DESIGN AND ANALYSIS OF

VIEW SYNCHRONIZATION IN DEVS-SUITE

by

Eric Joseph Helser

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

May 2009

DESIGN AND ANALYSIS OF

VIEW SYNCHRONIZATION IN DEVS-SUITE

by

Eric Joseph Helser

has been approved

April 2009

Graduate Supervisory Committee:

Hessam S. Sarjoughian, Chair
Hasan Davulcu
Stephen S. Yau

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Simulation software tools allow users to model and test processes in a graphical environment on a computer. In generating models to be simulated, data visualization is a useful tool for model verification and validation. This thesis addresses the problem of synchronization between data generation and visualization. This thesis introduces a synchronization mechanism within the Model-Facade-View-Controller (MFVC) architecture without compromising the simulation engine's validity or efficiency, which current simulators lack. The approach to this topic includes researching background on the MFVC architecture pattern, and other simulation tools' approaches to data visualization. After implementing synchronization, experiments were conducted to compare execution speed under different conditions. The synchronization method presented here restricts simulation speed to no faster than the visualization rate. While the synchronization feature is added to only one simulation tool, the approach used can be generalized and applied to other simulators.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Figure Page

1 Introduction

Some simulation software tools allow users to model and test various real-world and abstract processes in a graphical environment on a computer. Models can range in complexity anywhere from one component changing its state at certain intervals to interactions between the inhabitants of an entire city, and beyond. Simulations are constrained by the software and hardware executing them. While software suites and packages contain generally the same set of user interface features, there has been little research done on comparing the speed and memory efficiency of those graphical features.

1.1 Overview of Modeling and Simulation

It is highly desirable for simulation tools to allow users to observe the structure and behavior of a real system. Certain basic features are necessary in a simulator for a user to be able to interact with the system.

First, there needs to be a data repository where the information relevant to the models, or components being simulated, will be contained. Second, we will also need to be able to see what is happening within the system. A graphical user interface (GUI) that is loosely coupled with the data repository will be needed to display the data to the user. The reason it must be loosely coupled is that we should be able to easily replace or make changes to the GUI without needing to alter any other components. The last important basic feature to a simulator is the ability to control the simulation execution. The user should have the option to automatically run the virtual system from start to finish, or in small increments. To accomplish this, we add some controller features to the GUI, such as buttons, that will interact with the simulator. This overall concept of dividing the

simulator into separate, loosely coupled component is a design pattern called Model View Controller, and will be discussed later in this thesis.

There are many simulation tools currently available that offer these basic capabilities. For example, some well known simulation tools include Ptolemy [1] and Simulink [2]. These all feature similar capabilities to store, display, and control the simulation data. However, to our knowledge, DEVS-Suite, which is based on a formal modular, hierarchical modeling framework, is the only simulation tool available today that offers multiple, simultaneous views of the model information.

DEVS-Suite is a Java-based application used to represent models and their interactions in a graphical, interactive environment. It supports simulation of models described according to the Discrete Event System Specification (DEVS) [3]. Each model is visually represented by one of two basic shapes: a filled rectangle for basic components, or a rectangular outline for hierarchical models that are composed of one or more inner components. Each component may have input and output ports, which are used to transmit messages between components. All visualization systems surveyed by Mather have these basic capabilities [4], and some tools support hierarchical modeling, such as Ptolemy [1] and Simulink [2]. The DEVS-Suite framework (including DEVSJAVA) is the only one to support the visualization of hierarchical models in a single view [4]; Ptolemy and Simulink require separate displays for each level of the hierarchy. From the simulator perspective these tools (DEVS-Suite, Ptolemy, and Simulink) are distinct and serve different purposes [5].

