, t_c=28.000)-->
(TLS:
   (tlight:RED, t s=5.000, t e=0.000),
   (Gen:Generating, t_s=15.000, t_e=12.000)), t_r=3.000


(TLS:
   (tlight:RED, t_s=5.000, t_e=3.000),
   (Gen:Generating, t_s=15.000, t_e=15.000)), t_r=0.000

 --({!TLS.Gen.OutputGen:1,?TLS.tlight.InputGen:1}, t_c=31.000)-->
(TLS:
   (tlight:RED, t_s=5.000, t_e=3.000),
   (Gen:Generating, t_s=17.000, t_e=0.000)), t_r=2.000


(TLS:
   (tlight:RED, t_s=5.000, t_e=5.000),
   (Gen:Generating, t_s=17.000, t_e=2.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=33.000)-->
(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=17.000, t_e=2.000)), t_r=1.000


(TLS:
   (tlight:YELLOWAR, t s=1.000, t e=1.000),
   (Gen:Generating, t_s=17.000, t_e=3.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:0}, t_c=34.000)-->
(TLS:
   (tlight:GREEN, t_s=10.000, t_e=0.000),
   (Gen:Generating, t_s=17.000, t_e=3.000)), t_r=10.000


(TLS:
   (tlight:GREEN, t_s=10.000, t_e=10.000),
   (Gen:Generating, t_s=17.000, t_e=13.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=44.000)-->
(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=17.000, t_e=13.000)), t_r=1.000


(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=1.000),
   (Gen:Generating, t_s=17.000, t_e=14.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:1}, t_c=45.000)-->
(TLS:
   (tlight:RED, t s=5.000, t e=0.000),
   (Gen:Generating, t_s=17.000, t_e=14.000)), t_r=3.000


(TLS:
   (tlight:RED, t_s=5.000, t_e=3.000),
   (Gen:Generating, t_s=17.000, t_e=17.000)), t_r=0.000

 --({!TLS.Gen.OutputGen:1,?TLS.tlight.InputGen:1}, t_c=48.000)-->
(TLS:
   (tlight:RED, t_s=5.000, t_e=3.000),
   (Gen:Generating, t_s=24.000, t_e=0.000)), t_r=2.000
```

```
(TLS:
   (tlight:RED, t_s=5.000, t_e=5.000),
   (Gen:Generating, t_s=24.000, t_e=2.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=50.000)-->
(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=24.000, t_e=2.000)), t_r=1.000


(TLS:
   (tlight:YELLOWAR, t s=1.000, t e=1.000),
   (Gen:Generating, t_s=24.000, t_e=3.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:0}, t_c=51.000)-->
(TLS:
   (tlight:GREEN, t_s=10.000, t_e=0.000),
   (Gen:Generating, t_s=24.000, t_e=3.000)), t_r=10.000


(TLS:
   (tlight:GREEN, t_s=10.000, t_e=10.000),
   (Gen:Generating, t_s=24.000, t_e=13.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=61.000)-->
(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=24.000, t_e=13.000)), t_r=1.000


(TLS:
   (tlight:YELLOWBR, t_s=1.000, t_e=1.000),
   (Gen:Generating, t_s=24.000, t_e=14.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:1}, t_c=62.000)-->
(TLS:
   (tlight:RED, t s=5.000, t e=0.000),
   (Gen:Generating, t_s=24.000, t_e=14.000)), t_r=5.000


(TLS:
   (tlight:RED, t_s=5.000, t_e=5.000),
   (Gen:Generating, t_s=24.000, t_e=19.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=67.000)-->
(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=24.000, t_e=19.000)), t_r=1.000


(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=1.000),
   (Gen:Generating, t_s=24.000, t_e=20.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:0}, t_c=68.000)-->
(TLS:
   (tlight:GREEN, t_s=10.000, t_e=0.000),
   (Gen:Generating, t_s=24.000, t_e=20.000)), t_r=4.000


(TLS:
   (tlight:GREEN, t s=10.000, t e=4.000),
   (Gen:Generating, t_s=24.000, t_e=24.000)), t_r=0.000

 --({!TLS.Gen.OutputGen:0,?TLS.tlight.InputGen:0}, t_c=72.000)-->
(TLS:
```

```
   (tlight:BFBLACK, t_s=0.000, t_e=0.000),
   (Gen:Generating, t_s=21.000, t_e=0.000)), t_r=0.000


(TLS:
   (tlight:BFBLACK, t s=0.000, t e=0.000),
   (Gen:Generating, t_s=21.000, t_e=0.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:3}, t_c=72.000)-->
(TLS:
   (tlight:BLACK, t_s=Infinity, t_e=0.000),
   (Gen:Generating, t_s=21.000, t_e=0.000)), t_r=21.000


(TLS:
   (tlight:BLACK, t_s=Infinity, t_e=21.000),
   (Gen:Generating, t_s=21.000, t_e=21.000)), t_r=0.000

 --({!TLS.Gen.OutputGen:1,?TLS.tlight.InputGen:1}, t_c=93.000)-->
(TLS:
   (tlight:AFBLACK, t_s=0.000, t_e=0.000),
   (Gen:Generating, t_s=12.000, t_e=0.000)), t_r=0.000


(TLS:
   (tlight:AFBLACK, t_s=0.000, t_e=0.000),
   (Gen:Generating, t_s=12.000, t_e=0.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:1}, t_c=93.000)-->
(TLS:
   (tlight:RED, t s=5.000, t e=0.000),
   (Gen:Generating, t_s=12.000, t_e=0.000)), t_r=5.000


(TLS:
   (tlight:RED, t_s=5.000, t_e=5.000),
   (Gen:Generating, t_s=12.000, t_e=5.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:2}, t_c=98.000)-->
(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=0.000),
   (Gen:Generating, t_s=12.000, t_e=5.000)), t_r=1.000


(TLS:
   (tlight:YELLOWAR, t_s=1.000, t_e=1.000),
   (Gen:Generating, t_s=12.000, t_e=6.000)), t_r=0.000

 --({!TLS.tlight.OutputTlight:0}, t_c=99.000)-->
(TLS:
   (tlight:GREEN, t_s=10.000, t_e=0.000),
   (Gen:Generating, t_s=12.000, t_e=6.000)), t_r=6.000


