11-27-2007

A Contextualized Web-Based Learning Environments for DEVS Models

Inthira Srivrunyoo

A CONTEXTUALIZED WEB-BASED LEARNING ENVIRONMENT FOR DEVS MODELS

by

INTHIRA SRIVRUNYOO

Under the Direction of Xiaolin Hu

ABSTRACT

With the advance in applying technology in education, the traditional lecture-driven teaching style is gradually replaced by a more active teaching style where the students play a more active rule in the learning process. In this paper we introduce a new initiative to provide a suite of online tools for learning DEVS model.

The uniqueness of this tutorial project is the integration of information technology and multimedia into education through the development of an interactive tutorial and the characteristic of contextualized learning. The tutorial teaches students about the basic aspects of discrete event system and simulation. The interactive tutorial fully utilizes the power of the information and multimedia technology, web application and the programming language Java, to enhance students' learning to achieve rich interactivity. The tutorial greatly supports human-computer collaboration to enhance learning and to satisfy user goals by effectively allowing the user to interact.

INDEX WORDS: DEVS, Simulation, Web Application

A CONTEXTUALIZED WEB-BASED LEARNING ENVIRONMENT FOR DEVS MODELS

by

INTHIRA SRIVRUNYOO

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2007

A CONTEXTUALIZED WEB-BASED LEARNING ENVIRONMENT FOR DEVS MODELS

by

INTHIRA SRIVRUNYOO

Major Professor:	Xiaolin Hu
Committee:	Rajshekhar Sunderraman
	Ying Zhu

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

Dec 2007

ACKNOWLEDGEMENTS

I wish to record my sincere thanks to my advisor, Dr. Xiaolin Hu, for his guidance insightful suggestions and support. It is great honor for me to be able to work with Dr. Hu and I am thankful for having an advisor like him. I would also like to thank Dr. Raj Sunderraman, and Dr. Ying Zhu for serving as my advisory committee and taking precious time to review my thesis.

I wish to use this opportunity to express my thanks to my husband, Piyaphol Phoungphol, for his support for without it I would have never been able to achieve this today. I would also like to thank my father, Seng Srivrunyoo, and my mother Chamaiphon Srivrunyoo, for encouraging me to reach this far.

I would like to thank the Computer Science Department for this support during my gradate studies. I also thank all my friends and instructors at GSU for making these two years an enjoying experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HTML	HyperText Markup Language
DEVS	Discrete Event System Specification

1. INTRODUCTION

Today's advances in technology have provided new tools that allow us to be more creative. The purpose of this research is to design and develop a web-based tutorial that will facilitate the learning of the DEVS modeling and simulation framework and its underlying fundamental modeling concepts. The uniqueness of this tutorial project is not only the integration of information technology and multimedia into education through the development of an interactive tutorial, but also the characteristic of contextualized learning as we use a traffic light controller system simulation as an example. As we know, simulation has become one of the most powerful tools for the planning, design, and control of complex processes or systems. The DEVS formalism is a system-theory for simulation modeling and design which is theoretically a well-defined means of expressing hierarchical, modular models in discrete event simulation. A DEVS model works as a timed state machine so that state of the system is changed by external or internal events with elapsed time. The DEVS environment, based on the DEVS formalism offers a library with which users can easily build models in a hierarchical modular way. The coupled model in DEVS library is the major class to construct models hierarchically, while atomic models are the most basic classes. By using DEVS environment with the hierarchical, object-oriented feature, Models in the problem space can be developed deep in depth and expandable in width, as much as the user desires complex and detail models.

Unfortunately, it is known that DEVS has a steep learning curve for beginners, especially in the initial stages of the learning. Traditional learning methods of DEVS include lectures and tutorials. While these methods are effective in an enforced learning environment such as a lecture, their usage and influence are limited by the presence of the instructor and the limitation

of a physical environment. Web-based learning has the potential to influence a wide range of learners through the Internet. Current approaches of web-based learning of DEVS models mainly include online papers, online PPT slides, and online introduction of different type of DEVS applications (See e.g, ACIMS' website and DEVS Standardization Group website (http://www.sce.carleton.ca/faculty/wainer/standard/)). Without an instructor to guide through the process, these approaches are ineffective for new beginners. More recently, the WebStarter environment is made available through the Internet.

The DEVS-JAVA environment is an implementation of the DEVS formalism in Java, which enables the modeler to specify models directly in its terms. In the past few years, a visualization system known as the SimView was created to help students visualize a hierarchical model's structure and behavior and examine the model's source code. However, the structure of a model which expressed in a mathematical language called formalism is difficult to understand for a beginner. Moreover, a deep understanding of model is required in order to generate such a good simulator that can carry out the instructions of the model and a good model that can represent the real system.

Relating instructional content to the specific contexts of learners' lives and interests, known as contextualized learning, is believed to increase motivation to learn. This instructional strategy actively involves the learner in the learning process, combines content and context, and uses authentic materials. It is application oriented, learner centered, and time flexible. With this strategy, the purpose of the learning is explicit, learners share previous knowledge, transference of knowledge is explicit and immediately recognizable, and finally learners learn more effectively by integrating new knowledge/skills into already existing knowledge. As a result, a new form of teaching, contextualized web-based tutorial is developed to help students gain more

clearly understanding of the DEVS formalism. We exemplify the DEVS formalism atomic and coupled models using simple traffic light controller system. We implement the traffic simulation system based on the DEVS theory. Our model of intersection traffic is a simplified version of real-world intersection traffic. This will help students to gain more and more clearly understanding of the DEVS formalism by comparing to the real world system.

The outline of this thesis is as follows: In chapter 1, we give an introduction to this thesis. In chapter 2, we introduce a related works of DEVS model, web-based learning system, and contextualized learning system with some examples. In chapter 3, we briefly describe the DEVS modeling and simulation methodology. In chapter 4, we explain about the web-based learning system including an overview of DEVS and DEVS specification, atomic model, coupled model, simView, and advanced traffic system modeling. In chapter 5, we explain about system architecture and software design. In chapter 6, the conclusion and possible future works are listed. Chapter 7 provides the references.

2. RELATED WORK

This chapter briefly describes the background areas of research which served as the bulk of the foundation underlying the advances produced by this thesis. Additionally, it gives an example of other existing systems that are meant to perform a similar function to that of this thesis.

2.1 Related models of DEVS models

This section briefly describes other existing models or systems which are related to discrete event simulation model (DEVS).

2.1.1 Petri net

A Petri net [1] is a graphical and mathematical modeling tool. As can be seen in figure 2.1, a Petri net model consists of places, transitions, and arcs that connect them. Input arcs connect places with transitions, while output arcs start at a transition and end at a place. There are other types of arcs, e.g. inhibitor arcs. Places can contain tokens; the current state of the modeled system is given by the number and type if the tokens are distinguishable of tokens in each place. Transitions are active components. They model activities which can occur (the transition fires), thus changing the state of the system (the marking of the Petri net).

Transitions are only allowed to fire if they are enabled, which means that all the preconditions for the activity must be fulfilled (there are enough tokens available in the input places). When the transition fires, it removes tokens from its input places and adds some at all of its output places. The number of tokens removed / added depends on the cardinality of each arc. The interactive firing of transitions in subsequent markings is called token game.

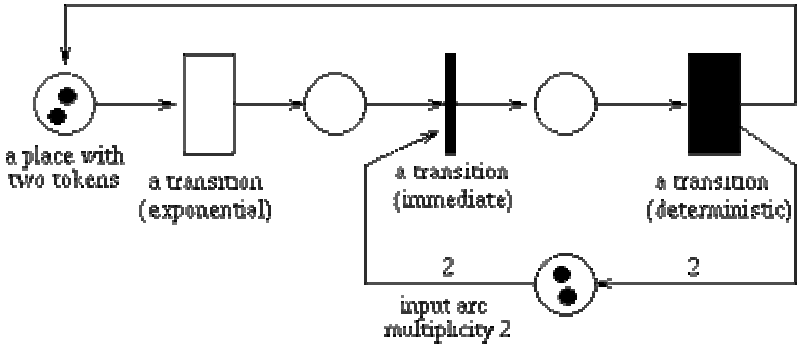

Figure 2.1 Example of a Petri Net

Petri nets are a promising tool for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. As a graphical tool, Petri nets can be used as a visual-communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems.

To study performance and dependability issues of systems it is necessary to include a timing concept into the model. There are several possibilities to do this for a Petri net; however, the most common way is to associate a firing delay with each transition. This delay specifies the time that the transition has to be enabled, before it can actually fire. If the delay is a random distribution function, the resulting net class is called stochastic Petri net. Different types of transitions can be distinguished depending on their associated delay, for instance immediate transitions (no delay), exponential transitions (delay is an exponential distribution), and deterministic transitions (delay is fixed).

The concept of Petri nets has its origin in Carl Adam Petri's dissertation Kommunikation mit Automaten, submitted in 1962 to the faculty of Mathematics and Physics at the Technische Universität Darmstadt, Germany.

2.1.2 Finite State Machine (FSM)

A finite state machine (FSM) [2] or finite state automaton (plural: automata) or simply a state machine is a model of behavior composed of a finite number of states, transitions between those states, and actions.

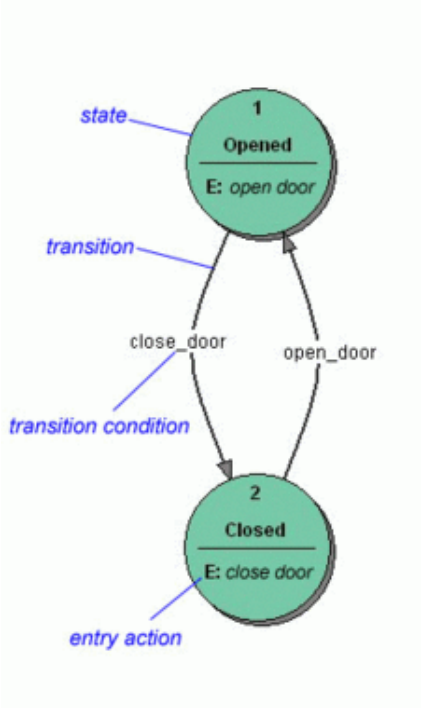

Figure 2.2 Finite State Machine

A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An action is a description of an activity that is to be performed at a given moment. There are several action types:

- Entry action which is performed when entering the state

- Exit action which is performed when exiting the state
- Input action which is performed depending on present state and input conditions
- Transition action which is performed when performing a certain transition

A FSM can be represented using a state diagram (or state transition diagram) as in figure 2.2. Besides this, several state transition table types are used. The most common representation is shown below: the combination of current state (B) and condition (Y) shows the next state (C). The complete actions information can be added only using footnotes. An FSM definition including the full actions information is possible using state tables as in table 2.1.

Table 2.1 State transition table

Current State -> Condition	State A	State B	State C
Condition X	…	…	…
Condition Y	…	State C	…
Condition Z	…	…	…

In addition to their use in modeling reactive systems presented here, finite state automata are significant in many different areas, including electrical engineering, linguistics, computer science, philosophy, biology, mathematics, and logic. A complete survey of their applications is outside the scope of this article. Finite state machines are a class of automata studied in automata theory and the theory of computation. In computer science, finite state machines are widely used in modeling of application behavior, design of hardware digital systems, software engineering, compilers, network protocols, and the study of computation and languages.

2.2 Web-based learning system

This section strategy employed by other web-based learning systems for depicting the structure and behavior of others models that are related to discrete event simulation models. Since the set of such systems that is designed to work specifically with DEVS/DEVSJAVA models is very limited, the discussion will be expanded to systems that support other implementations of others models that are related to discrete event simulation modeling.

2.2.1 Interactive Tutorials on Petri Nets

The interactive tutorials [3] introduce Petri nets, state spaces, and place/transition invariants. The tutorials are created by Wil van der Aalst, Vincent Almering, and Hermen Wijbenga from TU Eindhoven, the Netherlands. Figure 2.3 and 2.4 show the interactive tutorial of elevator model and two traffic lights model.

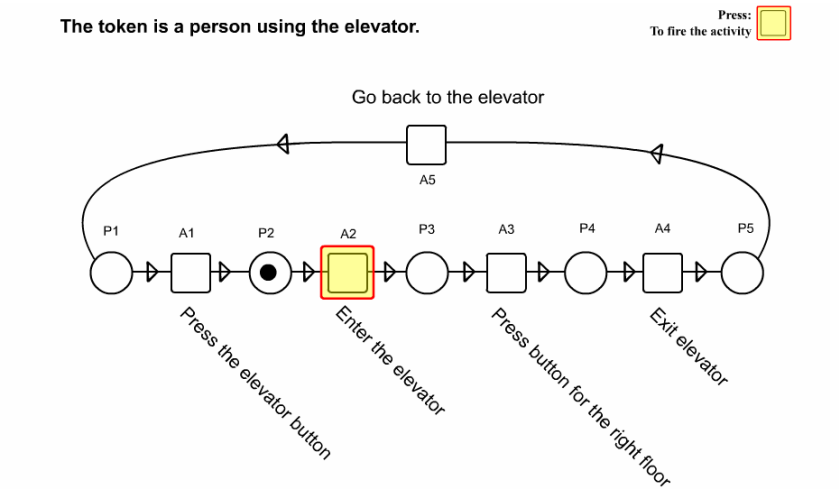

Figure2.3 An Interactive Tutorial of Elevator Model

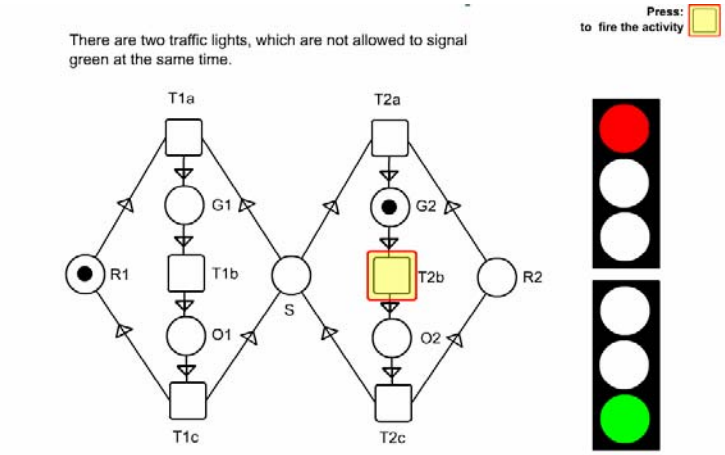

Figure2.4 An Interactive Tutorial of Two Traffic Lights Model

2.2.2 Petri net applet

As can be seen in figure 2.5, this Petri net applet tutorial [4] will allow users to create their own Petri net network model by dragging and dropping place and transition. After creating the desired places and transitions, arcs can be drawn between places and transitions. After completing the network, tokens can be added and then users can start the simulation by pushing the button from the applet. This tutorial will help users to learn Petri net model as the token is moving along the place and transition during the simulation.