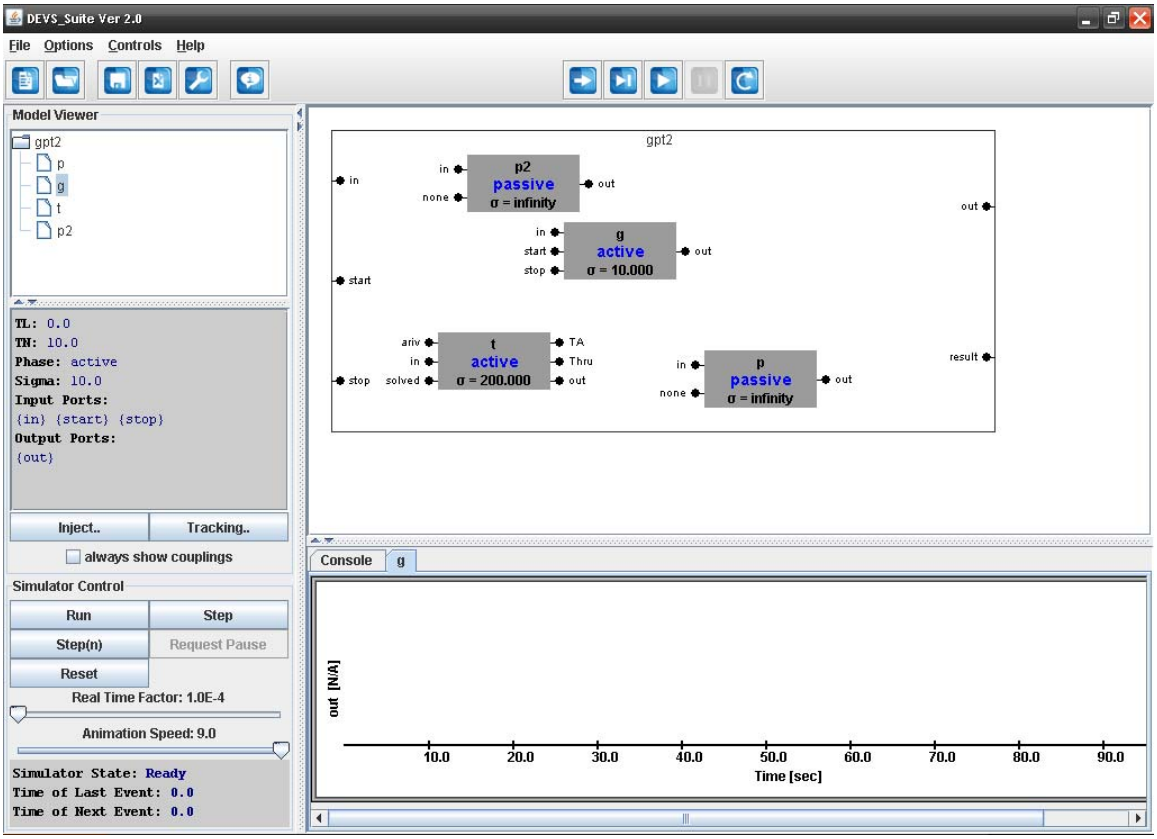


Figure 1: DEVS-Suite Tracking Environment

Figure 1 shows an example of the DEVS-Suite interface. There are four main sections to this screen: the Model Viewer, Simulator Control, SimView, and Tracking Window. After loading a model, the Model Viewer in the top left corner is populated with a list of the components, both atomic and coupled, contained in this model. Immediately below the component list is a box that lists the predefined variables pertaining to the model selected by the user. In Figure 1, we can see that the g component has three input ports, one output port, and an event that is set to occur at t=10. This box will be updated whenever the user selects another component. Immediately below that are two buttons: Inject, for manually providing data at arbitrary time points in the simulation; and Tracking, which users use to initialize data visualization windows.

The SimView on the top right displays the model visually, including any hierarchical components. This particular model, gpt2, contains four atomic components contained in one large coupled component. Below that is the Tracking Window, which contains the standard output console by default as well as any TimeView tabs, such as g in Figure 1. Finally, in the lower left, there is the Simulator Control. From here, the user can control the actions of the simulator.

Along the top of the window, there are two groups of buttons that act as shortcuts for the menu options. The left group controls files: New Model, Load Model, Save Console, Clean Console, Console Setting, and About. The right group only appears after a model has been loaded, and includes: Step(n), Step, Run, Pause, and Reset. These are analogous to the buttons provided in the Simulator Control.

DEVS-Suite offers a number of buttons and sliders for the user to control the simulator (see Figure 1). The first slider, Real Time Factor, controls how fast the logical time of the program progresses in relation to “wall-clock” time. This variable can adjust the scale of simulation logic time in order to get a faster, slower, or even soft real-time response [5]. If this number is set to 1, then each unit of time in the program will take one second to pass. When this is set to 1e-4, ten thousand units of time will pass in one second. The other slider, Animation Speed, determines how quickly messages will move around the screen between components. This slider ranges in value from 1 to 9, with 9 being the fastest. Both of these program parameters are subject to hardware and software limitations of the system.

Simulation execution time is defined in terms of logical time, which is defined as a real number between zero and infinity. In order to cut down on unnecessary processing,

the program calculates the next state change in the simulation and skips to it, since nothing relevant happens between those state changes. We refer to each skip in time as a "step."