============= Performance Indices of Sim. Run 1 =========
CPU Run Time: 00:01:40.2599843
TLS.tlight
GREEN: 0.450
YELLOWBR: 0.040
RED: 0.250
YELLOWAR: 0.050
BFBLACK: 0.000
BLACK: 0.210
AFBLACK: 0.000
```

Figure 29. Results (DEVS#)

The results do differ due to the random generation of failures. For test values and coupling information, CD++ requires a description file to be created which was shown. This requires the modeler to also learn the description language. All other tools require the modeler to define a new DEVS coupled model.

Execution times of all tools were similar. This could be as a result of the short simulation time of 100 secs specified.

DEVS-Suite provides an interactive interface for the whole simulation. The interface provides various options for controlling and monitoring the simulation. DEVS# uses an interactive console menu where the user can configure settings for the whole simulation. Such settings include: Log file creation, Speed of simulation and Test values injection. SimStudio gives the user the option to determine the time for the simulation through the *run()* of the simulator. Similarly, ADEVS requires the user to determine the run time for the simulation through the *nextEventTime()* of the simulator.

Chapter 4. Simulation Interoperability

Simulation interoperability involves the simulation of heterogeneous models developed with various programming languages. These models could be within same formalism like DEVS or across different frameworks (non-DEVS models). The proliferation of DEVS tools has necessitated the standardization of the simulation environment in other to promote model composability as well as the reusability of DEVS and non-DEVS models implemented with different programming languages and platforms. Model reuse helps to avoid the repetitious development of models in different DEVS domains. Advances in computing such as Grid computing and middleware technology have brought about a way for computing resources to be shared and interaction among different applications. CORBA [37] is a middleware concept that enables local and remote applications to be able to interact by building server and client stubs which can then be invoked. HLA [10] [11] is a Dod specification developed for the purpose of integrating different simulation environments. It involves the use of software (Run Time Infrastructure [10]) which is responsible for coordinating the whole simulation including time synchronization and message exchange. All these approaches have helped considerably in achieving simulation interoperability but still pose challenges. There is little documentation on HLA and the use of CORBA has been declining over the years due to the emergence of other concepts like Web services. Grid computing allows harnessing of computing resources but is quite expensive to implement and does not still enable model re-use.

One of the goals of the DEVS community is to have interoperability among different DEVS implementations whereby a model developed with a particular DEVS implementation can be executed with another. This should be the case considering the fact that all simulators implement the same Abstract simulator technique irrespective of the programming language. The challenge lies in the coupling of the model to the particular framework and programming language. These programming languages have libraries which are defined in different ways. So the question arises, how do we decouple these models and Simulators from these programming environments?

There is also the challenge as to how some of these tools address the issue of initialization and coupling information. Some tools define the coupled model in a tool-dependent textual format. Within the DEVS community, the strategies adopted to achieve simulation interoperability and address these issues include:

- Standardizing the Model representation: The model representation should be in a language-independent format;
- Using well known system integration techniques and middleware technologies to interface different simulation environments.

When we talk about simulation interoperability, we refer to two main areas of concern:

- DEVS to DEVS interoperability: This means interoperability among software components and modeling environments that adhere strictly to the DEVS formalism. To achieve interoperability at this level we can employ the two strategies identified earlier.
- DEVS to NON-DEVS interoperability: This is much more complex and involves interoperability among DEVS models and models developed with other formalisms.

Interoperability at this level will require focusing on system integration techniques that help to hide as much as possible the details of the underlying framework.

4.1. DEVS standard and interoperability in modeling

In other to aid model re-use and composability, a standard model representation needs to be defined. This representation will be, language-independent, portable and will aid resource sharing in the DEVS community. Once defined, we then face the challenge of integrating it into existing frameworks taking into cognizance the fact that existing DEVS implementations define their local models in different ways which the simulators have been designed to execute. A way needs to be identified to enable the transformation of the standardized model to the local models. Another challenge lies with the fact that some DEVS implementations use textual formats to define their models. This also has to be taking into account in the model transformations. Several of the attempts made at achieving all these will be discussed in the subsequent sections. It is also important to note that even though several attempts have been made, there is still no standard representation of a DEVS model.

4.1.1. Shared Abstract Model

It involves defining an Abstract Model Interface (AMI) for models written in different programming languages. A simulator during simulation usually invokes the operations of the model but with the Shared abstract model [23], the invocations are made to the Abstract model using middle-ware technology like CORBA. This helps to create location transparency for remote models i.e. some of the sub-models can be remotely invoked but to the simulator, these invocations are local.