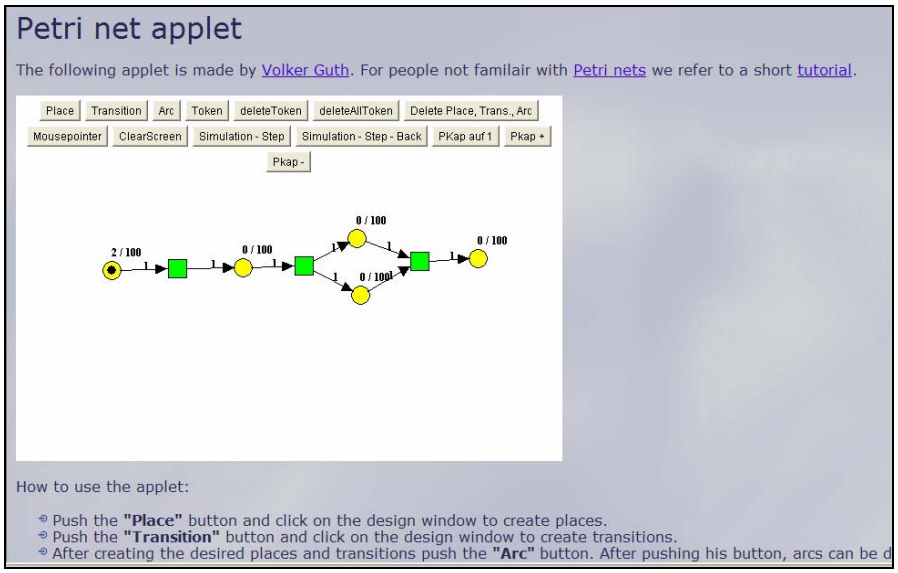

Figure 2.5 An Applet Tutorial for Petri Net Model

2.2.3 Finite State Machine Explorer

As the study of theoretical computational machines has always been based around static representations, usually on paper, and the complexity of machines being studied increases, it becomes increasingly difficult to represent their behavior statically, and therefore correspondingly harder to visualize mentally.

This finite state machine explorer [5] attempts to rectify this situation through the development of an interactive graphical system which supports the construction of the class of computational machines known as finite state automata. Animation is then used to dynamically illustrate their behavior on given inputs. As can be seen in figure 2.6, user can add states, create a connection between states, and simulate the machine.

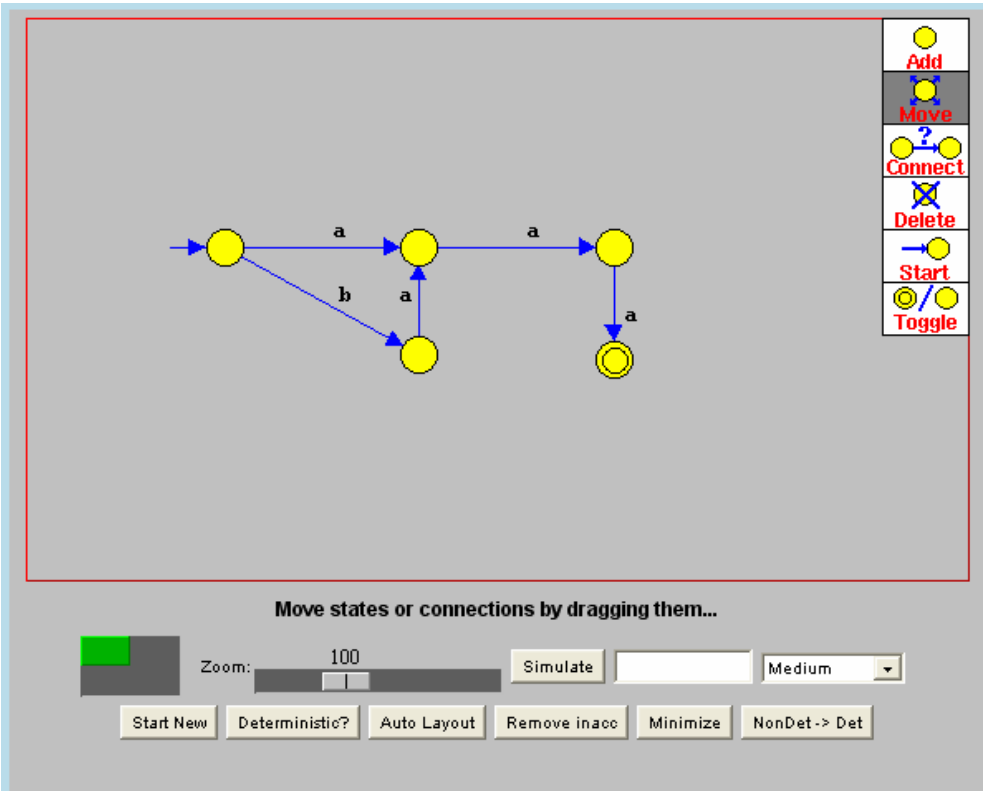

Figure 2.6 A Finite State Machine Explorer

Furthermore, powerful manipulation facilities are developed to support the study and exploration of automata, including the conversion between equivalent classes of machine, and the automatic generation of layouts. Finally, input and output capabilities are provided to enable the continued development and reuse of constructed machines. This tutorial was created by Matt Chapman and is available at http://www.belgarath.org/java/fsme.html

2.3 Contextualized learning

This section strategy is employed by other contextualized learning systems that help students to learn about basic and the fundamental of computer science.

2.3.1 The Alice software programming Tutorial

The goal of the Alice software programming tutorial [6] is to allow traditional programming concepts to be more easily taught and more readily understood.

Although computer programming has existed in its modern form for half a century, it still eludes all but a small fraction of society. While programming is an inherently difficult activity, there are currently many barriers, both mechanical and sociological, that prevent large portions of the population from learning to program a computer.

Alice addresses both the mechanical and sociological barriers that currently prevent many students from successfully learning to program a computer. Alice addresses the mechanical barriers to programming by making it much easier for students to create programs. Rather than having to correctly type commands according to obscure rules of syntax, students can drag-and-drop words in a direct manipulation interface. This user interface ensures that programs are always well-formed. In addition, Alice reifies object-based programming by providing animated, on-screen 3D virtual objects. Figure 2.7 and 2.8 show the graphic user interface of Alice software.

Sociological barriers are far more complex. Alice addresses the specific needs of the subpopulation of middle school girls. By supporting storytelling, an intrinsically motivating activity for middle school girls, Alice will make programming a means to an exciting end.

This approach allows students to author on-screen movies and games, where the concept of an "object" is made tangible via on-screen objects that populate a three-dimensional micro world. Students create programs by dragging and dropping program elements (if/then statements, loops, variables, etc.) in a mouse-based editor that prohibits syntax errors. The Alice system is a powerful, modern programming environment that supports methods, functions, variables, parameters, recursion, arrays, and events. They use this strong visual environment to support either an objects-first or an objects-early approach with an early introduction to events. In Alice, every object is an object that students can visibly see. They begin with objects in the very first chapter.

Figure 2.7 Program Code Editors of Alice's Software Tutorial

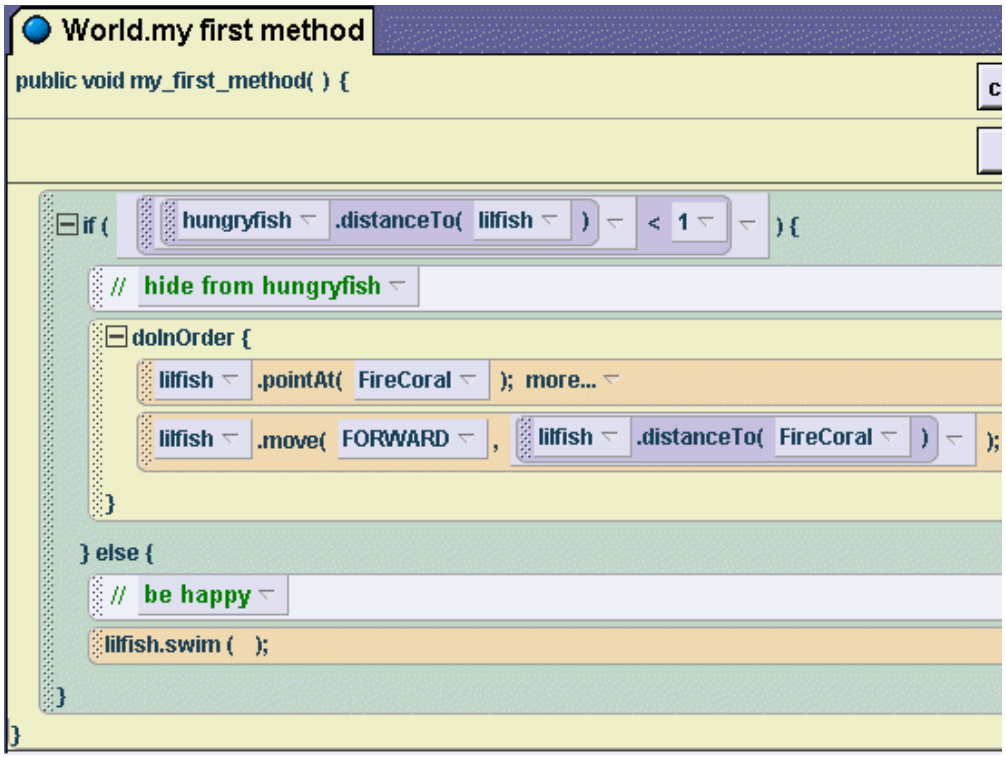

Figure2.8 Programs in Alice can optionally be displayed with a Java-like syntax

2.4 Simulation and Gaming for learning

This section strategy is employed by other simulation or gaming that help students to learn about basic and the fundamental of computer science.

2.4.1 SimSE

SimSE [7] is a computer-based simulation environment for teaching the software engineering process. The goal is to bridge the gap between the large amount of conceptual software engineering knowledge given to students in lectures and the comparably small amount of this they actually get to put into practice in an associated "toy" software engineering project.

As can be seen from figure 2.9, SimSE allows students to practice a “virtual” software engineering process or sub-process in a fully graphical, interactive, and fun setting in which

direct, graphical feedback enables them to learn the complex cause and effect relationships underlying the processes of software engineering.

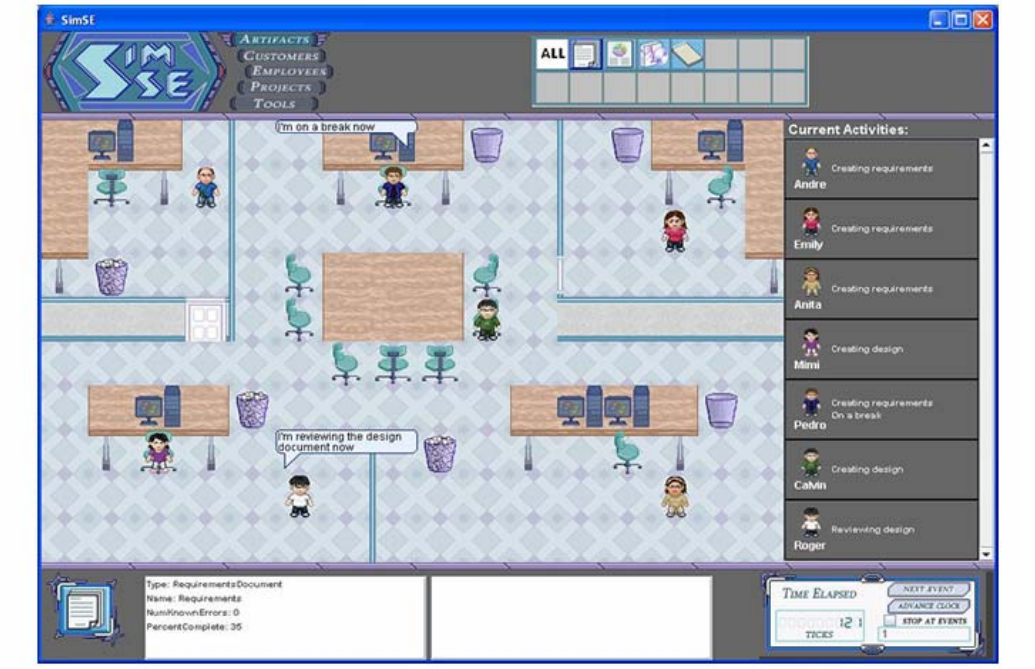

Figure 2.9 A Glimpse of A SimSE Game

The user interface of SimSE is fully graphical, displaying a "virtual" office in which the software engineering process is taking place, including typical office surroundings; employees, customers, and project information, as well as representations of software engineering artifacts that include such information as that artifact's completeness, correctness, and other similar qualities. Information about the status of individuals is provided through automatic pop-up "bubbles" over the heads of individuals and through explicitly querying an individual. Players use information gleaned from these sources to make decisions and take actions, driving the software engineering process to complete a project within budget, schedule, and at or above the customer's desired quality requirement.

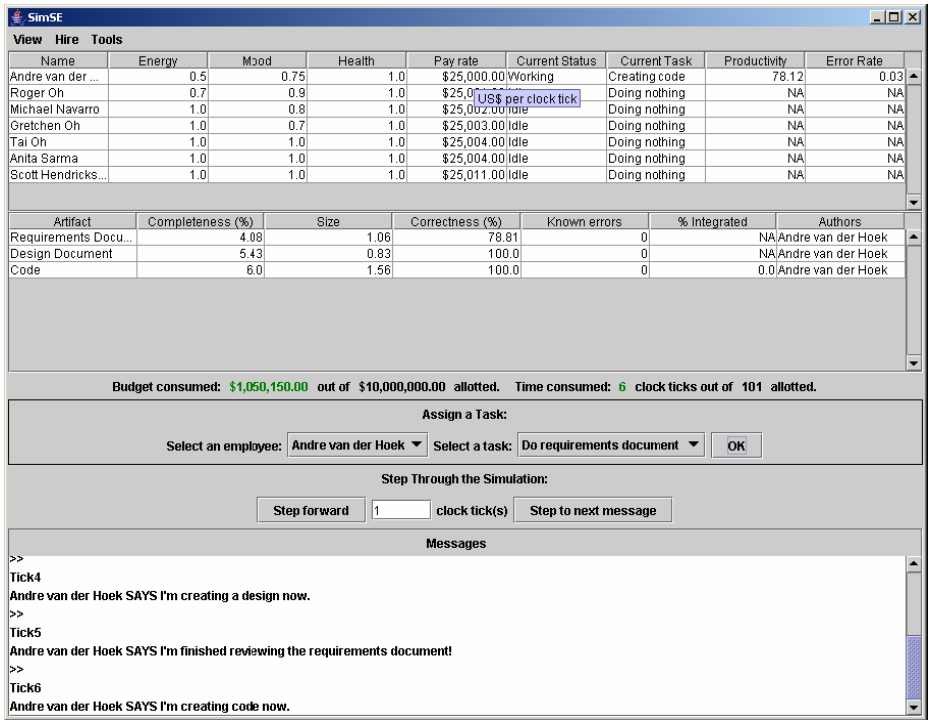

Figure 2.10 Object Builder User Interface.