Near the sliders, there is a group of buttons that controls the behavior of the simulator. The first button, Run, tells the simulator to automatically step through the model interaction until the user manually halts execution or no more state changes exist. Next, the Step button runs the program until a state change occurs, and then returns control to the user. A similar button is Step(n), which lets the user decide how many steps to run in sequence. Note that for Step and Step(n), all messages generated by the components will be animated on the model display, while when the user clicks Run, no messages will be displayed. The fourth button, Request Pause, is only enabled after the user has clicked Run, and is used to stop the simulator. Once the user pauses simulation, he may select to either Step through a fixed number of iterations, or Run the simulation again. The last button, Reset, resets the components to their initial states. This option is available at any point except when the simulation is running.

After each step, new information must be presented to the user. One update that occurs at each step is a state change. The state is printed in the middle of the components, and is always visible. When the user clicks Step or Step(n), the simulator also displays the animated messages that move across the screen. As stated earlier, these messages are not visible when the user clicks Run. If the user has selected to track any ports or components in the TimeView window, new information about their messages or states will be displayed there, as well.

DEVS-Suite handles the separation of concerns by processing component data and visualization separately. The models maintain the components' data, while different objects take care of the view. This is consistent with the MFVC (Model, Facade, View, Control) design pattern, and makes the models interchangeable without compromising the quality of the simulation interface.

1.2 Architecture of DEVS-Suite

The DEVS-Suite simulation environment uses the fundamentals of the MFVC software design concept. This defines four specific systems for the software, and how they interact. They are the Model, Façade, View, and Controller systems.

As the composite of two important subsystems, the Model is responsible for encapsulating core Modeling and Simulation logic [6]. The model and simulation engines exist within this layer and define all of the components, behaviors, and relationships of the application. This layer communicates with the next layer called Façade.

The Façade layer of the MFVC pattern acts as a mediator between the Model, View, and Controller. While the Façade does not add any new functionality to the Model, it does expose a consistent set of functionalities to the View and Controller, so that those components can interact with any Model effortlessly. The presence of this façade allows the Model to maintain its black box nature [7].

The View and Controller layers of this architecture enable human interaction with the simulator. First, the View is responsible for interacting with the Façade in order to obtain state information from the Model about the components to be visualized. The View then interprets these values and displays them to the user. In the case of DEVS-

Suite, this data is shown in the animated component windows as well as any TimeView graphs. The View must also provide ways of interacting with the system, such as text fields or buttons.

The user can interact with the Model through the Controller layer. Although the interaction methods are provided by the View layer of this architecture, we generally consider the Controller as the one responsible for interpreting these user inputs and communicating with the Model via the Façade. Once it receives a signal, such as Run or Step, the Controller will send an appropriate message through the Façade's exposed functionality, and on to the Model.

While this MFVC pattern has been used by DEVS-Suite and possibly other simulation tools to support the separation of concerns in an application environment, it has not been used to build a simulator that is both efficient and capable of supporting multiple, synchronized views, such as animated and graphical plots. At this point, DEVS-Suite can support both views at the same time, but they are not synchronized, especially as the scale of the model grows. To our knowledge, all existing simulators have major limitations in supporting different kinds of visualizing simulation dynamics in near-real-time.

In the course of this thesis, we will study the limitations of existing M&S tools, and propose a new software architecture that will allow configuration flexibility in terms of visualization and synchronization.

1.3 Problem Description

The problem we are addressing with this thesis regards the synchronization between data generation and data visualization. All of the tools surveyed thus far [4, 1, 2] lack the ability to synchronize simulation execution with data visualization. While the tools can produce graphs that accurately reflect the final results of the simulation, they are not produced concurrently with the simulation, often leading to the simulation finishing well before the output can be displayed to the user.

For the purposes of this study, we are defining synchronization as the condition where the simulation is no more than one logical step ahead of any of the views. We cannot constrain synchronization any further than this, because in order to display the data, it must be generated by the simulation first. Once this newly created information has been displayed on all of the appropriate views, the views will have caught up to the simulation, allowing the simulation to proceed with the next logical step.

We will show how it is possible to force the simulation and views to stay synchronized in this sense throughout the execution. This will allow the user to see the data that the simulation generates with minimal delay.

1.4 Contributions of this Study

We will demonstrate how our synchronization techniques for DEVS-Suite can be generalized and applied to any simulation environment. Our principles and design choices will be applied to the MFVC architecture so that any application that implements this design can easily integrate the synchronization shown here.

Visualization synchronization benefits end-users in multiple ways. The effects of synchronization are not limited to eliminating the lag between data generation and display. Forcing the simulator and views to stay in sync assists in the process of model verification and validation. The user can more easily verify the behavior of a model if he can see the data it generates while it is being simulated, rather than after the simulation has completed.