A model proxy needs to be developed for a simulator so that the invocations of a model are translated to invocations of the Abstract Model Interface. This model proxy only needs to be developed once for each DEVS implementation. Model implementations can be developed to directly adhere to the AMI or use a model adapter that translates the invocations of the AMI to invocations of the model. The concept is shown in Figure 30.

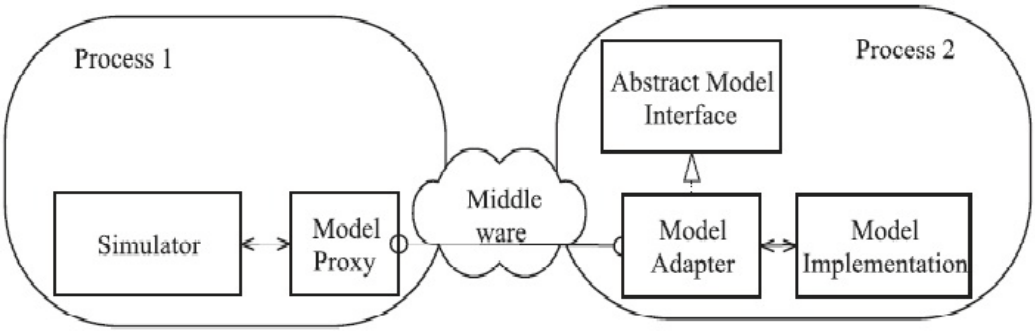

Figure 30. Shared Abstract Model [23]

4.1.2. DEVSML

It is an XML representation of a DEVS model. It is based on JAVASML [39] research by Vladimir for DEVS meta language. Hence, it lends its use to models developed in JAVA although efforts are underway to extend its capabilities to the C++ domain.

DEVSML concept follows a user developing a model in a DEVS environment with a particular programming language. The model is then transformed to the DEVSML representation with enough information or metadata about the model. This information will be useful during the simulation process and the whole transformation process is automated. The DEVSML model can be integrated in a DEVS architecture that deploys simulation services as web services. This was demonstrated with DEVS/SOA. In such an environment, the DEVSML descriptions can be packaged into SOAP (Simple Object Access Protocol) messages and transported to any remote location where it can be simulated or stored for reuse. The appropriate simulator is called based on information contained in the XML description of the model.

4.1.3. DEVS Markup Language

DEVS Markup Language is a platform and language-independent format for describing and sharing DEVS models and is based on XML.

DEVS Markup Language aims to tackle some of the challenges faced with standardizing the DEVS model. These challenges are:

- How do we create a standard representation of a model that is not language dependent?

- How do we represent the model dynamics and still maintain the genericness and flexibility of the model representation?
- How do we deal with parameterization and initialization of the model?

DEVS Markup Language attempts to address the first problem by defining three **t**ypes.

- Intrinsic types (e.g. integer, float)
- Custom types but not language dependent
- Language-dependent types

The first two types provide a language independent way of representing any part of a model i.e. a type like an integer is common to all object oriented languages. The third type is language dependent. Hence during modeling, types in other programming languages that can perform the functions of this language-dependent type need to be specified.

Second problem of model dynamics can be addressed by representing a major part of the model's behavior in a generic way and introducing code snippets for language specific parts. This creates a kind of semi-generic language.

Parameterization and initialization can be handled by introducing a function for each of them separately in each atomic model. With this, models can be coded in one particular language and simulated with another language.

4.2. DEVS standard and interoperability in simulation

In other to carry out distributed simulation, the various execution engines running on different machines need to be integrated to work in concert in executing models. The middleware layer of computer systems provides a way to achieve this. It allows software components implemented at a high level language and running on different machines to communicate with each other. Research and development in this area has lead to various concepts and technologies like CORBA, DCOM and Web services.

We are fully aware of the fact that several DEVS packages have been developed with various programming languages and that some of them are platform specific. Middleware communication can provide a way to integrate these diverse DEVS implementations. In other to achieve this, several issues need to be addressed:

- A particular middleware technology needs to be identified for use in integrating these software components. There are several middleware products and concepts and they each go about communication in a unique way.

- The simulation protocol needs to be standardized and adapted to the middleware technology adopted. Message formats need to be identified and means of exchange them also need to be established.
- A standard interface for the simulator needs to be defined.

Web services framework has emerged as the prevailing technology for systems integration within the DEVS community over the last few years as a result of its increasing popularity in computing and waning interest in other middleware technologies like CORBA and DCOM. The subsequent sections discuss approaches that have used web services to provide an interoperable environment for distributed simulation of heterogeneous models. These include: DEVS/SOA; RESTFUL-CD++. Other attempts outside web services will be discussed such as DEVS/HLA[28] and PlugSim [29].

4.2.1. DEVS/SOA

It is a modeling and simulation environment that provides distributed simulation of heterogeneous models. It maintains the design principles of a service oriented architecture and its architectural framework originates from the proposed DEVS standard of the working group of the Simulation Interoperability Standard organization (SISO) which defines standard interfaces for the Model and Simulator. It even goes further as to incorporate a third layer or interface called the Coordinator interface.

There are three layers as depicted Figure 31, the Client application, Coordinator and the Simulator. The Client application which has a web service component has the model specification in DEVSML i.e. the XML description of the coupled model and its component models. This description among other things specifies the name of the coupled model, the address of a host server for the coupled model (a server that offers Coordinator services), input and output ports of the component models and their connections, class of each atomic model and the host of the atomic model (a server offering simulation services and also has access to a class repository of the model). This XML description is supplied to the coordinator. The coordinator which also has a web service component extracts the address of the hosts of the atomic models and establishes connections with them. It then sends the DEVSML model to each of them and waits for them to each instantiate a class of the atomic models they are responsible for after which simulation can begin.