This model builder completely hides the un-derlying modeling language from the modeler, and pro-vides a graphical user interface for specifying the object types, start state, actions, rules, and graphics for a model. Figure 2.10 shows the user interface for the object builder, the part of the model builder that supports defining object types. For the sake of space, the interfaces for the other parts of the model builder are not shown, but they are similar in appearance to the object builder in that they all facilitate building a model using buttons, drop-down lists, menus, and dialog boxes—no programming is required. Once a model is specified, the model builder then generates Java code for a complete, executable, customized simulation game based on the given model.

3. DEVS BACKGROUND

This chapter briefly describes the DEVS modeling and simulation methodology. The Discrete Event System Specification [8] (DEVS) formalism provides a means of specifying a mathematical object called a system [Zeigler, et. al, 2000].

3.1 Discrete Event System Modeling and Simulation

Basically, a system has a time base, inputs, states, and outputs, and functions for determining next states and outputs given current states and inputs. Discrete event systems represent certain constellations of such parameters just as continuous systems do. For example, the inputs in discrete event systems occur at arbitrarily spaced moments, while those in continuous systems are piecewise continuous functions of time. The insight provided by the DEVS formalism is in the simple way that it characterizes how discrete event simulation languages specify discrete event system parameters.

The conceptual framework underlying the *DEVS formalism* is shown in Figure 3.1. The modeling and simulation enterprise concerns three basic objects:

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

- ***Experimental frame***, which captures how the modeler's objectives impact on model construction, experimentation and validation. As we shall see later, in DEVJAVA experimental frames are formulated as model objects in the same manner as the models of primary interest. In this way, model/experimental frame pairs form coupled model objects with the same properties as other objects of this kind. It will become evident later, that this uniform treatment yields key benefits in terms of modularity and system entity structure representation.

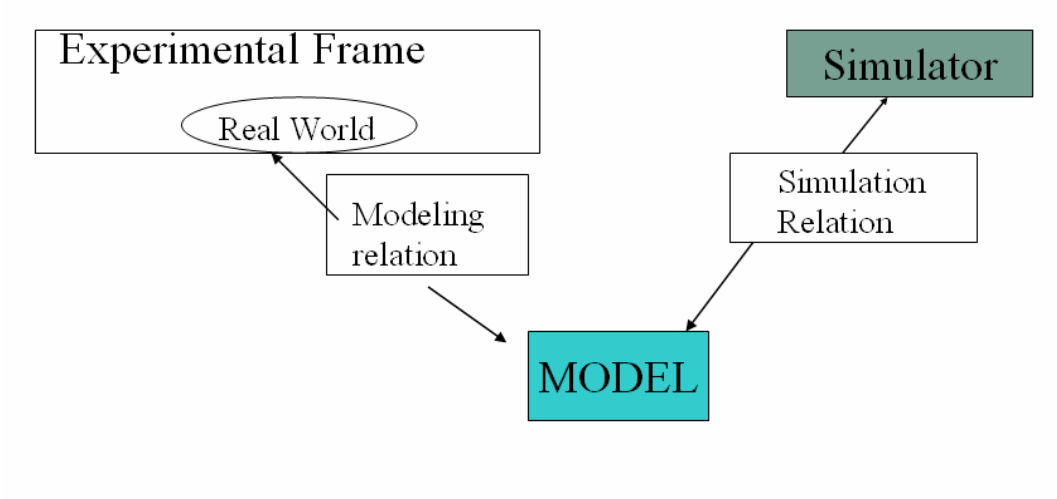

Figure 3.1 Basic Entities and Relations

The basic objects are related by two relations:

- ***Modeling relation*** linking real system and model, defines how well the model represents the system or entity being modeled. In general terms a model can be considered valid if the data generated by the model agrees with the data produced by the real system in an experimental frame of interest.

- ***Simulation relation****,* linking model and simulator, represents how faithfully the simulator is able to carry out the instructions of the model.

The basic items of data produced by a system or model are *time segment*s. These time segments are mappings from intervals defined over a specified time base to values in the ranges of one or more variables. The variables can either be observed or measured. An example of a data segment is shown in Figure 3.2.

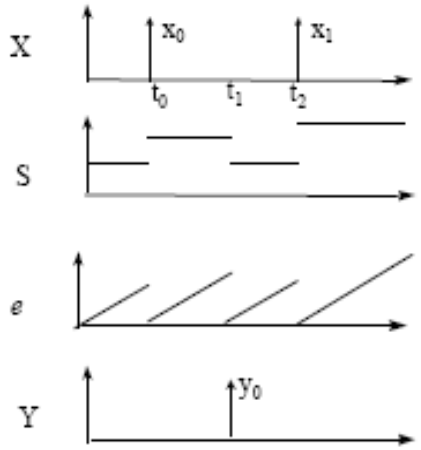

Figure 3.2 Discrete Event Time Segments

The structure of a model may be expressed in a mathematical language called formalism. The discrete event formalism focuses on the changes of variable values and generates time segments that are piecewise constant. Thus an event is a change in a variable value, which occurs instantaneously.

In essence the formalism defines how to generate new values for variables and the times the new values should be take effect. An important aspect of the formalism is that the time intervals between event occurrences are variable (in contrast to discrete time where the time step is a fixed number).

3.2 DEVS Formalism

The DEVS (Discrete Event System Specification) [9] formalism is derived from generic dynamic systems theory and has been applied to both continuous and discrete phenomena. It provides a formal modeling and simulation (M&S) framework with well-defined concepts of coupling of components, and hierarchical modular model construction. DEVS has been operationalized to serve as a practical simulation and execution tool in a variety of implementations [8, 10, 11].

There are two kinds of DEVS models: atomic model and coupled model. Atomic models are the basic components. Coupled models have multiple sub-components and can be constructed in a hierarchical way.

3.2.1 Atomic Model

In the DEVS formalism, one must specify:

1) Basic models from which larger ones are built, and

2) How these models are connected together in hierarchical fashion.

To specify modular discrete event models requires that we adopt a different view than that fostered by traditional simulation languages. As with modular specification in general, we must view a model as possessing input and output ports through which all interaction with the environment is mediated. In the discrete event case, events determine values appearing on such ports. More specifically, when external events, arising outside the model, are received on its input ports, the model description must determine how it responds to them. Also, internal events arising within the model change its state, as well as manifesting themselves as events on the output ports to be transmitted to other model components.

An atomic model of a standard DEVS is a structure:

$$M = (X,S,Y,\delta_{int},\delta_{ext},\lambda,ta)$$

where,

X is the set of inputs

S is a set of states

Y is the set of outputs

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

$\delta_{ext}: Q \times X \rightarrow S$ is the external transition function, where

$\lambda : S \rightarrow Y$ is the output function

$ta : S \rightarrow R^{+}_{0,\infty}$ is the time advance function

$Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total state set

e is the time elapsed since last transition

An atomic model contains the following information:

- the set of input ports through which external events are received
- the set of output ports through which external events are sent
- the set of state variables and parameters: two state variables are usually present, “phase” and “sigma” (in the absence of external events the system stays in the current “phase” for the time given by “sigma”)
- the time advance function which controls the timing of internal transitions – when the “sigma” state variable is present, this function just returns the value of “sigma”.
- the internal transition function which specifies to which next state the system will transit after the time given by the time advance function has elapsed
- the external transition function which specifies how the system changes state when an input is received – the effect is to place the system in a new “phase” and “sigma” thus

scheduling it for a next internal transition; the next state is computed on the basis of the present state, the input port and value of the external event, and the time that has elapsed in the current state.

- the confluent transition function which is applied when an input is received at the same time that an internal transition is to occur –the default definition simply applies the internal transition function before applying the external transition function to the resulting state.
- the output function which generates an external output just before an internal transition takes place.

The interpretation (semantic) of these elements is briefly described below. The input event set defines all possible inputs that may occur on the input ports; the output set consists of all possible outputs the atomic model may send out. External inputs received on the input ports invoke a model's external transition function, which determines how the model changes its state based on the inputs and model's current state. The model remains in a state for an amount of time that is determined by the time advance function. When this time has expired the output function is invoked, which sends output on the output ports. Following the invocation of the output function, the internal transition function is invoked to transit the model to a new state. The confluent transition function is invoked when extern events and internal events happen at the same time.

3.2.2 Coupled Models

Atomic models may be coupled in the DEVS formalism to form a *coupled model*. A coupled model tells how to couple (connect) several component models together to form a new model. This latter model can itself be employed as a component in a larger coupled model, thus

giving rise to hierarchical construction. Two major activities involved in coupled models are specifying its component models and defining the couplings which create the desired communication networks. A coupled model is defined as follows:

$$DN = <X, Y, D, \{Mi\}, \{Ii\}, \{Zi,j\}>$$

where,

X : set of external input events;

Y : a set of outputs;

D : a set of components names;

for each i in D,

Mi is a component model

Ii is the set of influencees for i

for each j in Ii,

Zi,j is the i-to-j output translation function

A coupled model contains the following information:

- the set of components
- the set of input ports through which external events are received
- the set of output ports through which external events are sent

These components can be synthesized together to create hierarchical models having external input and output ports.

The coupling specification consisting of:

- the external input coupling which connects the input ports of the coupled to model to one or more of the input ports of the components — this directs inputs received by the coupled model to designated component models

- the external output coupling which connects output ports of components to output ports of the coupled model- thus when an output is generated by a component it may be sent to a designated output port of the coupled model and thus be transmitted externally
- the internal coupling which connects output ports of components to input ports of other components- when an input is generated by a component it may be sent to the input ports of designated components (in addition to being sent to an output port of the coupled model)

4. THE WEB-BASED LEARNING SYSTEM

4.1 Traffic Light Controller System.

For the purposes of this project we will use a modeling problem that most everyone will understand. We implement the traffic simulation system based on the DEVS theory. Our model of intersection traffic is a somewhat simplified version of real-world intersection traffic. We do not allow cars to turn, all cars begin traveling roughly the same speed, and all traffic lights only have three signal; red, green, and flashing red. These simplifications make the implementation a lot easier but do not detract from the fundamental challenges of our goal. Since we are trying to create a web-based tutorial which will allow the users to learn the DEVS modeling and simulation framework, it is not unreasonable to assume that each vehicle is always trying to travel at the speed limit. Additionally, almost every vehicle in production is capable of attaining the speed limit of any road, so this assumption is not a far stretch. Another reason why we choose a traffic light controller system as our modeling example is that the ability to model transport scenarios through simulation has already proven an effective tool in traffic management and infrastructure planning. Many researches have been done through out this area for the past few years as traffic congestion have become a very serious problems. The intelligence traffic light controllers based on various algorithms have been proposed among those researches.

Although using an implementation of DEVS to develop a prototype of the traffic model does not cover all kinds of roads and crossing and traffic conditions, we can implement modules which can be used to simulation some simple traffic systems. Based on the prototype developed

we can conduct a case study to study and analyze the traffic light controller system or even an intelligence traffic light controller system.

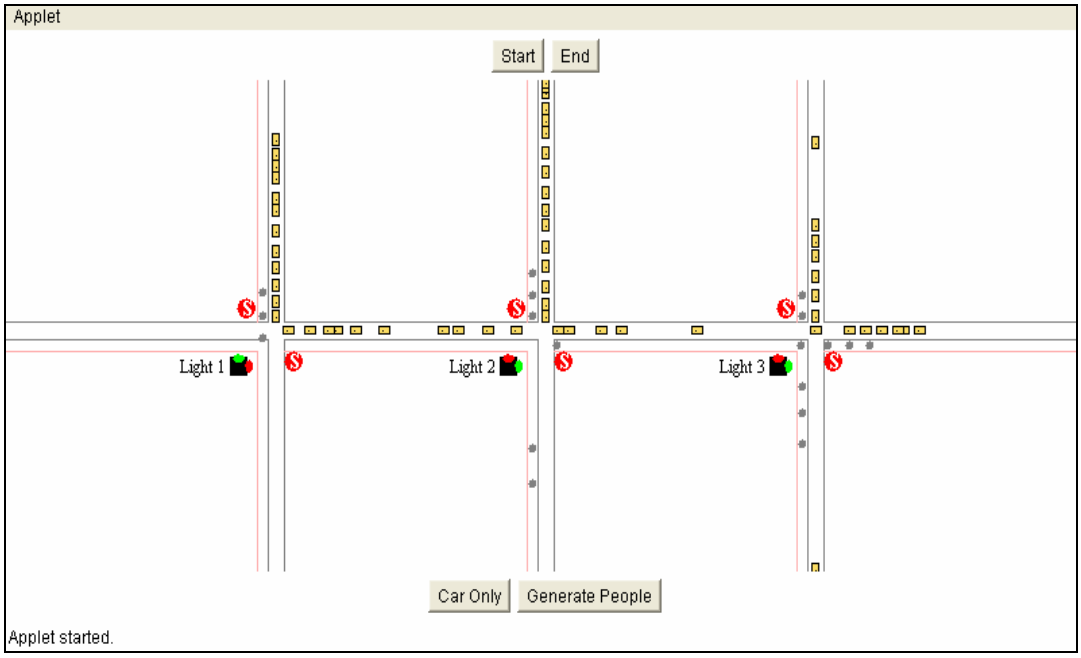

Figure 4.1 Traffic Flow Simulation

Figure 4.1 shows the simulation of traffic flow using Java programming language. It takes advantage of multithread, a unique feature function in Java and DEVS Java, a powerful modeling & simulation language. All of the traffic light controllers are controlled by DEVS. Two threads operate concurrently with DEVS to represent the cars and people walking across the street.

4.2 DEVS Formalism of Traffic Light Controller System

In this section we discuss our traffic light controller system as an example. We then explain how to coupling atomic models together to create a coupled model for our traffic light controller system. Finally, we discuss an advanced traffic system modeling.

4.2.1 Example – A Traffic Light Controller

A DEVS model of a traffic light controller has two input ports, one for button and the other for signal, and one output port.