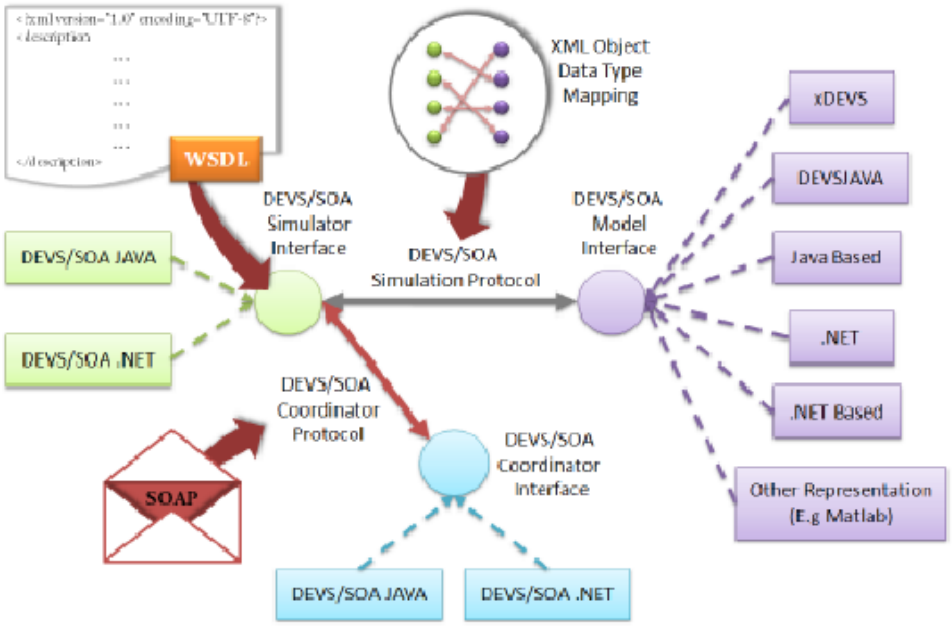

Figure 31. DEVS/SOA

Knowing that there is no standard interface for models, a DEVS common interface was defined to integrate several modeling and simulation frameworks. It includes atomic DEVS operations and includes an XML to data type binding that is responsible for converting messages originating from a model to XML and vice versa. SOAP is the medium of communication between the application layer, coordinator and the simulator layers. There are currently two supported environments, JAVA based models and .NET based models.

4.2.2. RESTFUL-CD++

Some of the criticisms of SOAP based Web services are: the verbosity associated with defining the service stubs and the time taken for the stubs to marshal and unmarshal the parameters. This has lead to the popularity of **REST** (Representational State Transfer) as a way of implementing web services. REST is wholly dependent on HTTP for message exchange and involves manipulation of resources using HTTP methods. It is stateless and exposes everything as a resource. Its simplicity and flexibility can be leveraged to provide a way for distributed simulation of heterogeneous models.

RESTFUL-CD++ is an interoperable environment based on REST principles that exposes services or resources with URIs and allows manipulation of these services via HTTP methods. Although it is wholly dependent on HTTP, it can still consume services from a SOAP based environment. The design specifies three main URIs from which everything originates:

- /admin: For administtrative activities like user account creation
- /util: Utilities that might be needed for client programming
- /sim: Consist of all resources needed to carry out simulation.

The modeler only needs to manipulate these resources using HTTP requests.

4.2.3. DEVS simulation protocol standard

So far, we've mentioned efforts at achieving a DEVS standard through the model and simulator interfaces but not the simulator protocol responsible for exchanging simulation messages. The question arises as to how we can standardize message exchange. How can we ensure that messages originating from a particular DEVS domain can be interpreted in another domain? An attempt has been made to address this problem by proposing a way to standardize the simulation protocol [1]. Their approach is based on a service-oriented architecture and proposes a DEVS wrapper for each DEVS implementation. The DEVS wrapper will be responsible for:

- Converting incoming simulation messages to domain specific messages
- Converting outgoing simulation messages to other domains
- Route incoming messages to the proper ports of the models within the domain

Each DEVS domain should deploy a pack of services (createNewSession, stopSimulation, closeSession and receiveDEVSML) necessary for coordinating the simulation. The approach makes the assumption that each DEVS domain is responsible for simulating its own model and a particular DEVS domain is designated as the Master Coordinator for the simulation. This Master coordinator is responsible for simulating the whole coupled model and interfaces with Slave coordinators that are responsible for simulating the component coupled models.

Simulation messages which in an ideal case will be exchanged among different DEVS domains are passed as XML documents via SOAP. These messages responsible for driving the simulation should specify:

- SessionID
- Simulation Message type(e.g Init message)
- Next change Time (Used to specify the time of the next internal change in the model, The minimum is sent to the Root coordinator)
- Message sending time
- Source model
- Destination model
- Source port
- Destination Port
- Value
- isFromSlaveDomain (used to determine origin of message i.e slave or master domain)

All these fields capture enough information to enable distributed simulation of the models. As identified earlier, all messages are exchanged in XML format so that the communication framework is not dependent on the underlining architecture for simulation.

4.2.4. DEVS/HLA

It is a modelling and simulation environment that complies with the High Level Architecture standard. The underlying framework includes DEVS-C++ mapped to a C++ version of the Defence Modeling Simulation Organization (DMSO) Run Time Infrastructure (RTI).

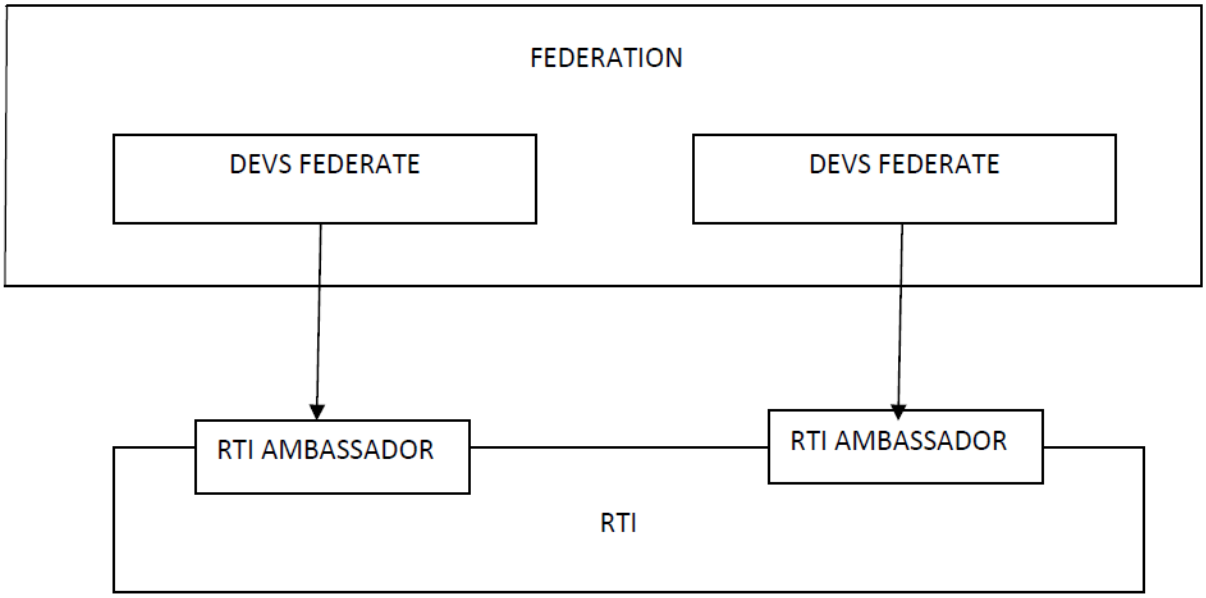

Figure 32. DEVS/HLA

HLA allows combination of computer simulations into a larger one. It is an object-oriented approach used for distributed computer simulations. It specifies set of Rules, Federates, RTI and an Object Model Template.

A Federate is an individual compliant simulator. A combination of federates working in concert is termed a Federation. With DEVS/HLA, the simulators are adapted to implement an HLA interface specification. This adaptation involves mapping DEVS coupling information to HLA interactions. The RTI is middleware software that coordinates the whole simulation. It manages data exchanges, time synchronization and federates interactions. DEVS/HLA also provides integration of other DEVS domains by implementing the DEVS/HLA interfaces provided.

4.2.5. PlugSim

PlugSim is a modeling and simulation environment that uses plug-in method to integrate models of various frameworks. It provides two interfaces: DEVS and non-DEVS model interface. The DEVS model interface specifies the DEVS atomic model specification and the non-DEVS model interface enables distributed simulation of models.

The architecture of the PlugSim is specifies two parts: the changeable part and the non changeable part. The changeable part is the model. Users can change the simulation models with other models. The non-changeable part includes: simulation algorithms; data exchange and Time synchronization. The framework is depicted in Figure 33.

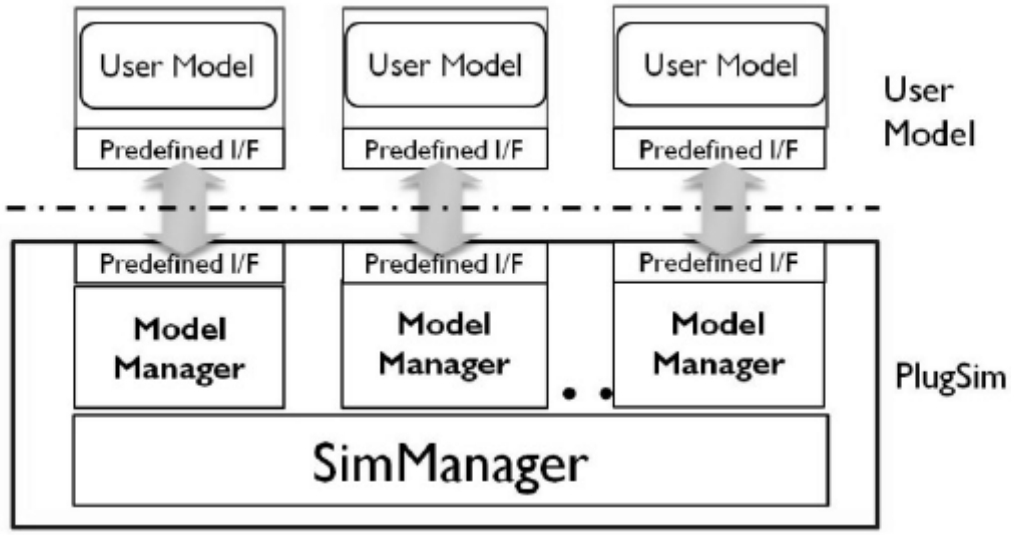

Figure 33. PlugSim [29]

The SimManager is responsible for managing data exchange and time synchronization. It connects all the Model Mangers and provides them with routing information. The Model manager is connected to a single model and is responsible for simulating it using DEVS simulation algorithms. It gets routing information from the SimManager necessary for data exchange with other model managers.