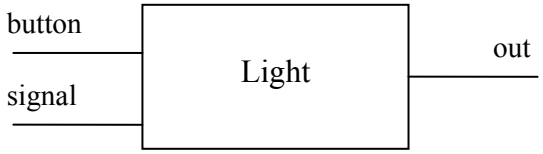

Figure 4.2 Traffic light Controller with input and output ports

A traffic light controller is modeled as a DEVS, as shown in Figure 4.2. The input port "button" is for receiving an input from people who want to walk across an intersection while the traffic light is in green. The input port "signal" is for receiving a message or an output from the other traffic lights. The traffic light has three phases; red, green, and flashing red. The traffic will be in red for some period of time and then change to green. After staying in green for some period of time, it will change back to red and so on. When the traffic light is in green and receive a message from input port "button" or "signal", it will change to red for some period of time and then send out a message through output port "out" to the other traffic light. A DEVS specification of a basic traffic light controller can be described as follows:

$$DEVS_{traffic\ light} = (X,S,Y,\delta_{int},\delta_{ext},\lambda,ta)$$

where

X = {"button", "signal"}

Y = {"signal to the other traffic lights}

S = {"red", "green", "flashing red"}

δ_{int} (“red”,σ) = (“green”, 7)

δ_{int} (“green”,σ) = (“red”, 7)

δ_{ext} (“green”,σ,e,x) = (“red”, 5)

δ_{ext} (“red”,σ,e,x) = (“red”, 5)

λ (“red”, σ) = “signal to the other traffic lights”

ta (phase, σ) = σ

4.2.2 DEVSJAVA Implementation of the Traffic Light Controller.

The implementation of the traffic light controller in DEVSJAVA is presented as follows:

```
public class trafLight extends ViewableAtomic{

public trafLight(String name, int time, String initS){
     super(name);
     addInport("button");
     addInport("signal");
     addOutport("out");
}

public void initialize(){
     holdIn(green, 7);
}

public void  deltext(double e,message x)
{
     Continue(e);

     if (somethingOnPort(x, "button")) {

          if (phaseIs(“green”)) {
               holdIn(“red”,5);
          }else if(phaseIs(“red”)){
               holdIn(“red”,5);
          }

     } else if (somethingOnPort(x, “signal”)){

          if (phaseIs(“green”)) {
               holdIn(“red”,5);
          }else if(phaseIs(“red”)){
               holdIn(“red”,5);
          }
```

```
      }
}

public void  deltint( )
{
      if(phaseIs("red")){
            holdIn("green",7);
      }

      else if(phaseIs("green")){
            holdIn("red",7);
      }

}

public message out( )
{
      message  m = new message();
      if(phaseIs("red")){
            m = new message();
            content con = makeContent("out", new entity("signal"));
            m.add(con);
      }
       return m;
}
}
```

4.2.3 Simulation Viewer

The Simulation Viewer [12] , known as SimView was crafted as an extension to the DEVSJAVA simulation framework, which in turn is an implementation of the DEVS formalism. It helps users to visualize a hierarchical model's structure and behavior during the running of the simulation. Figure 4.3 shows the DEVSJAVA Simulation Viewer of an atomic traffic light model with an initial state "green" and sigma equal to 4. It has two input ports; "button" and "signal" and one output port "out".

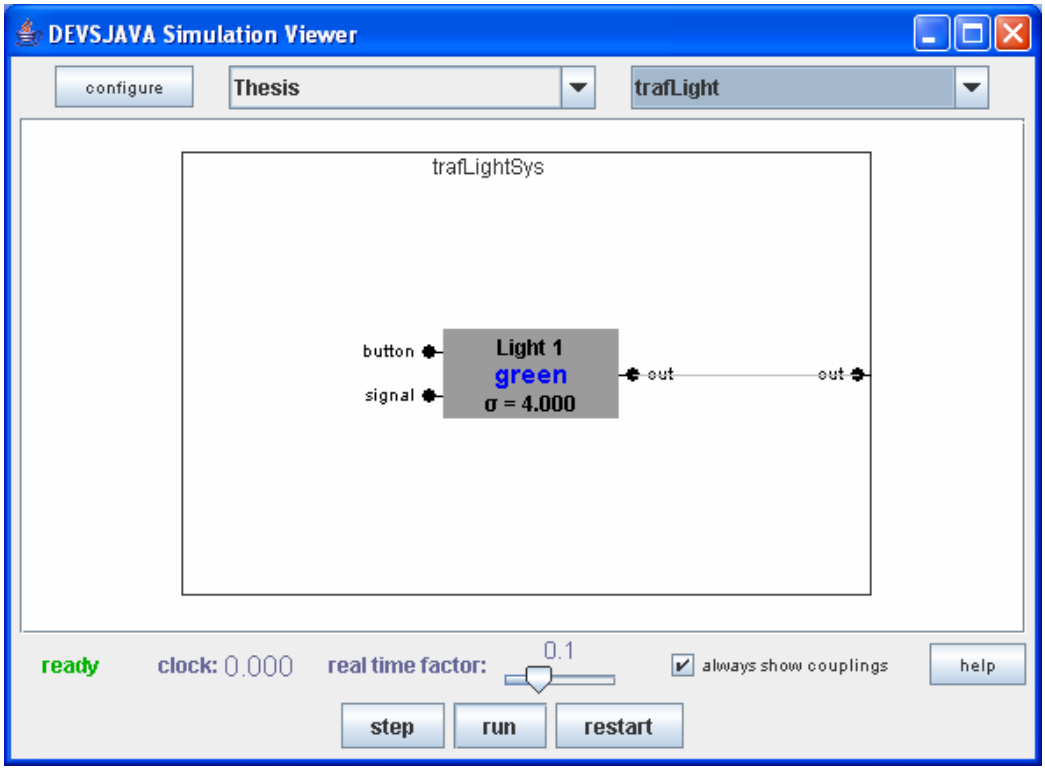

Figure 4.3 SimView of a Traffic Light Controller Model

Figure 4.4 shows the DEVSJAVA Simulation Viewer of a coupled model representing our traffic light controller system. Our traffic light controller system has three traffic lights coupling together. The output port "out" of the first traffic light is coupling to an input port "signal" of the second traffic light. Similarly, an output port "out" of the second traffic light is coupling to an input port "signal" of the third traffic light. All traffic lights initialized with green phase and have different time elapse as can be defined by users. The first traffic light will be in green for 4 sec. Then an internal transition function will be called and the first traffic light will change to red. After staying in red for 7 sec., an output function will be called and the message will be send out through output port "out" into input port "signal" of the second traffic light.

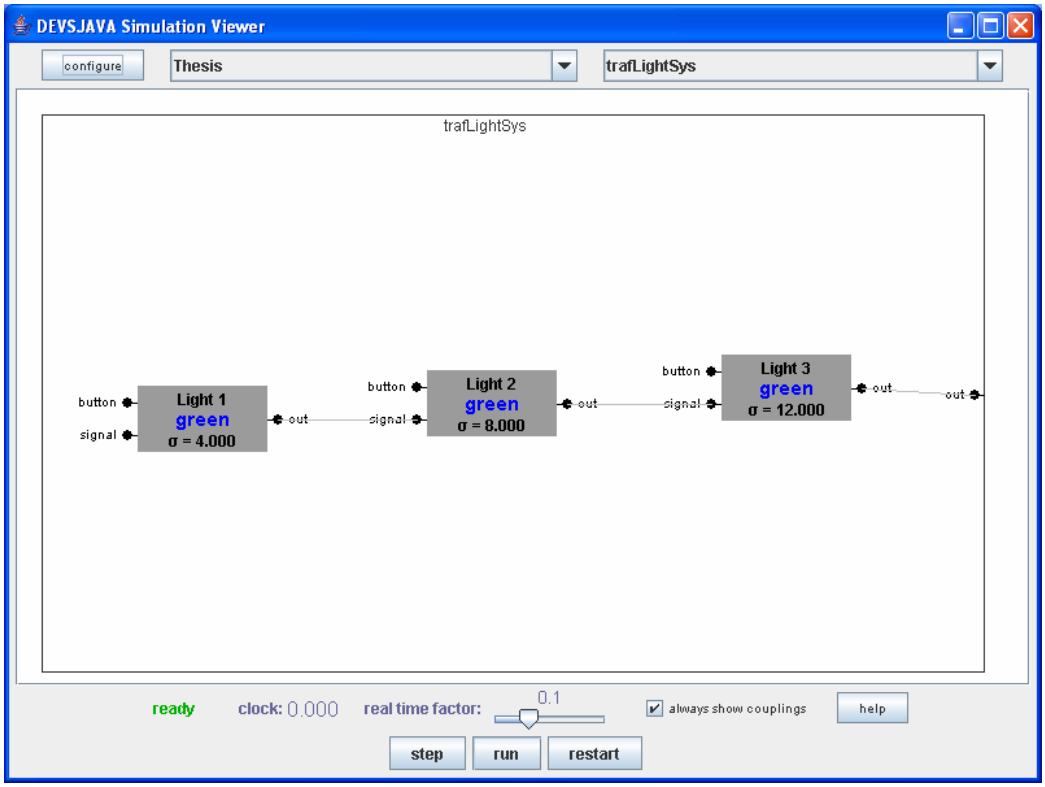

Figure 4.4 SimView of a Traffic Light Controller System

4.2.4 Advanced Traffic System Modeling

A traffic light controller can be modeled using DEVS continuous system [13] which results in a model that is closer to a real world traffic light controller system. In discrete time modeling, there is a state transition function which computes the state at the next time instant given the current state and input. In DEVS continuous system using an approach of differential equations, the state transition relation is quite different. For differential equation models, we do not specify a next state directly but instead, use a derivative function to specify the rate of change of the state variables. At any particular time instant on the time axis, given a state and an input value, we only know the rate of change of the state. The state at any point in the future must compute from this information.

Consider modeling traffic. The basic element is a one way roadway such as a highway or street lane between intersections. We divide the roadway into segments as illustrated in Figure 4.5. The speed of a vehicle in such as segment is to be determined by the number of vehicles in its segment as well as the number in the segment ahead. Roughly, the more congested the road ahead the slower a driver must go.

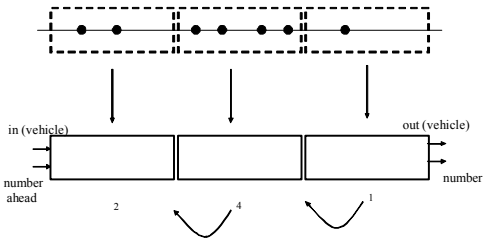

Figure 4.5 Partitioning a Roadway

We can represent the roadway as a one-dimensional cellular space where the cells represent segments. Each cell keeps track of the positions and speeds of the vehicles within it. When a vehicle reaches the end of a segment it is added to the following segment. Thus, one output port of a cell signals the occurrence of a vehicle at the far boundary of the associated segment. The other output port will transmit the number of the vehicles in the segment to the rear segment.

The time line of a vehicle can be depicted as in Figure 4.6. Because the number of vehicles in a segment changes discretely, the speed (v) of a vehicle changes in a step-like (piecewise constant) fashion. The distance along the segment (s) therefore increases in line segments as illustrated. When the vehicle reaches the end of the segment it is transferred to the next segment. This is represented by the output (y) which is zero until the segment boundary is

crossed, at which point it becomes 1. The number of vehicles in the segment is also computed and transmitted to the predecessor cell at this time as well.

The Euler approach can be used to update the vehicle positions along the way.

$$s(t+h) = s(t) + v(t)*h$$

where `v(t)` is speed of the vehicle at time t as computed from the number of vehicles in this segment and the one ahead. Since there may be a number of vehicles within a segment at any one time, we need a container to hold them. To make this possible, we defined a derived class of entity:

```
class vehicle extends entity{
    public double position, speed;
    public vehicle(double Position, double Speed){
         position = Position;
         speed = Speed;
   }
}
```

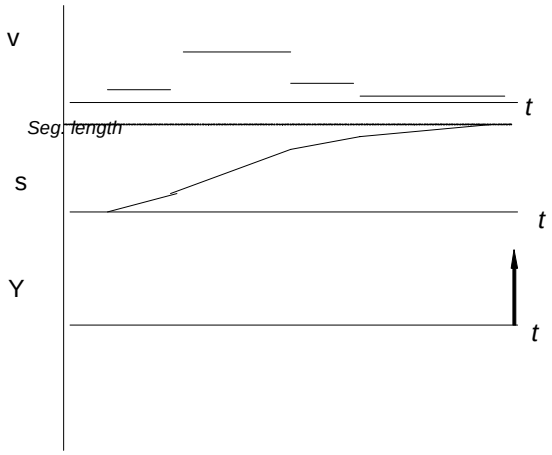

Figure 4.6 Vehicle Time Segments

Following the approach to DEVS implementation of cellular automata, we define a subclass of cell to update the vehicles in the segment it represents:

```
public class vehicleStep extends cell{
    protected set vehicles,crossed;
    protected double time_step,segLength;
    protected numberAhead;

public vehicleStep (){
super("vehicleStep ");
phases.add("active");
inports.add(“numberAhead”);
outports.add(“number”);
initialize();
}
public void initialize(){
     passivate();
     vehicles = new set();
     crossed = new set();
     segLength = 100;
     time_step = .1;
     numberAhead = 0;
     super.initialize();
}
public void  deltext(double e,message   x)
{
    Continue(e);
 for (int i=0; i< x.get_length();i++){
  if (message_on_port(x,"numberAhead",i)) {
      entity ent = x.get_val_on_port("numberAhead",i);
      intEnt d = (intEnt)ent;
      numberAhead = d.getv();
  }
 }
 for (int i=0; i< x.get_length();i++){
  if (message_on_port(x,"in",i)) //comes with posn & speed
      vehicles.add(x.get_val_on_port("in",i));
      //add all vehicles before updating
}
  update(); // new + old vehicles for next time step
  nextCrossed();//determine next crossings for output
  hold_in("active”,time_step);
}
public void update(){

for (entity   p = vehicles.get_head();p != null;p= p.get_right()){
       entity   ent = p.get_ent();
       vehicle v = (vehicle)ent;
       double newSpeed =
computeSpeed(vehicles.get_length(),numberAhead);
```

```
       v.position = v.position + newSpeed*time_step;
       v.speed = wSpeed;
     }
}
public void nextCrossed(){
   if (speedAhead <= 0.01)return;
     for (entity   p = vehicles.get_head();p != null;p=
p.get_right()){
       entity   ent = p.get_ent();
       vehicle v = (vehicle)ent;
       if (v.position >= segLength){
            v.position = v.position - segLength;
            vehicles.remove(ent);
            crossed.add(ent);
     }
}
}
public void  deltcon(double e,message   x){

//same as deltext()
 …

 crossed = new set();//empty after outputting

…
}

public void  deltint( )
{
if (!vehicles.empty()){
  update();
  crossed = new set();//empty after outputting
  nextCrossed();
  hold_in("active", time_step);
  }
else passivate();
}
public message  out( )
{
  message   m = new message();
if (phase_is("active")){
     m.add(make_content_address("number",
        new intEnt(vehicles.get_length()),
           new addrclass(my_location.i-1,
                         my_location.j, my_location.k)));
 for (entity  p = crossed.get_head(); p!=null;p=p.get_right())
         m.add(make_content_address("out",p.get_ent(),
                        new addrclass(my_location.i+1,
                         my_location.j, my_location.k)));
}
 return m;
}
```

```
}
```

Note that we predict what the state and output will be at the next time step. When that the time of the step elapses we the output is emitted and the next state and output are recomputed.

Compute Speed is function that embodies our hypotheses about how traffic "cloud" around a driver influences his/her speed. There is obviously an approximation here in that most of experience driving decisions in terms of the more detailed behavior of the cars immediately surrounding us on the highway.

We embed a row of such cells into a block model with a traffic generator:

```
public class roadSeg extends block {

public roadSeg(String nm)
{
   super(nm);

cell g  = new trafficGenr("g",10,new addrclass (0,0,0));
add(g);

for (int i = 1; i <= 4;i++)
 add(new vehicleStep("v"+i,new addrclass(i,0,0)));

  add_coupling("out","in");
  add_coupling("myNumber","numberAhead");
  initialize();
}
}
```

Once we make the assumption, as in the traffic model above, that the input and output of a component are piecewise constant, a much more efficient implementation of the internal dynamics becomes possible. Instead of the piecewise constant speed input we have discrete event inputs that provide the new values when the speed changes. Indeed, any piecewise constant time segment can be represented in this manner.

When a vehicle first enters the segment, its position is zero and the time it will take to reach the far end is easily predicted:

```
Tcross  = segLength/v
```

where v is the speed at the time of entry. Now we can schedule an internal event to occur at that time – this would enable a cell to produce the same output at the same time as computed in the discrete time approach – provided no additional inputs arrive. What do we do when such an external event occurs? In the external transition function we update the position to where it would be had we been updating it at every discrete time step. Then from this new current position, we recomputed the time to reach the far end. In other words, at time ti+1 when the i+1 input arrive, we have:

$$s_{i+1} = s_i + v_i * e$$

$$T_{cross,i+1} = segLength - s_{i+1} / v_{i+1}$$

Notice that we use the old speed, vi to update the position to its current value, but we use the new speed, vi+1 and new position to predict the next crossing time.

4.3 Web-based Tutorial Learning Process

The website can be divided into two main parts. The first part is a basic traffic light controller system which allows users to play with DEVS model. The second part is an advanced traffic light system modeling which was described in the previous section.

The first part is basically the main part of our tutorial. This tutorial is divided into a tutorial for an atomic model and a tutorial for a coupled model. As can be seen from figure 4.7, the main menu is located on the left hand side of the web page. Users are able to select the type of model that they want to play with. On the right hand side of the web page, we gives user an overview of a traffic light controller system, DEVS model, external transition function, internal transition function, and examples of DEVSJAVA code.

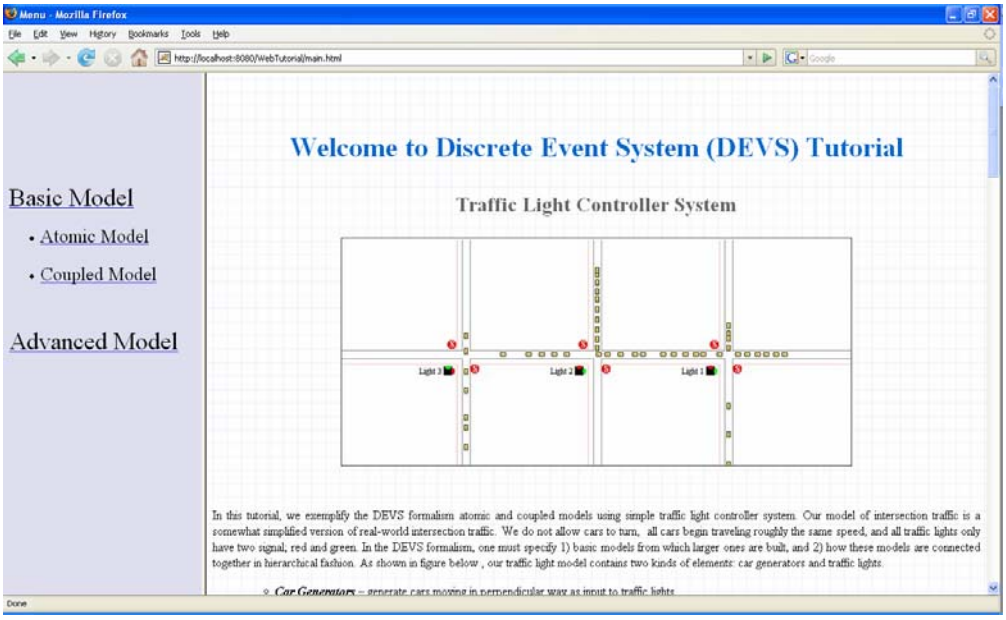

Figure 4.7 A tutorial's main page

4.3.1 Atomic Model

Figure 4.8 shows a tutorial web page for atomic model. There are two input ports; button and signal. User is able to adjust the car's speed as well as frequency for both directions. By select the drop down list and input parameters to text boxes, user can define:

- initialization function
- internal transition function
- external transition function
- output function
- confluent function

At the beginning, the simulation is running by the default model. Once user clicks on the submit button, all parameters defined by the users will be submitted to the server. A new

DEVS object will be created at the backend as well as the simulation will be refresh and running representing the new DEVS model.

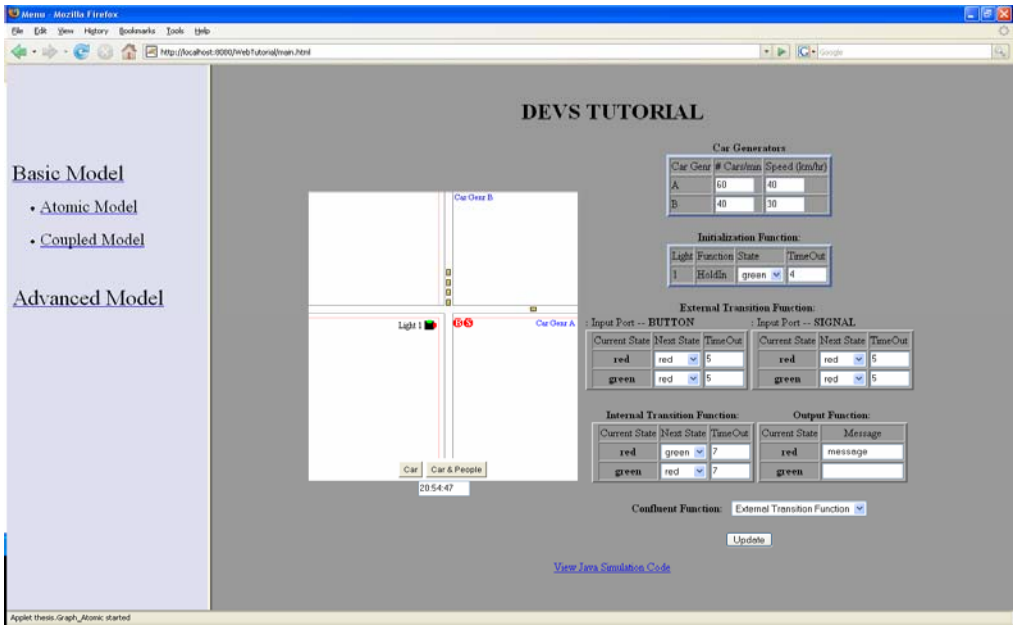

Figure 4.8 A Tutorial web page for Atomic Model

At the bottom of the web page, there is a link “View Java Simulation Code” which will allow the users to see all of the DEVSJAVA source code that is running behind the scene as can be seen in figure 4.9.

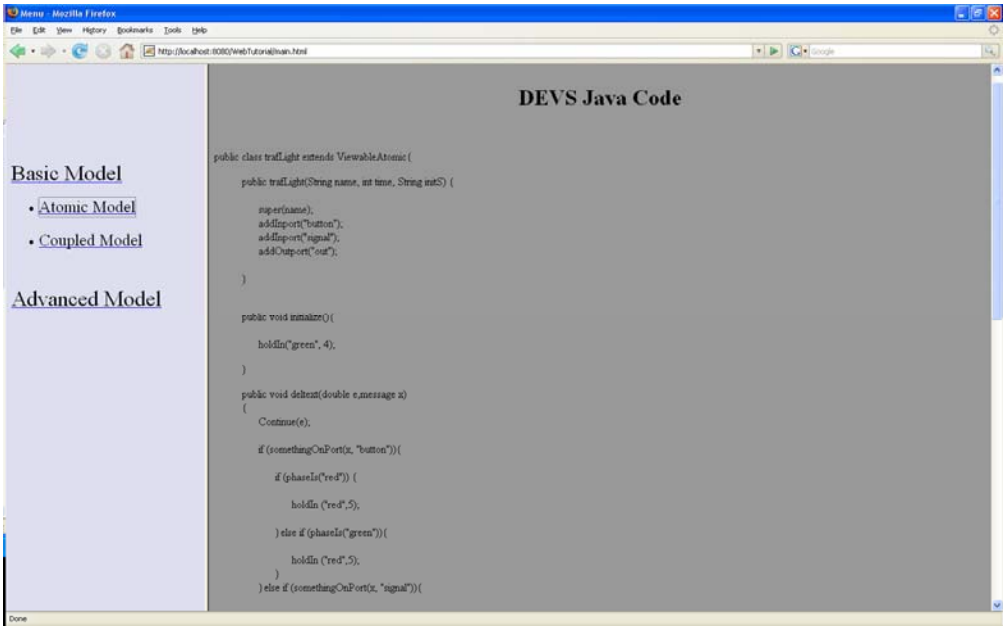

Figure 4.9 A tutorial web page showing DEVSJAVA code.

4.3.2 Coupled Model

A tutorial for coupled model will be different from atomic model. In order to help user to gain an idea how to coupling atomic models together to create a system, our tutorial starts with no traffic lights at all.. As can be seen from figure?, all cars are moving across each of the intersections without stopping. Users then will be able to add traffic light to the system and coupling each of the traffic lights together. For each traffic light, users are able to change car's speed and frequency along with, like atomic model, defining all of the functions that need to create each of an atomic model representing one traffic light.

The following step explains how to play with our tutorial with an example model:

Step 1. Figure 4.10 shows that the tutorial webpage starts with no traffic light at all.

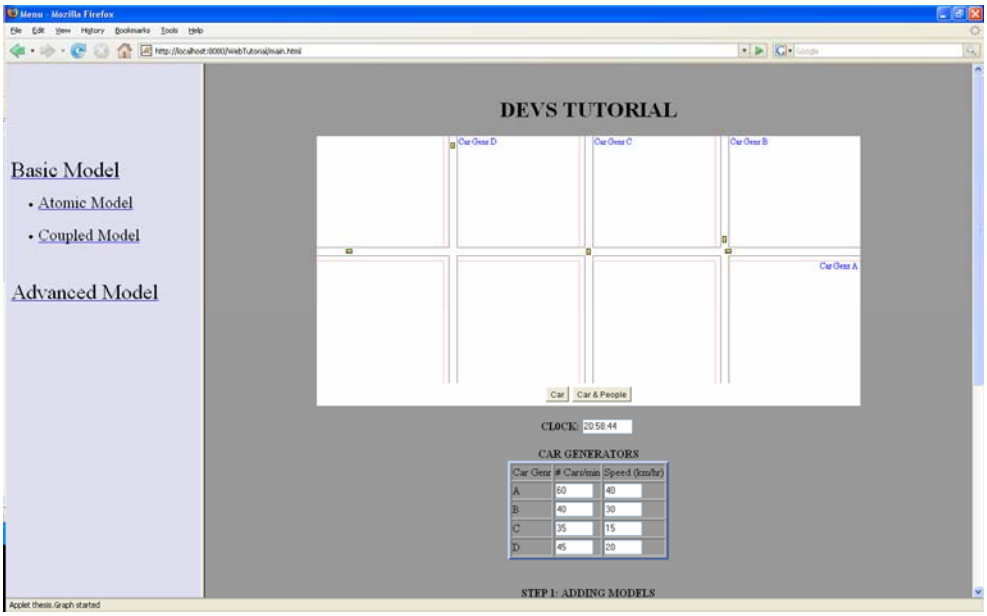

Figure 4.10 A Tutorial for Coupled Model 1

Step 2. As can be seen in figure 4.11, user adds LIGHT1 and LIGHT3 and couplings them together through input port "SIGNAL" and output port "OUT"

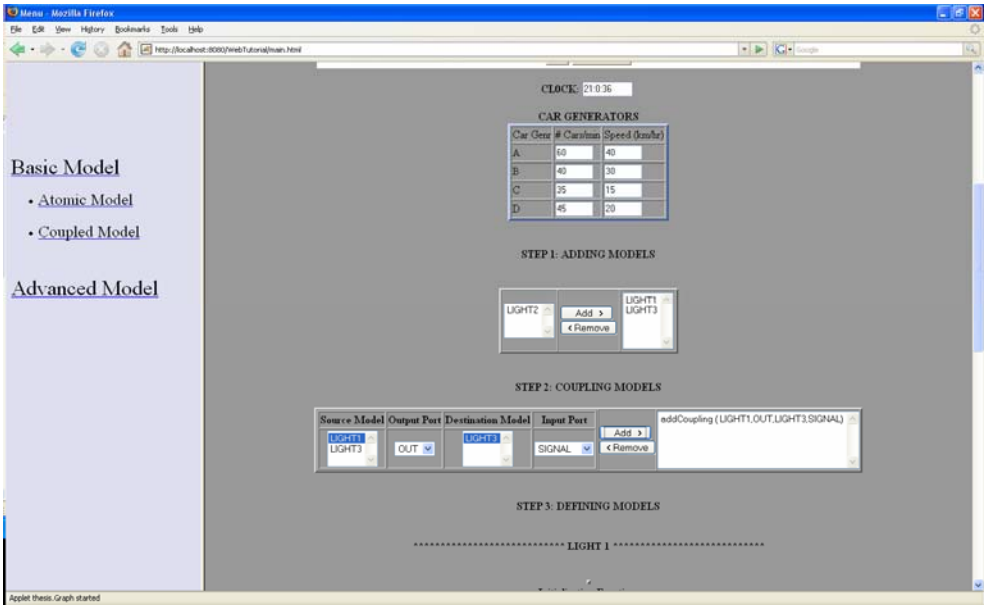

Figure 4.11 A Tutorial for Coupled Model 2

Step 3. Then, for each of the traffic lights, user will define all of the functions that need to be implemented in order to create each of the atomic models.