The goal of the PlugSim is to provide distributed simulation of DEVS and non DEVS models which implement the provided interfaces.

Chapter 5. Virtualization

Virtualization is the process of making something virtual. In computing, it is a framework for dividing the resources of a computer into multiple environments. Ever since the M44/44X project by IBM in the 1960s, virtualization has grown in importance in computing and has lead to a range of technologies and products: PVM [41] is a software package that allows the aggregation of Unix or Windows systems for parallel computing; VirtualBox [40], also a software package that supports the execution of multiple operating systems on a single operating system.

5.1. Types of Virtualization

In a particular environment virtualization could involve dividing a single resource into multiple utilization domains like in the case of partitioning a hard disk while in some other environment; it could involve the aggregation of resources as is the case with PVM. Virtualization has been applied to various areas of computing and system architecture some of which include: Hardware; Software; Network; Memory and Storage.

5.1.1. Hardware Virtualization

This is virtualization at the hardware level. It could be **Full**, **Partial** or **Hardware-assisted**. In full virtualization, the Virtual machine simulates the underlying hardware completely thus allowing multiple operating systems to run parallel to each other but in an isolated environment.

With Partial virtualization, there isn't complete simulation of the underlying hardware. Multiple operating systems cannot run but it supports multiple applications running parallel to each other. Hardware-assisted virtualization involves the processor providing architectural support to the hypervisor.

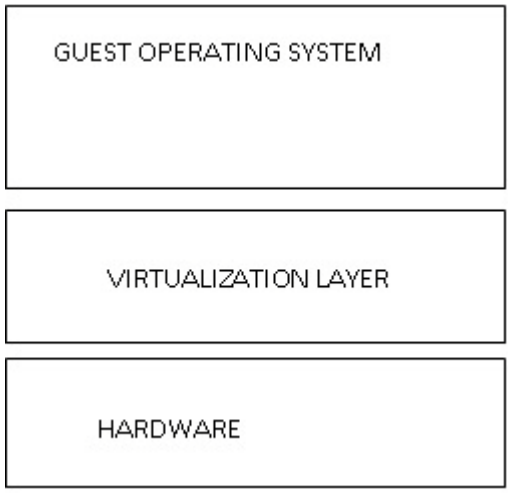

Figure 34. Hardware Virtualization

5.1.2. Software Virtualization

Software virtualization can be further divided into **Operating system level** virtualization and **Application** virtualization. With OS level virtualization, all VMs share the host operating system Kernel. The Virtual machines exist on top of the host OS.

Application Virtualization involves creating isolated virtual environments for the execution of applications. Guest operating systems are not supported; only programs or processes are supported. This enables the portability of applications.

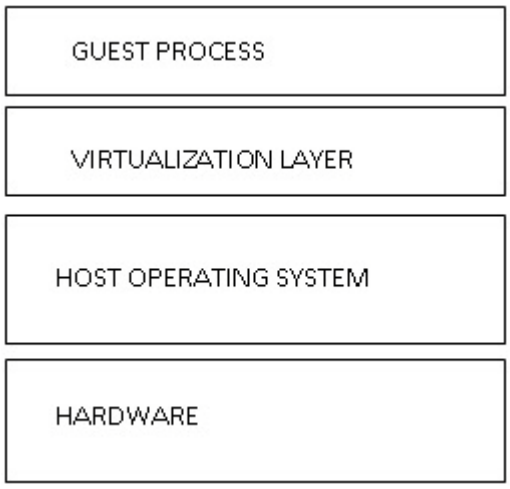

Figure 35. Software Virtualization

5.1.3. Memory virtualization

Programs running on a system occasionally require address space which is larger than the physical memory installed. In other to address this problem, the concept of virtual memory arose where by physical memory can be decoupled from the systems and integrated into a shared pool of memory available to networked systems and the applications running on them.

5.1.4. Storage Virtualization

Similar to memory virtualization and involves decoupling physical storage devices from individual systems and aggregating them to form a single storage unit available over a network.

5.1.5. Network Virtualization

This involves dividing the available bandwidth into multiple utilization domains. Each available bandwidth space is isolated from the others and this enables the containment of congestion in a particular space. This is useful in network management and congestion control.

5.2. System Architecture

Computer systems are divided into two areas: Software and Hardware. The software consists of the Application or programs, libraries and the Operating system while the hardware consists of the Input/output devices, CPU and Physical Memory. There are three layers in operating systems which are important to the concept of virtual machines. These are: Application Programming Interface, Application binary Interface and Instruction Set Architecture.

5.2.1. Application Programming Interface

This is an interface or set of rules and procedures provided by a program to another in other to carry out services on its behalf. In operating systems, it is a high level language interface that provides a program access to the resources of a system like the file system and the registry. This is usually provided to an application developer so that application can make system-library calls.

5.2.2. Application Binary Interface

This is a low level interface that provides a program access to system resources using the user ISA. It differs from API in that it is an interface at the binary level and not at the source level.

5.2.3. Instruction Set Architecture

It is the interface between the software and the hardware. It is the part of the processor that is visible to the programmer. It includes specifications for op-codes, instruction formats, addressing modes and registers.

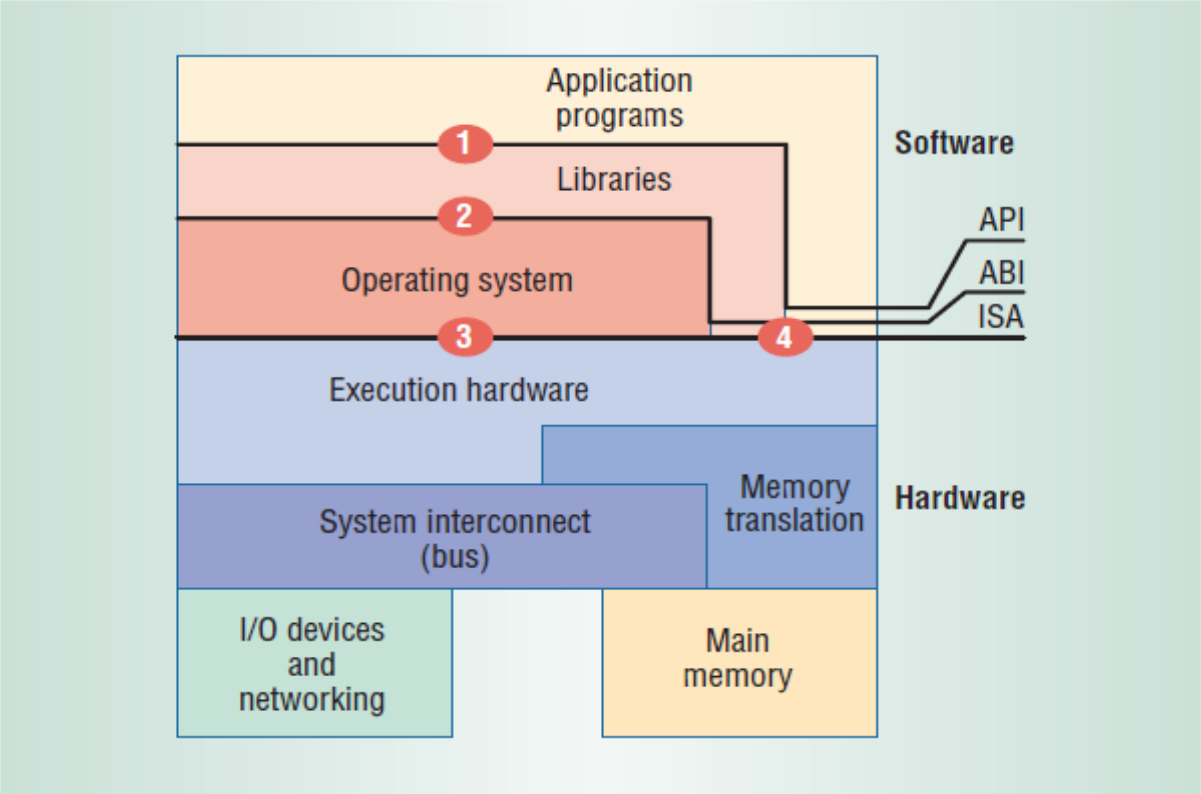

Figure 36. System Architecture [31]

A virtual machine is an execution environment that gives the illusion of a real system to individual processes or operating systems. It provides a means of executing instructions meant for a particular system on another system thus, we then have multiple execution domains on a single physical system. It could involve some form of emulation or virtualization software. Some of the benefits of VMs include:

- Software portability like in the case of the JAVA virtual machine.
- Server consolidation in Enterprises. Instead of having multiple under-utilized physical servers running, we could have multiple VMs acting as servers running on a single machine e.g. VMware server.
- Can provide different hardware configurations for different operating systems.
- VMs can be useful in software testing and research.
- Can be used to create isolated environments for the execution of un-trusted applications.

There are two categories of virtual machines: Process and System virtual machines.

5.2.4. Process Virtual Machines

It is a virtual platform that handles a single process. It runs as an application on the host operating system and provides a platform independent programming environment. The process that runs on the virtual machine is termed the guest and the underlying platform, the host. The virtualization software that implements the VM is called the Runtime and is usually implemented at the ABI or API level of system architecture. It emulates the user ISA and system or library calls i.e., when a process makes a system call, the process VM intercepts this request and handles it on its behalf.