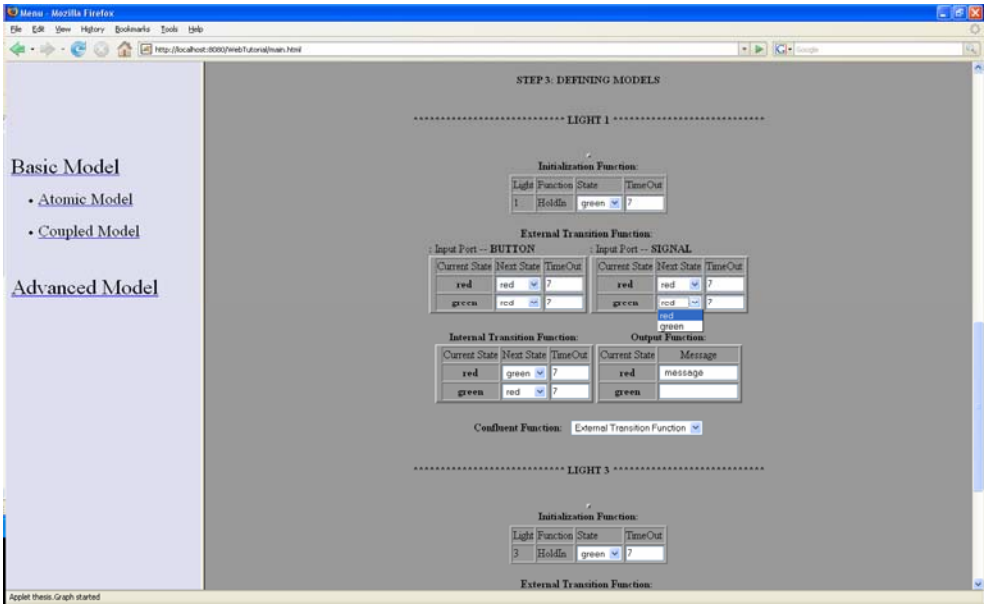

Figure 4.12 A Tutorial for Coupled Model 3

Step 4. Click on the "submit" button and the new simulation begins.

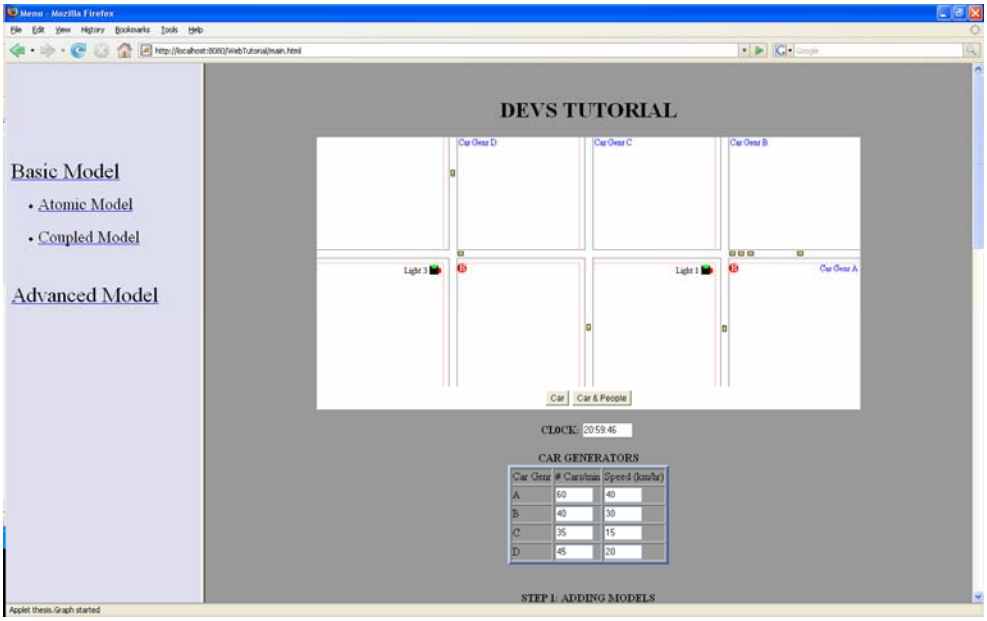

Figure 4.13 A Tutorial for Coupled Model 4

Step 5. Click on the link "View Java Simulation Code". User will be able to see all of the source code of DEVSJAVA running on the server.

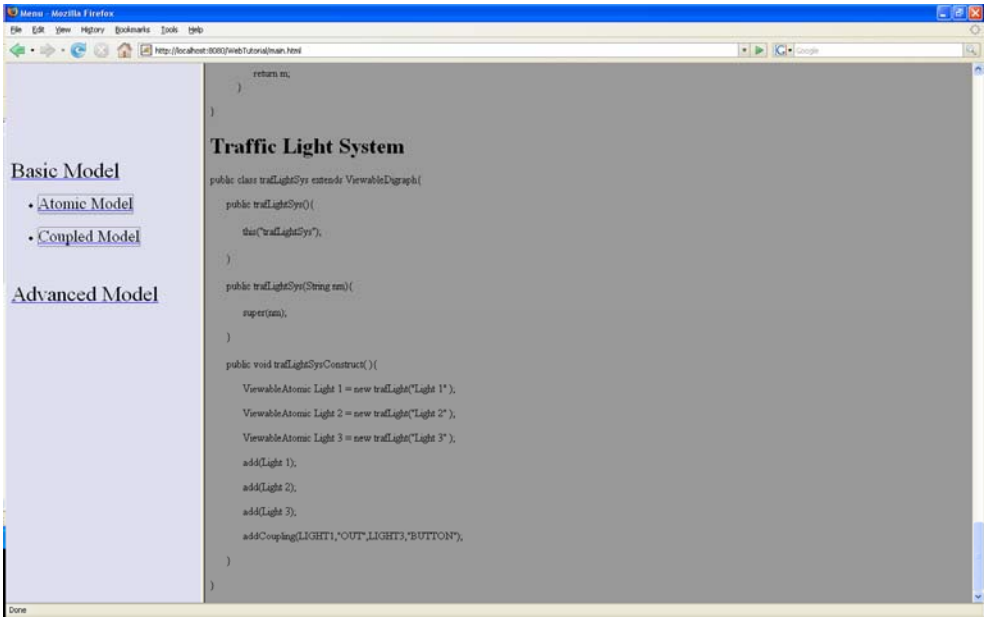

Figure 4.14 A Tutorial for Coupled Model 5

4.4 DEVS Resource

Useful Links:

- *ACIMS website* - http://www.acims.arizona.edu/
- *Download DEVSJAVA* - http://www.acims.arizona.edu/SOFTWARE/software.shtml
- *DEVS Tools* - http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm
- *DEVS Modeling and Simulation Tutorials* - http://www.acims.arizona.edu/EDUCATION/education.shtml#c.sim

Papers & Slides:

- *Introduction to DEVS Modeling & Simulation with JAVA, Bernard P. Zeigler, Hessam S. Sarjoughian, Arizona Center of Integrative Modeling and Simulation, 2003* -

http://www.acims.arizona.edu/EDUCATION/ECE575Fall03/notes/Manuscript_MSDJ_090903.pdf

- *Introduction to Modeling and Simulation with DEVS, Gabriel A. Wainer, Carleton University.* - http://www.cs.gsu.edu/DEVS/TutorialSpringSim.ppt
- *DEVS Introduction* - www.acims.arizona.edu/EDUCATION/ECE575Fall03/Notes/Ch1Intro.ppt

Textbook:

- Zeigler, B.P., T.G. Kim, and H. Praehofer.: Theory of Modeling and Simulation. 2 ed. 2000, New York, NY: Academic Press
- [Zeigler, B.P. and H.S. Sarjoughian. 2003. Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models, Technical Document, University of Arizona.
- Y. K. Cho, X. Hu, and B. P. Zeigler: The RTDEVS/CORBA Environment for Simulation-Based Design Of Distributed Real-Time Systems, Simulation: Transactions of the Society for Modeling and Simulation International, 2003, Volume 79, Number 4
- Zeigler, B.P., Y. Moon, D. Kim, J.G. Kim. 1996. DEVSC++: A high performance modelling and simulation environment. Proceedings of the 29th Annual Hawaii International Conference on System Sciences, 1996

5. SYSTEM ARCHITECTURE AND SOFTWARE DESIGN

This chapter describes the system architecture and software design. The system architecture will include Web Server, Servlet, APPLET , and DEVSJAVA. We also explain how web application is implemented and embeded to this tutorial project.

5.1 System Architecture

The uniqueness of this tutorial project is the integration of information technology and multimedia into education through the development of an interactive tutorial. The tutorial, developed to include multimedia modules that are interactive and knowledge-based, teaches users about the basic aspects of discrete event system and simulation. The interactive tutorial fully utilizes the power of the information and multimedia technology, web application and the programming language Java, to enhance students' learning to achieve rich interactivity. The tutorial contains an applet to enhance the delivery of information. The tutorial greatly supports human-computer collaboration to enhance learning and to satisfy user goals by effectively allowing the user to interact with the system in real time like a private tutor.

The advent of the World Wide Web (WWW) has provided researchers with a wealth of new techniques for sharing, displaying, navigating and indexing information. Portable and highly intuitive user interfaces can be easily built using HTML and the Java language.

The field of Software Engineering has much to gain from the WWW. Many of the goals of the field can be achieved in a better way by utilizing the facilities provided by the web. Providing access by the developers, for example, is greatly facilitated by the familiar user interface of well-known web browsers. Architectural diagrams of software systems can "come

alive" with the use of clickable maps.

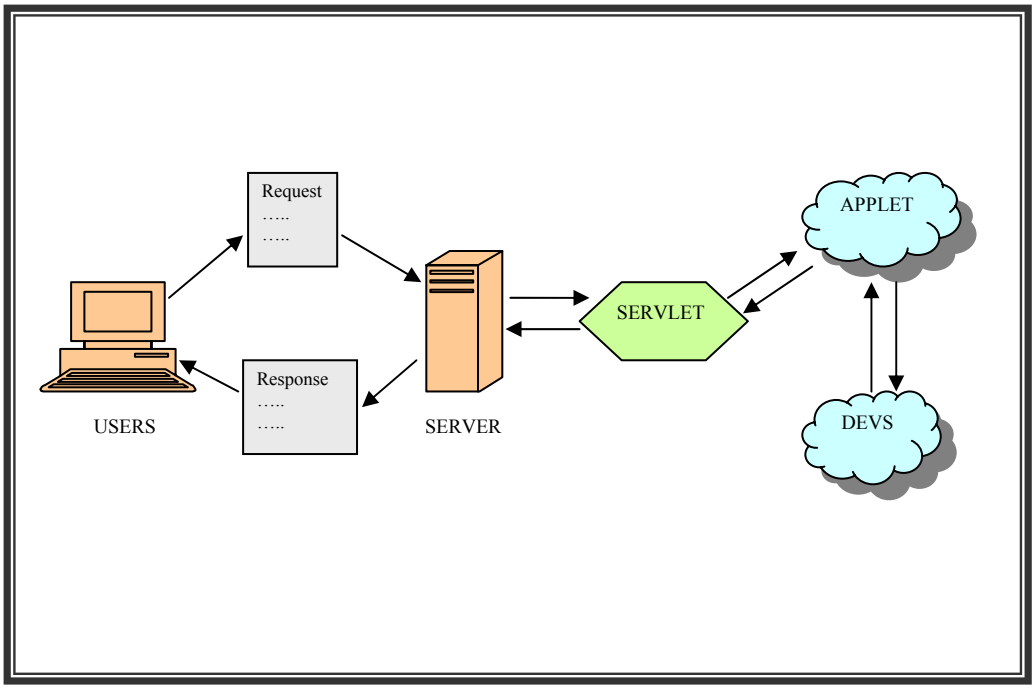

Figure 5.1 Web Server J2EE Architecture

There are a lot of reasons why web technology has undergone such an impressive growth in the past few years. One of the main reasons is the fact that developing software for the web is a standardized procedure, in that it uses common notations and tools, such as HTML, Java and CGI scripts. This in turn makes the software easier to develop and more portable. Also, the standardized user interface makes web software easier to access. Other reasons include the fact that no installation is required (other than a standard web browser), as well as the hot links between documents that make it easier to acquire a better overview of the subject of interest. This project benefited from all these factors, which helped in approaching the developers and getting feedback on how to improve our display of architectural information.

Web applications generate dynamic HTML-based WebPages accessed through a web browser. Since WebObjects applications are object-oriented and written in Java, the application

generates WebPages by creating instances of web components. A web component is a combination of a Java subclass of WOComponent and an HTML template. Our web components contain standard HTML elements and components including JavaScript programs, and Java Applets.

As can be seen from figure 5.1, a user first clicks a link in the browser. Browsers will format the request and sends it to the server. Server finds the requested page. In our application the request Html page will contain components of Web pages, Java applet, Java Servlet and DEVSJAVA objects. Then, server formats the response and sends it back to the users (browser). Finally, browser gets the HTML and renders it into a display for the user.

This tutorial, we use Java Servlet to develop and build our web application. The Java Servlet architecture [14] provides an excellent framework for server-side processing. Because Java is are written in Java, they can take advantage of Java's memory management and rich set of APIs. Java Servlet can also run on numerous platforms and HTTP servers without change.