One of the advantages of a process VM is portability of programs. It provides a high level language abstraction. The programmer does not need to worry about the underlying platform during development and is only required to write the code once. The VM will be responsible for executing the code on different platforms. Another advantage of a process VM is security of the underlying operating system. As a result of the high level language abstraction provided by the VM, the program is unable to cause harm to the underlying operating system.

The JAVA virtual machine (JVM) is an example of a process VM. It enables JAVA programs written and compiled on a particular platform to be executed on another. The JAVA compiler usually compiles JAVA code to Byte-code. Byte-code is an intermediary code between source code and native code or machine code. It is platform independent and can be ported to any location and executed. The JVM is then responsible for executing the byte-code. It consists of an interpreter that uses Just-in-time compilation which involves translating the byte-code to machine code and caching the translated code for future references. The JVM has been implemented for different platforms and that is why JAVA programs can run on different operating systems like Linux and Windows.

The Common language runtime (CLR) is also a process VM and is central to the .NET framework. It also uses Just-in-time compilation to translate CIL code (byte-code of .NET languages) to machine code which can be executed by the CPU. The .NET framework is only available in the Microsoft Windows environment.

5.2.5. System Virtual Machines

This is a virtual platform that enables multiple operating systems to run on a computer. The concept original surfaced in the 1960's when computing resources where relatively expensive and large in size. The idea is to share hardware resources among several operating systems. The operating system running on the VM is termed the guest while the underlying operating system is called the Host. The virtualization software is called the Virtual Machine Monitor (VMM) or hypervisor. It is a layer of software that provides virtualization of hardware resources to multiple instances of virtual machines. The VMM emulates the hardware ISA thus enabling the guest operating system to execute a different ISA. In other words, when the guest OS requests for hardware resources (Input / Output resources) the VMM intercepts such requests and makes the requests on behalf of the guest OS; this way the guest OS and the various processes are under the control of the VMM.

System VMs can be implemented on the bare hardware. In this case, the VMM runs on the hardware directly and the VM on top. They can also be implemented with an operating system hosting the VMM. The VMM runs on the operating system and the VM on top. One of the advantages of System VMs is the isolation of the several instances of the VMs. When a particular process or operating system fails, it does not affect the other operating systems. The VMM provides hardware resource transparency and an isolated execution environment. An example of this type of VM is VirtualBox.

In the world of computing, virtualization is increasing in importance and has helped to provide interoperability, scalability and portability of applications.

Chapter 6. Architecture for DEVS interoperability

DEVS framework has been implemented as software packages with several programming languages. The challenge has been how to get these different software components to work in concert to simulate models (DEVS and non-DEVS). In chapter four, we discussed some of the attempts at achieving this. DEVS/SOA and RESTFUL-CD++ adopted service oriented architecture to integrate interoperable simulators. These environments deployed simulators as web services that could be consumed on demand. One problem with this technique is the heavy reliance on the internet. Network connections cannot guarantee 100% up-time as a result of issues such as congestion and network equipment failures. We have adopted virtualization concepts to provide simulation interoperability. Recall that interoperability is dealing with heterogeneous simulation codes written in different programming languages and on different platforms.

6.1. Virtualization Architecture

This approach attempts to unify three concepts: DEVS framework; Simulation Interoperability; and Virtualization to provide a means of integrating different DEVS implementations such as SimStudio, DEVSJAVA and ADEVS. With our design, we set out to accomplish three goals:

- Provide an environment for the execution of models written in various programming languages i.e. The Virtual Machine should be able to compile and execute source codes of various programming languages like JAVA, C++ and C#
- Provide interoperability among various DEVS implementations
- The virtual machine should be portable across multiple platforms and operating systems.

To achieve these objectives, we followed three design angles:

- A process virtual machine to tackle (1).
- Employ middleware communication to achieve (2).
- A system virtual machine to tackle and (3).

Process VM

Survey of existing DEVS tools shows that five languages need integration. These are JAVA, C++, C#, Smalltalk and Python. To provide platform independence and integration of these languages, we envisage the use of the JVM as depicted in Figure 37.

The JVM is a virtual machine that executes Java bytecode. It has been implemented on multiple platforms and operating systems. In keeping with our design goal of (1) above, we can leverage its capabilities to integrate different programming languages. To do this, translators will be required for translating source code of other programming languages to Java bytecode. Both

Python and Smalltalk already have support for Java. Jython is an implementation of the Python language written in JAVA and enables Python programs to import classes in Java. Redline Smalltalk is an implementation of Smalltalk in Java and allows Java cllases to be called within a Smalltalk program.

The challenging aspect of this design is the translators for C++ and the Common Intermediate Language (CIL). A lot of the class libraries in C++ do not have their equivalent in Java and will have to be implemented. We also the issues associated with pointer arithmetic of C++ and all the memory management is explicitly done.

CIL to Java bytecode also presents a challenge. The opcodes defined within the CIL will have to be mapped to their Java equivalents. This is possible owing to the fact that both JVM and CLR are stack-based machines although not all opcodes within the CIL have their corresponding implementations in Java.

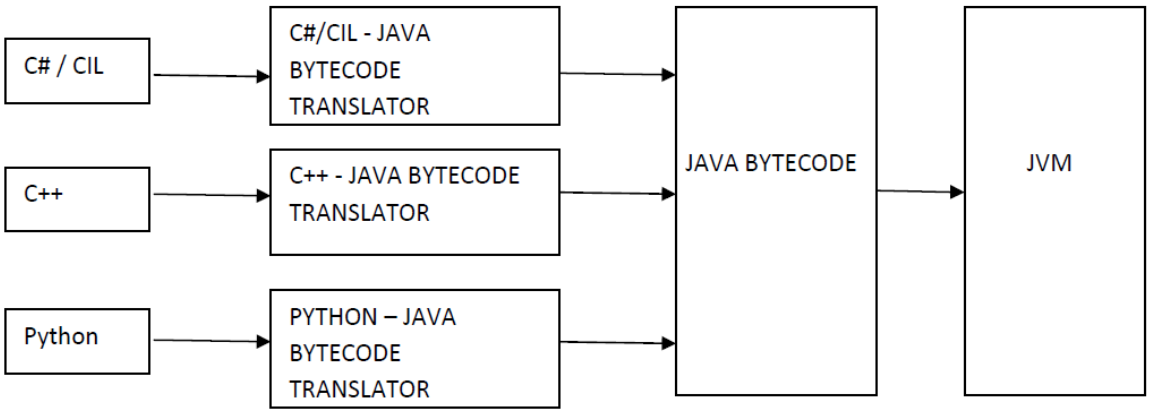

Figure 37. Process Virtual Machine

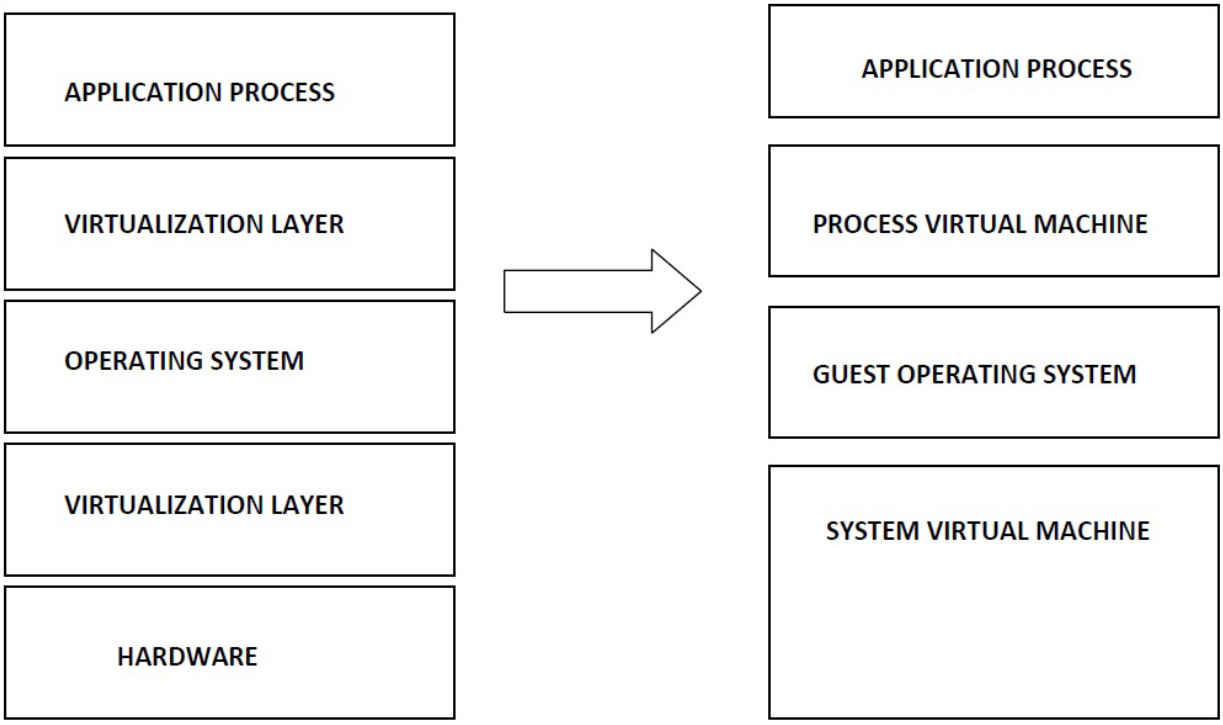

Figure 38. Virtualization Architecture

6.2. Simulation Interoperability

To provide interoperability among various DEVS implementations, we adopted the Shared Abstract Model concept. It is a recently proposed approach to standardize a DEVS model. A simulator during simulation usually invokes the operations of a model but with the SAM concept, the invocations are made to the Abstract model using middle-ware technology like CORBA. The middleware layer of a computer system provides a means for communication and interaction between programs running on same machine or different machine. There are two ways to apply the SAM concept:

- Carry out simulation of a model over a network of simulators running on different machines. The problem with this approach is the reliance on the availability of the network connections although it provides a means for distributed simulation
- Have all the simulators running on the same machine working in concert to simulate the model. The problem with this approach is that not all DEVS implementations are platform independent as a result of the programming languages used to develop them. For example DEVS# is an open source library developed with C# which is tied to the .NET framework of Microsoft operating systems.

To circumvent these issues and to enable a model to be simulated anywhere irrespective of the DEVS domain, we envisage the use of a process virtual machine discussed above to provide an

environment in which all these simulators can co-exist and run irrespective of the programming languages used. The virtual machine will responsible for providing platform independence and portability for all these simulators.

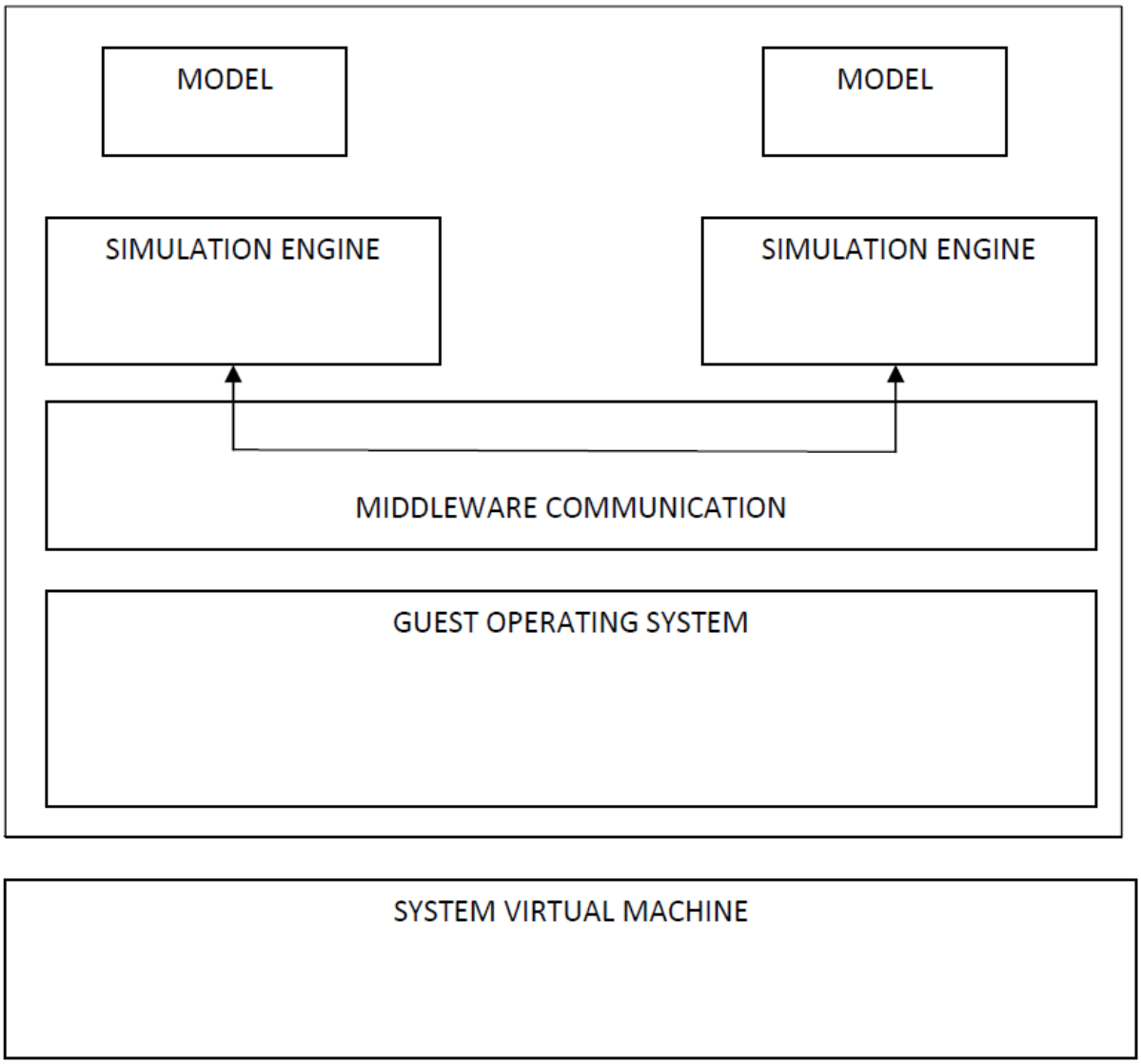

Figure 39. Relationship between Virtualization, DEVS framework and Simulation Interoperability

6.3. Road Network Model

To illustrate our concept, we have developed a Road Network model of a Crossroad. This example is only conceptual and is used to demonstrate how the simulators will interact. The Road network model is a coupled model with subcomponents: Traffic Light; Generator (generator of

Cars); Road; Platform. The design is shown in Figure 40. Each subcomponent will be realized with a particular DEVS tool and the architecture for coupling all these diverse software components is shown in Figure 41.

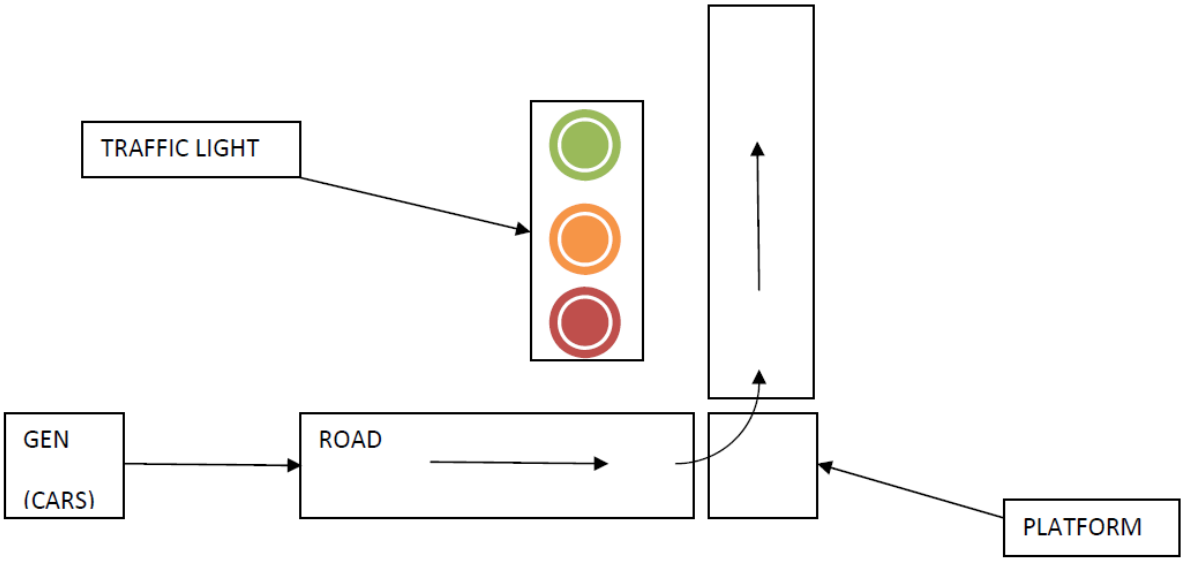

Figure 40. Road network model

6.4. SimStudio Integration

Individual processes for the different DEVS implementations will be started on theguest operating system. The various source codes specified in different programming languages will be compiled and executed by the Process VM. SimStudio will be in charge of simulating the coupled model. The others models will be simulated as follows:

- Traffic Light – CD++
- Road – DEVSJAVA
- Gen- DEVS#
- Platform – ADEVS

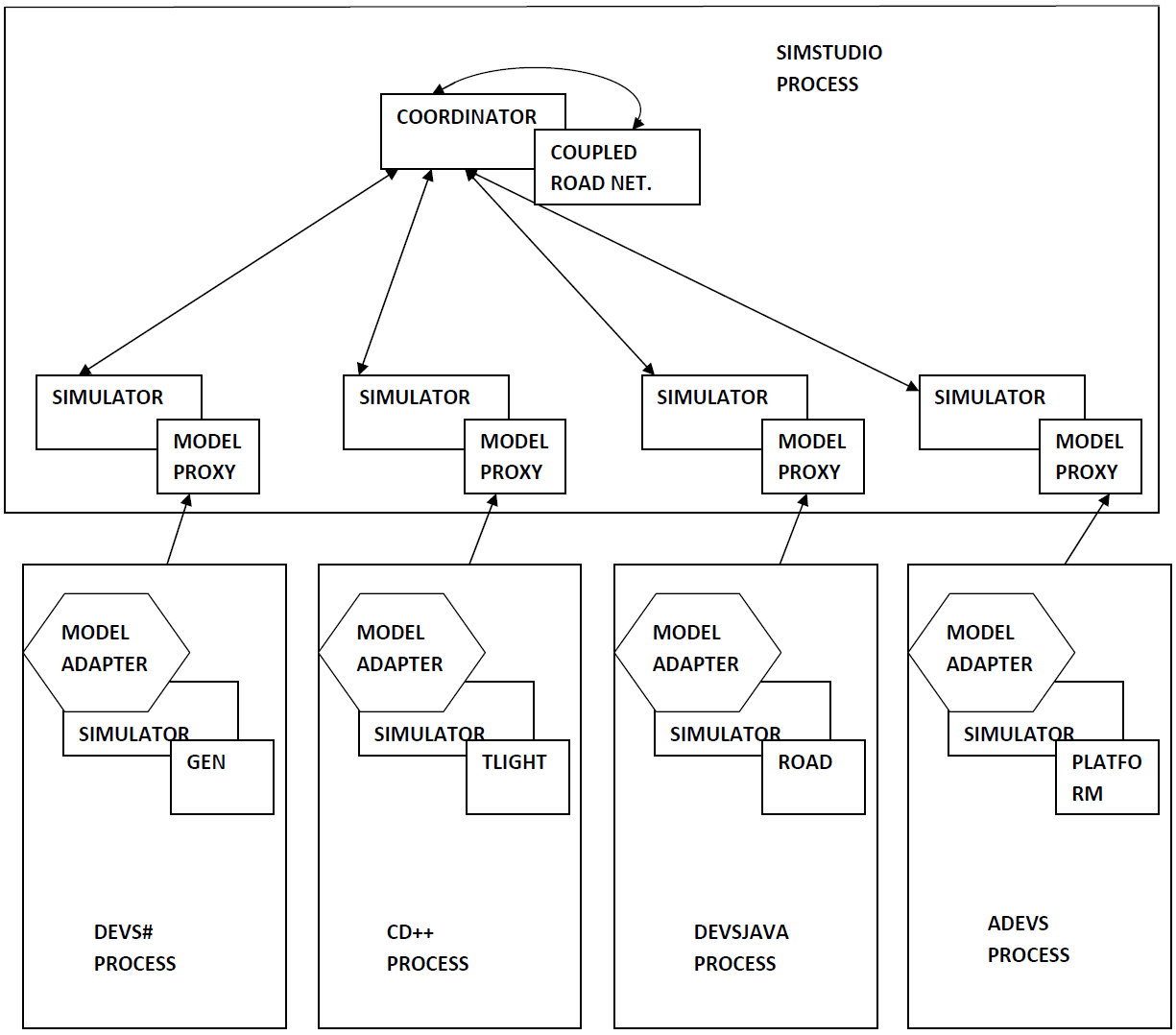

Figure 41. Coupling architecture for Road network model

6.5. Discussion

In other to carry out the simulation of the Road Network, model proxies and model adapters will have to be developed for each DEVS implementation. The model proxies will be responsible for translating the invocation of the methods of the models to the invocations of the Abstract model. Although this is straight forward, handling of message contents may be a challenge. An additional program will have to be written to handle message synchronization.

Although this approach is only conceptual, it provides a way for different DEVS tools to exist and be run within same space. This will obviate the need to have a reliable network connection to integrate different DEVS implementations. This was the case with DEVS/SOA.

Conclusion

With this approach, diverse DEVS implementations will be able to co-exist within same domain and exchange information. Our aim was to come up with a conceptual model that uses well known technologies and recently proposed concepts to provide a simulation environment for DEVS and non-DEVS models developed in various programming languages. The design lends itself to well known concepts such as System and process virtualization as well as simulation interoperability principles. A message translator needs to be developed to ease the burden of having the modeler to write a program to handle message contents been transferred to another domain.

Chapter 7. Conclusion

We introduced the DEVS framework and Simulation interoperability. We also identified the challenges associated with simulation interoperability and some of the efforts made at achieving it. We have presented a design that aims to incorporate DEVS framework, Simulation interoperability and Virtualization. This design is only prospective but we have leveraged well known virtualization concepts to ensure software portability of the models designed with several programming languages. This way models and simulators can be executed anywhere. This is just the architectural framework but we hope to implement it in the future.

Future work includes the following:

- Designing the Process and System Virtual Machines.
- Message Translator should be developed for the Shared Abstract model.
- A way of incorporating parallel and distributed simulation of models on these Virtual Machines should also be considered.

References

[1] Zeigler, B.; Praehofer, H; Kim, T. 2000. "**Theory of Modeling and Simulation**". 2nd Edition, Academic Press.

[2] Christen, G., A. Dobniewski, and G. Wainer. 2004. "**Modeling state-based DEVS models CD++**". Proceedings of MGA, Advanced Simulation Technologies Conference 2004, Arlington, VA, USA.

[3] Kidisyuk, K., and G. Wainer. 2007. "**CD++Modeler: A graphical viewer for DEVS models**". Technical report SCE-017, Ottawa, ON, Canada.