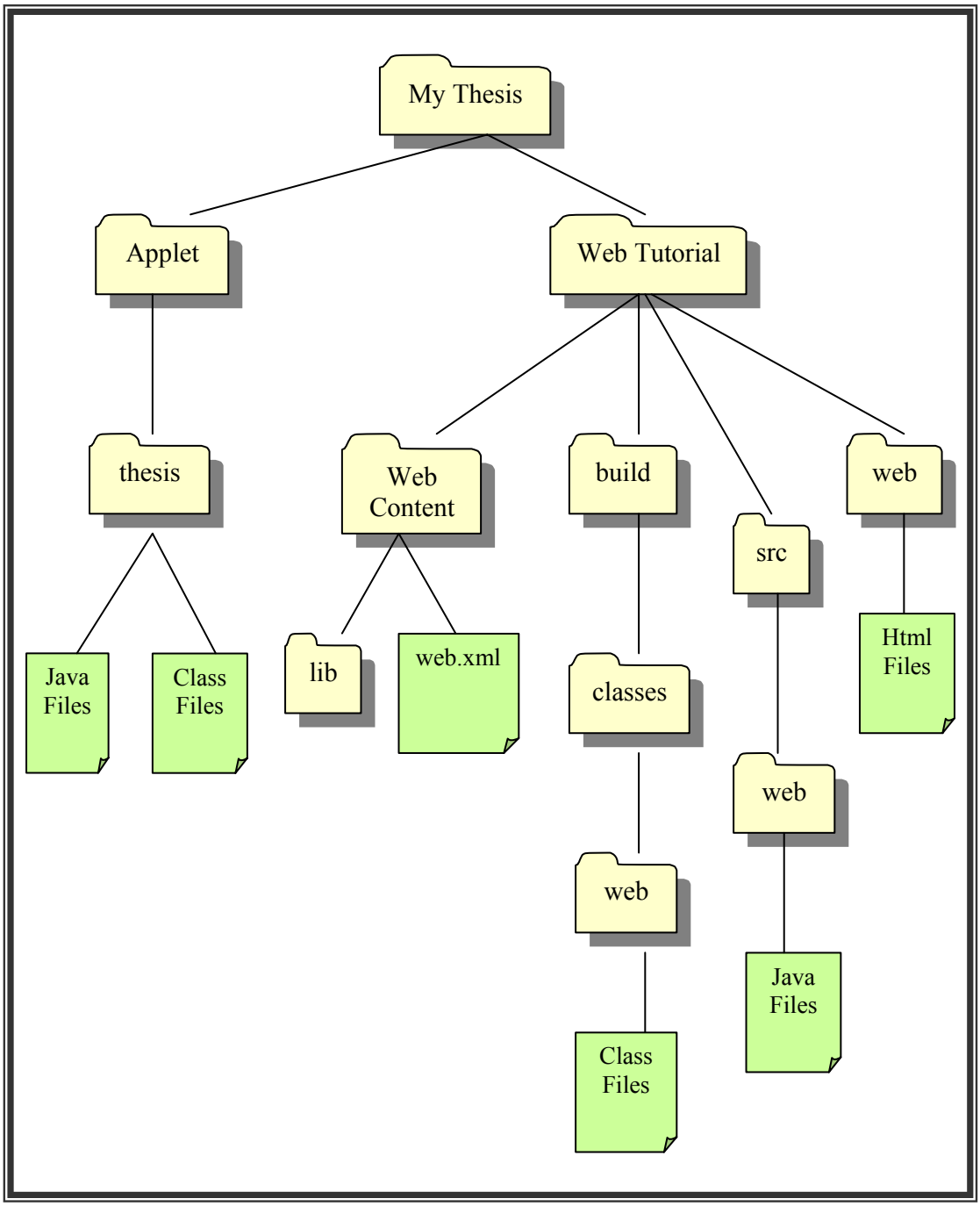

Figure 5.2 Web Application Development Environment

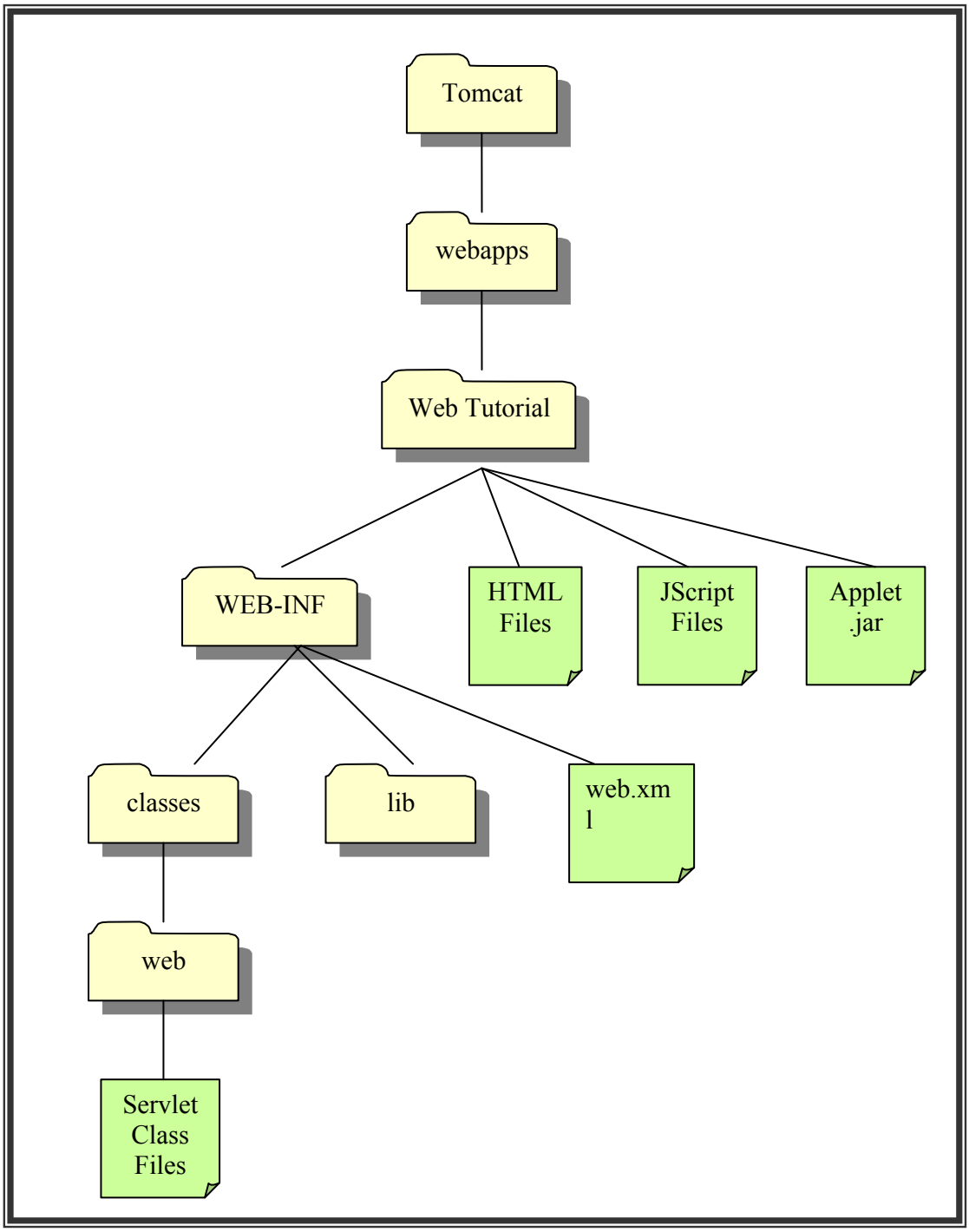

Figure 5.3 Web Application Deployment Environments

Further, the Servlet architecture solves the major problem of the CGI architecture: a Servlet request spawns a lightweight Java thread, while a CGI request will spawn a complete

process. Unless the project has a large budget for high-end Java-based application server projects, Servlet offers the best technology to develop server-side components and Web-based applications.

Apache Tomcat is the Servlet container that is used in the official Reference Implementation for the Java Servlet. It has been used as the web server for our web application. Figure 5.2 shows Web Application Development Environment [15]. In fact, there are lots of ways we could organize the development directory structure, but this is recommended for small – and medium – sized projects. All of the Java code will be under the src directory. The lib folder is where we put all JAR files. The directory structure under classes folder is derived when we compile our Java classes. All static and dynamic view components like HTML files, Java Script files or JSP files will go to web folder.

As can be seen in figure 5.3, Web Application Deployment Environment [15], deploying a web app involves both Container-specific rules and requirements of the Servlets specifications. The first three folders on top are Tomcat – specific. This part of the directory structure is required by Tomcat, and it must be directly inside the Tomcat home directory. Everything below this but above web folder is the webapp which is part of the Servlets specification, and will be the same regardless of the container vendor. Everything below classes folder is Application-specific. This package must be put immediately under WEB-INF/classes.

5.2 Software Design

This part describes the software design specification. We also discuss about the structure of our web application and the relationships between them. Our objective is to create web-based tutorial that integrates an APPLET and DEVSJAVA.

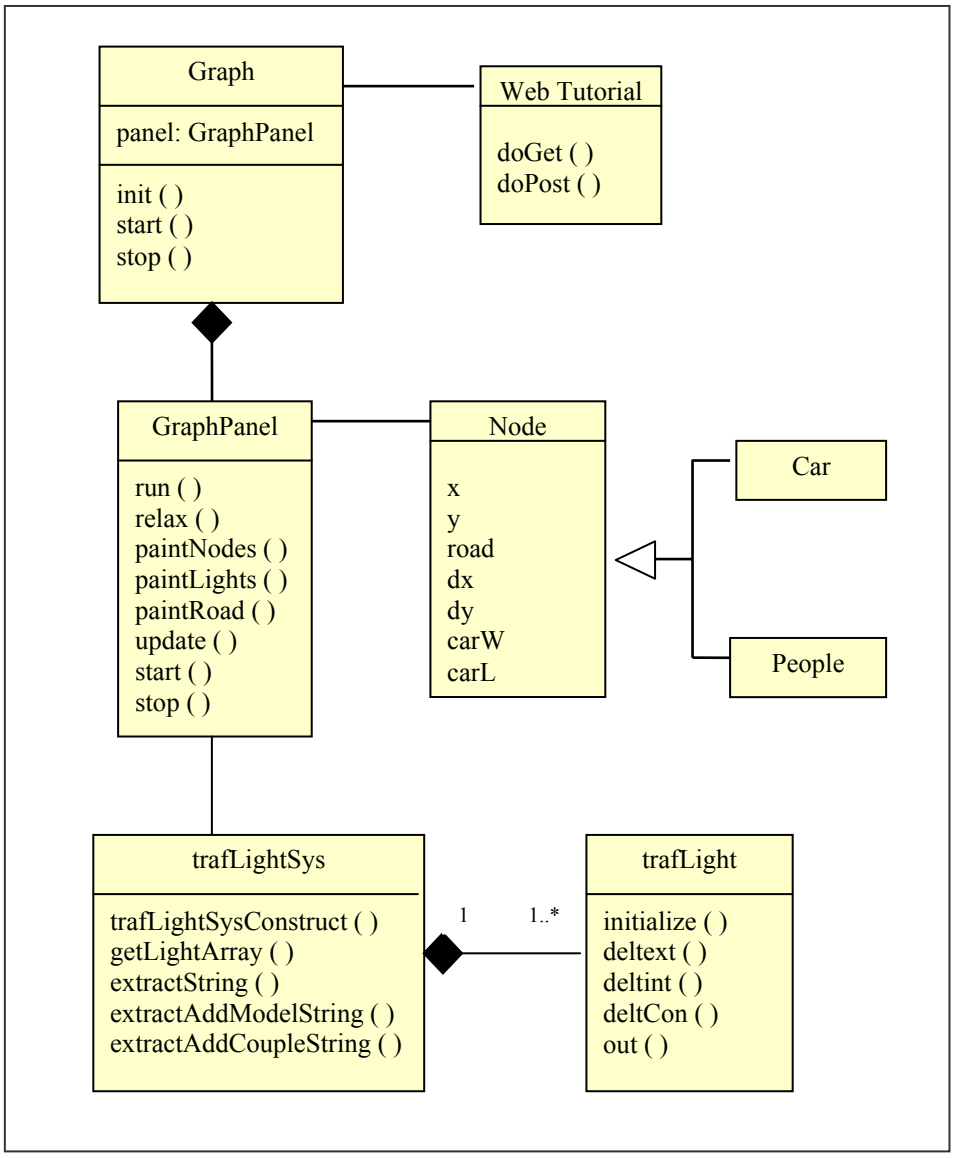

Figure 5.4 Class Diagram of Web-based Tutorial Software

Figure 5.4 shows a class diagram describing the structure of all the web application. The whole web application system contains six classes; one Servlet class, three applet classes, and two DEVS classes.

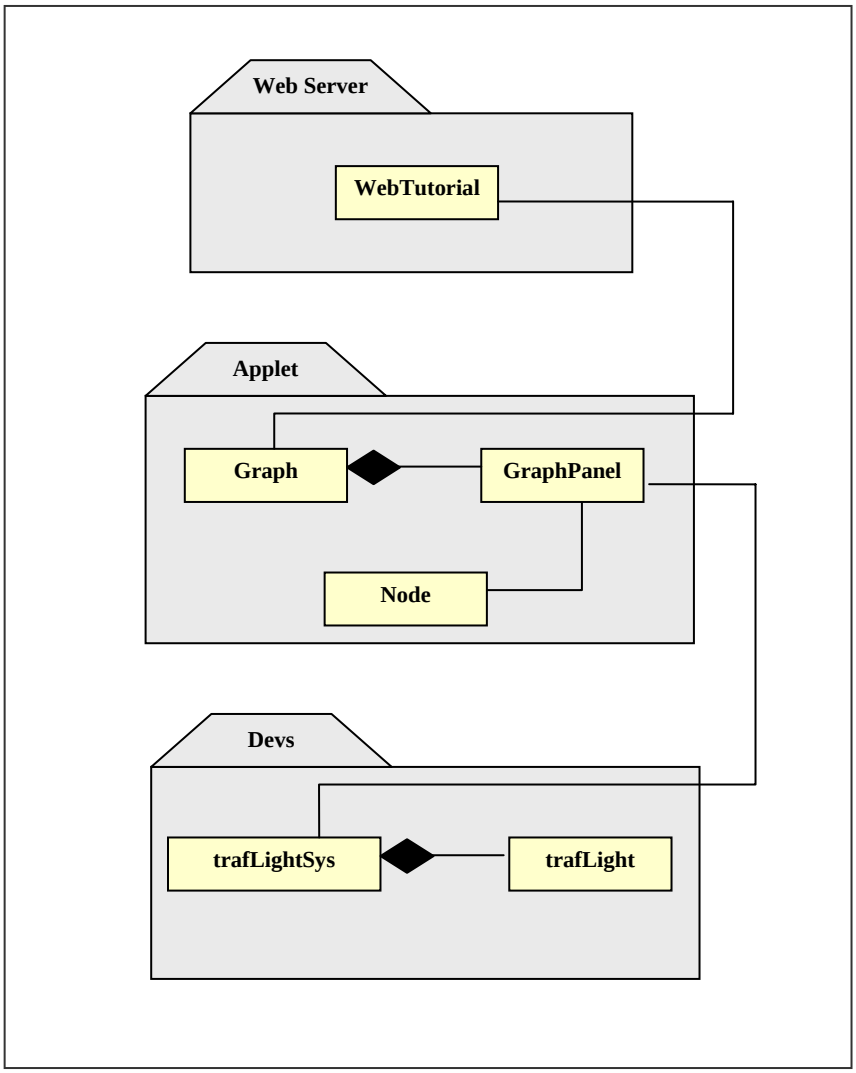

Figure 5.5 Subsystem Architecture

As can be seen in figure 5.5, the system is composed of three subsystems. The system has been implemented in a dynamic web application (Java and Jscript in Active Server Pages on an Internet Information Server). The system architecture has been designed with emphasis on a high level of flexibility among subsystems.

During the system design phase, the main idea is the separation of Presentation (user interface) from business logic.

Web Server Subsystem

- Web Tutorial (Servlet) – is module of Java code that run in a server application and service HTTP requests as it implements the javax.servlet.Servlet interface.

Applet Subsystem

- Graph and GraphPanel – extend the Applet class. GraphPanel also implements Runnable. These applet classes are used to provide interactive simulation to web applications. They are executed by web browser and run on client side.
- Node – is a regular class that is used to represent cars and people.

DEVS Subsystem

- trafLight – extends ViewableAtomic and represents one Atomic model in DEVS specification.
- trafLightSys – extends ViewableDigraph and represents a system or a group of atomic models in DEVS specification.

Figure 5.6 is a sequence diagram of our web application describing the relationship and interactions among a set of objects. From the user interface (Web Tutorial), once users input applicable parameters and click on the "submit" button, an applet (Graph) is initialized and pass all parameters to DEVS object (trafLightSys), as well as initialize another applet (GraphPanel).