[4] Matias, B.; G. Wainer; R. Castro; 2010. "**Advanced IDE for Modeling and Simulation of Discrete Event Systems**". Proceedings of 2010 Symposium on Theory of Modeling and Simulation, DEVS'10. Orlando, FL. 2010.

[5] Nutaro, J. ADEVS. URL: http://www.ornl.gov/~1qn/adevs/index.html. Accessed: November 1, 2010.

[6] Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1996. "**Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally**". Transactions of the SCS 13 (3): 135–154.

[7] Zeigler, B., Y. Moon, D. Kim, and D. Kim. 1996. "**DEVS-C++: A high performance modeling and simulation environment**". Proceedings of the 29th Hawaii International Conference on System Sciences, Honolulu.

[8] Zeigler, B. P. 1990. "**Object-oriented simulation with hierarchical, modular models: Intelligent agents and endomorphic systems**". Boston: Academic Press.

[9] Zeigler, B., and D. Kim. 1995. "**Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control**". Technical report, Department of Electrical and Computer Engineering, University of Arizona.

[10] Sarjoughian, H. S., and B. P. Zeigler. 2000. "**DEVS and HLA: Complementary paradigms for M&S?**" Transactions of the SCS 17:187–197.

[11] Zeigler, B. P. 1999. "**Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions**". Simulation Interoperability Workshop, Orlando, FL.

[12] IEEE Std 1516.1-2000. 2001. IEEE standard for modeling and simulation. "**High level architecture (HLA)—Federate interface specification**". IEEE Std 1516.1-2000: 1–467.

[13] Sarjoughian, H. S., and B. P. Zeigler. 1998. "**DEVSJAVA: Basis for a DEVS-based collaborative M&S environment**". Proceedings of SCS International Conference on Web-Based Modeling and Simulation, San Diego, CA.

[14] Sungung Kim, Hessam S. Sarjoughian and Vignesh Elamvazhuthi. 2009. "**DEVS-Suite: A Simulator supporting Visual Experimentation Design and Behavior monitoring**". Proceedings from the Spring Simulation Multiconference.

[15] Kim, T. G. 1994. DEVSIM++ user's manual. CORE Lab, EE Dept, KAIST, Taejon, Korea.

[16] DÃ¡vila, J., and M. UzcÃ¡gegui. 2000. "**GALATEA: A multi-agent, simulation platform**". Proceedings of International Conference on Modeling, Simulation and Neural Networks, MÃ©rida, Venezuela.

[17] Himmelspach, J., and A. Uhrmacher. 2004. "**A component-based simulation layer for JAMES**". Proceedings of 18th Workshop on Parallel and Distributed Simulation (PADS), Kufstein, Austria, 115–122.

[18] Filippi, J. B., and P. Bisgambiglia. 2004. "**JDEVS: An implementation of a DEVS based formal framework**". Environmental Modeling and Software 19:261–274.

[19] Bolduc, J. S., and H. Vangheluwe. 2001. "**The modeling and simulation package PythonDEVS for classical hierarchical DEVS**". Technical report MSDL-TR-2001-01, McGill University.

[20]. de Lara, J., and H. Vangheluwe. 2002. "**AToM3: A tool for multi-formalism and meta-modeling**". Proceedings of Fundamental Approaches to Software Engineering, 5th International; Lecture Notes in Computer Science, 174–188.

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

[22] Traore, M. K. 2009. "**A Graphical Notation for DEVS**". Proceedings from the Spring Simulation Multiconference.

[23] Thomas Wutzler and Hessam S. Sarjoughian 2007. "**Interoperability among Parallel DEVS simulators and Models implemented in Multiple programming languages**". Society for Computer Simulation International.

[24] Saurabh Mittal, José Luis risco-Martin and Bernard P. Ziegler 2007. "**DEVSML: automating DEVS execution over SOA towards transparent simulators**". Proceedings of the spring simulation multiconference.

[25] Luc Touraille, Mamadou K. Traoré and David R. C. Hill 2009. "**A mark-up language for the storage, retrival, sharing and interoperability of DEVS models**". Proceedings of the Spring Simulation Multiconference.

[26] Alejandro Moreno, José L. Risco-Martín, Eva Besada, Saurabh Mittal and Joaquín Aranda 2009. "**DEVS/SOA: Towards DEVS interoperability in Distributed M&S**". Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications

[27] Khaldoon Al-Zoubi and Gabriel Wainer 2009. "**Using REST Web-Services Architecture for Distributed Simulation**". Proceedings of the ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation

[28] Bernard P. Ziegler, George Ball, Hyup Cho, J.S. Lee and Hessam S. Sarjoughian 1998. "**The DEVS/HLA Distributed simulation environment and its support for predictive filtering**". A deliverable in fulfillment of Advance SimulationTechnology Thrust, DARPA contract N6133997k-0007.

[29] Jang Won Bae and Tag Gon Kim 2010. "**A DEVS based plug-in framework for interoperability of simulators**". Proceedings of the Spring Simulation Multiconference.

[30] Khaldoon Al-Zoubi and Gabriel Wainer 2008. "**Interfacing and Coordination for a DEVS simulation protocol standard**". Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications.

[31] James E. Smith and Ravi Nair 2005. Computer 38. DOI: 10.1109/MC.2005.173

[32] "**Modeling and Simulation using DEVS#**," Accessed June 23, 2011. http://xsy-csharp.sourceforge.net/Devsharp.

[33] B. Zeigler, S. Mittal, X. Hu. 2008. "**Towards a formal standard for interoperability in M&S/System of Systems Integration**," GMU-AFCEA Symposium on critical issues in C41.

[34] Gudgen Martin, Hadley Marc, Mendelsohn Noah, Moreau Jean-Jacques, Nielson Henrik. "**SOAP version 1.2 Part 1: Messaging framework**" Accessed November 2, 2011. Available via http://www.w3.org/TR/soap12-part1.

[35] T. Lindholn and F. Yellin. 1999. "**The JAVA virtual machine specification**", second edition, Addison-wesley.

[36] D. Box. 2002."**Essential .NET, volume 1: The Common Language runtime**", Addison-wesley.

[37] Gerald Brose, Andreas Vogel, Keith duddy. 2001."**JAVA programming with CORBA: Advanced Techniques for building Distributed Applications**", Third Edition, Wiley computer publishing.

[38] Michael Champion, Chris Ferris, Eric Newcomer, David Orchard.2002. "**web Services Architecture**". W3C

[39] Janousek, V., Polasek, P., Slavicek, P., 2006. "**Towards DEVS meta language**". In ISC proceedings zwinjinaarde, BE, 69 -73, ISBN-90-77381-26-0.

[40] "VirtualBox". Accessed November 2, 2011. http://en.wikipedia/wiki/VirtualBox.

[41] "Parallel Virtual Mchine". Accessed November 2, 2011. http://en.wikipedia/wiki/Parallel_Virtual_Machine.