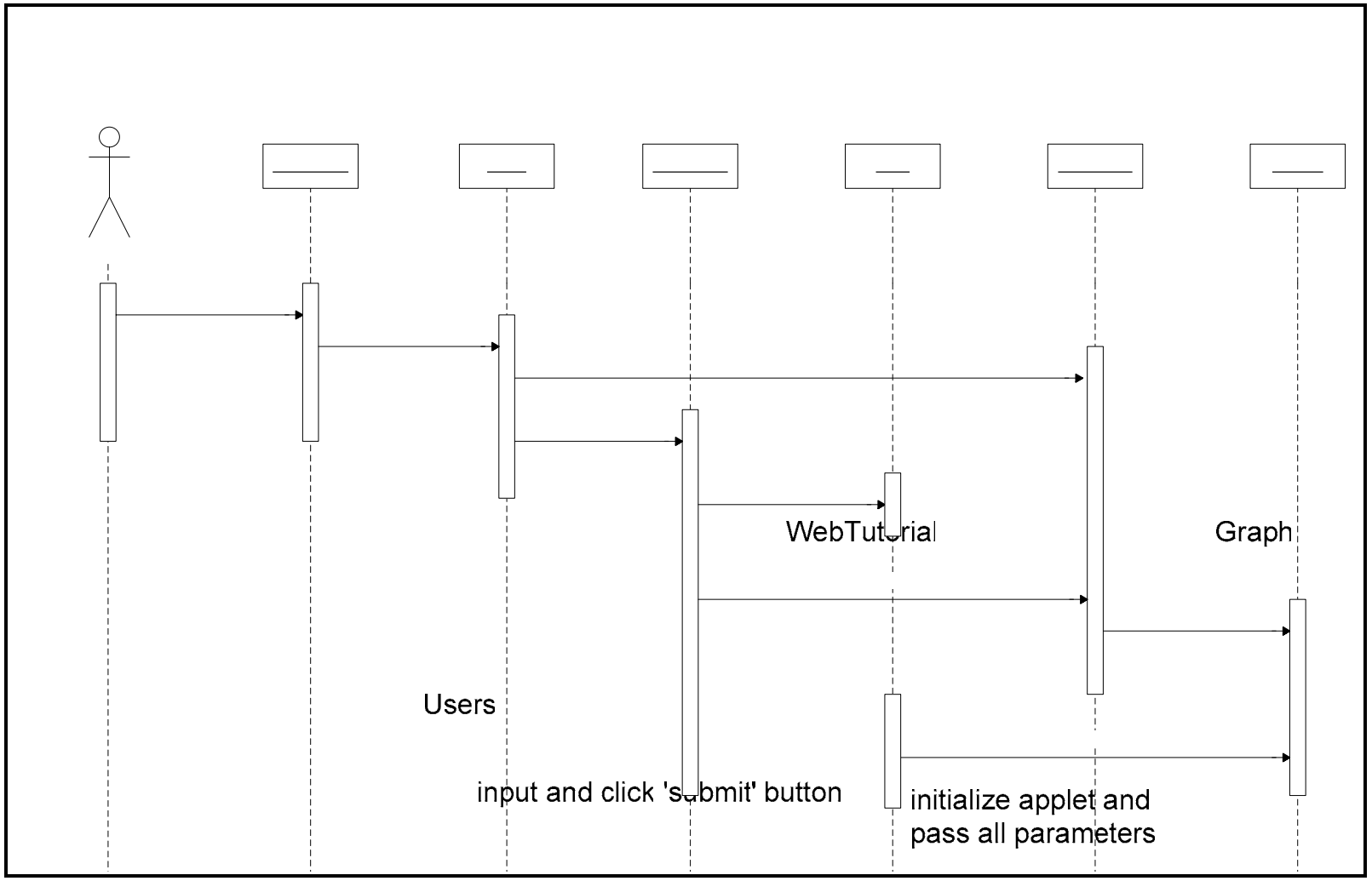

Figure 5.6 Sequence Diagram of Web-based Tutorial Software

initialze

The Node objects are generated from this applet as well as the system DEVS object (trafLightSys). Then, the trafLightSys object will create all applicable trafLight objects by passing all parameters that are specified by users to the constructor of trafLight class.

During the simulation, an applet GraphPanel which implements Runnable will keep an applet Graph refresh which causes all nodes moving as well as changing the traffic light's color as a thread is running throughout the simulation.

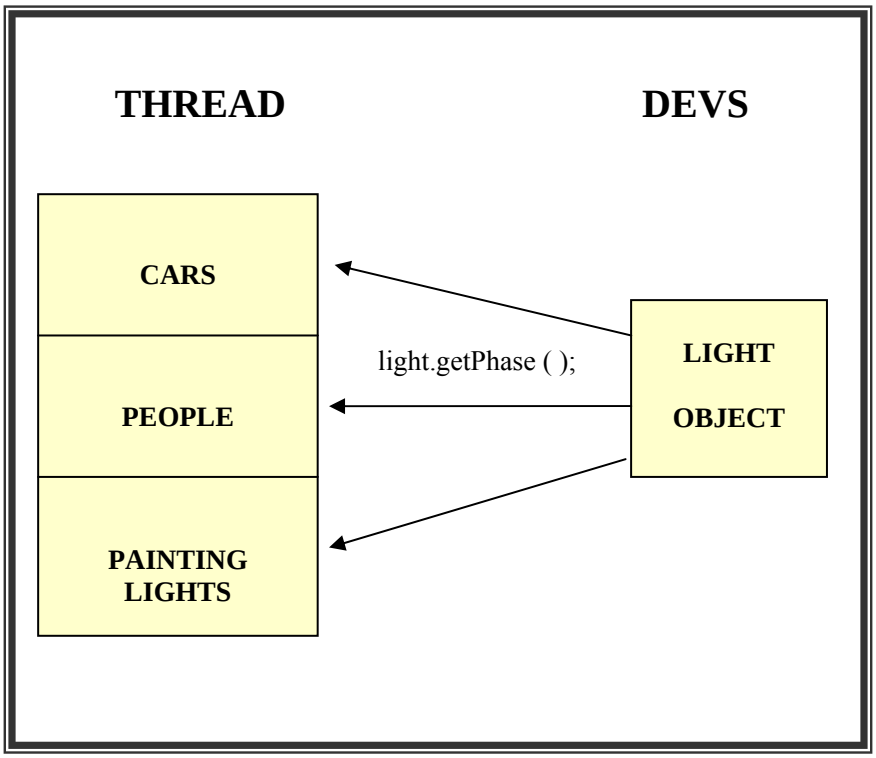

Figure 5.7 Relationship between Thread objects and DEVS objects

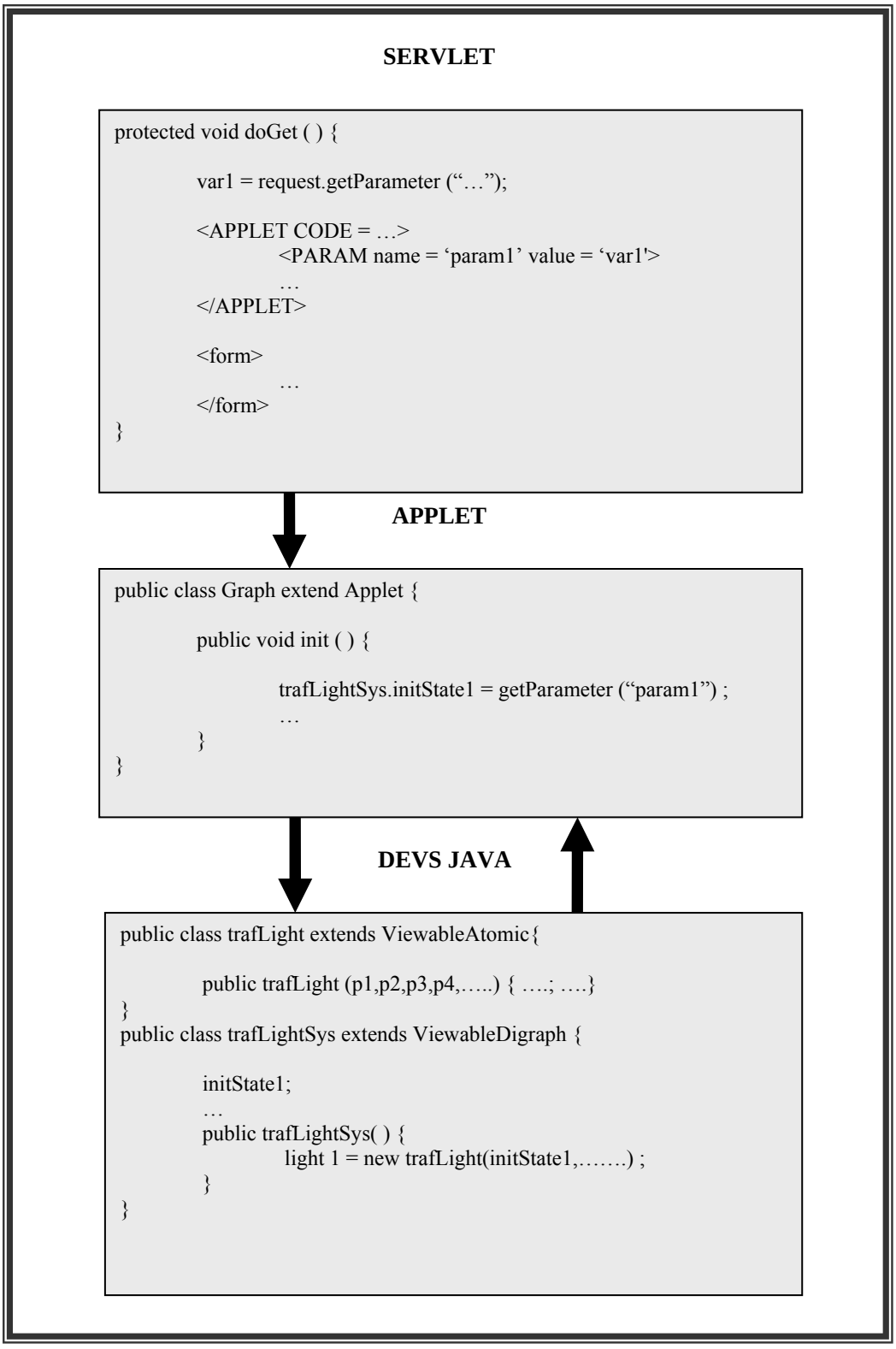

Figure 5.8 Relationship among three Subsystems.

As all nodes are moving through out the simulation and passing all traffic lights at each intersections, light.getPhase ( ) will be called to determine if the node should stop or keep going as be illustrated in figure 5.7. Not only this method is used by Node object, all traffic lights will also call light.getPhase ( ) to indicate their color (red, green, or flashing red).

As can be seen in Figure 5.8, the sample of codes shows us how the parameters are passed from the client side user interface which is a Servlet class 'WebTutorial' through an applet class 'Graph' which then pass it to the DEVS object 'trafLightSys'. Finally, the new traffic light objects are created. This happens every time user clicks on the 'submit' button.

6. CONCLUSION AND FUTURE WORK

In this thesis, we introduce a new initiative to provide a suite of online tools for learning DEVS model. This project consists of HTML, JavaScript, Servlet, DEVSJAVA, and Java Applet which is used for interactive animation and simulation. This project is specifically concerned with the teaching and learning process in DEVS model and this program of work has addressed some of these problems through the development of interesting and appealing interactive web-based software to stimulate and facilitate student learning in DEVS model. Another strategy known as contextualized learning has been integrated to the tutorial in order to increase motivation to learn. With this strategy, learners learn more effectively by integrating new knowledge/skills into already existing knowledge. As a result, a new form of teaching, contextualized web-based tutorial is developed to help students gain more clearly understanding of the DEVS formalism. We exemplify the DEVS formalism atomic and coupled models using simple traffic light controller system. We implement the traffic simulation system based on the DEVS theory. Our model of intersection traffic is a simplified version of real-world intersection traffic. This will help students to gain more and more clearly understanding of the DEVS formalism by comparing to the real world system.

There are many issues remaining for future work that should be addressed, and many improvements are needed to increase quality of this tutorial.

1) The implementation of DEVSJAVA Webstart, a web-enabled DEVSJAVA simulation environment: This will definitely help users to gain more understanding by comparing SimView and the simulation interface. Even though, users are able to easily see how the

2) state is changed, and how to inject an input but they are unable to see the remaining time and the message passing between models by looking at the simulation interface.
3) Have two-stage learning for the Atomic model. First stage is the basic ones. Second stage will learn two more things which are elapse time (e) and remaining time sigma (w) in external transition function.
4) Have an animated view of state-time, elapse time-time, input-time, and output-time.
5) Have an analysis window to show the statistics of the simulation

REFERENCES

[1] Petri Nets World (2005). Retrieved October 10th, 2007, from http://www.informatik.uni-hamburg.de/TGI/PetriNets/

[2] Wikipedia, "Finite State Machine" (2007). Retrieved October 10th, 2007, from http://en.wikipedia.org/wiki/Finite_state_machine

[3] Wil van der Aalst, V. Almering, and H. Wijbenga (2005). Interactive Tutorials on Petri Nets. Retrieved October 4th, 2007, from http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/

[4] Volker Guth (2000). Petri Net Applet. Retrieved October 5th, 2007, from http://is.tm.tue.nl/staff/wvdaalst/pn_applet/pn_applet.html

[5] Matt Chapman (1996).Finite State Machine Explorer. Retrieved September 15th, 2007, from http://www.belgarath.org/java/fsme.html

[6] Carnegie Melon University (1999). The Alice software programming Tutorial. Retrieved September 7th, 2007, from http://www.alice.org/

[7] University of California (2007). SimSE Online: an educational software engineering simulation environment. Retrieved September 27th, 2007, from http://www.ics.uci.edu/~emilyo/SimSE/

[8] Zeigler, B.P. and H.S. Sarjoughian (2005). Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models, Technical Document, University of Arizona.

[9] Zeigler, B.P., T.G. Kim, and H. Praehofer (2000). Theory of Modeling and Simulation. 2 ed. New York, NY: Academic Press

[10] Y. K. Cho, X. Hu, and B. P. Zeigler (2003). The RTDEVS/CORBA Environment for Simulation-Based Design Of Distributed Real-Time Systems, Simulation: Transactions of the Society for Modeling and Simulation International Volume 79, Number 4

[11] Zeigler, B.P., Y. Moon, D. Kim, J.G. Kim. (1996). DEVSC++: A high performance modeling and simulation environment. Proceedings of the 29th Annual Hawaii International Conference on System Sciences, 1996.9

[12] Jeff Mather (2003). The DEVSJAVA Simulation Viewer: A modular GUI that visualizes the structure and behavior of hierarchical DEVS models.

[13] B. P. Zeigler (2003). DEVS Continuous Systems. Retrieved September 27th, 2007, from http://www.acims.arizona.edu/EDUCATION/ECE575Fall03/notes/DEVSContinuousSys.doc

[14] Servlet Essentials (2005). Retrieved September 27th, 2007, from http://www.novocode.com/doc/servlet-essentials/chapter1.html

[15] B. Basham, K. Sierra, B. Bates (2004). Head First Servlets & JSP, O'Reilly