One specific benefit to implementing synchronization applies to simulated processors. A DEVSJAVA-based MIPS32 simulator developed by Yu Chen at ACIMS allowed students in an undergraduate computer architecture course to better learn and instructors to better teach MIPS processor designs [8, 9]. Synchronizing the data visualization with the model will further assist students to more easily understand how the data sets displayed on the screen relate to each other.

2 Background and Related Works

There are many papers that discuss how to make a graphical user interface (GUI) functional and intuitive to user as well as aesthetically pleasing, but there are few that explain the ramifications of interface design on reusability [10] as well as performance [11]. One paper [10] explains how design patterns such as model-view-controller (MVC) can be used to develop a simulation environment.

Once the models have been developed, the execution speed of the simulation software becomes a concern. Depending on the amount and method of visualization, the simulation may run quickly or very slowly. In their paper, Mitchell and Power [11] discuss one approach to benchmarking graphical simulation software by using a program to fill in the fields on a form and automatically click “submit,” thereby taking user error and hesitation out of the equation.

As a simulation runs, the computer must not only calculate results and keep track of the different models involved, but also display feedback and output to the user. Obviously, as the amount of output generated increases, the time to complete a particular set of tasks also increases.

The amount of this slowdown varies among different simulation engines. Some handle it better than others, but in each case, simulation execution speeds up when the program is left to run by itself without any intermediate output or animated displays.

There is also the problem of deciding how much information is too much to show the user. In most simulations, few options are provided to allow users restrict the amount or type of output generated and viewed during execution.

2.1 Model Display Issues

One way the simulation can provide information to the user is via model displays. For example, in DEVS-Suite, each model is displayed as a colored rectangle containing three lines of text: the name of the model, its current phase, and the amount of time until its next self-invoked phase change. Currently, this kind of view cannot be disabled via UI at arbitrary instance during simulation execution. Each time a new event occurs, the time value is updated, and the phase changes if necessary. In typical simulation runs with few to many millions of operations, there is a significant slowdown in overall processing speed when there is a need for run-time visualization.

2.2 Generic Output Display

The graph below demonstrates the effect of output display on the time required to execute a very simple program. This base case test consists of a Java program that performs one floating-point operation, and then prints any number of characters to the screen. These two steps repeat inside of a loop an arbitrary number of times, and then the program calculates the amount of time required per loop. This experiment was run on a machine running Windows XP Home SP2, with 2.8 GHz CPU and 960MB RAM.

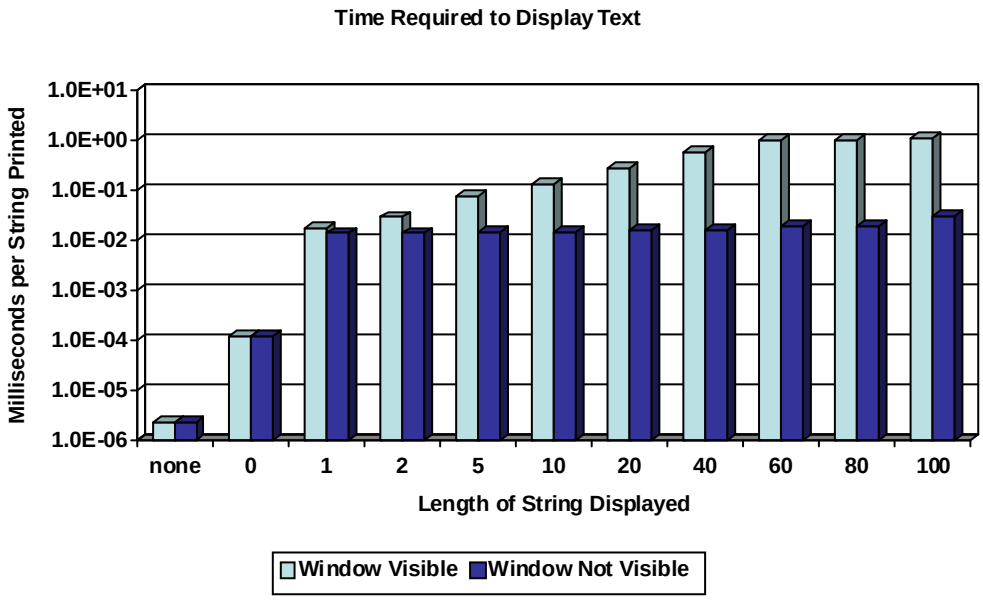


Figure 2: Time Required to Display Output

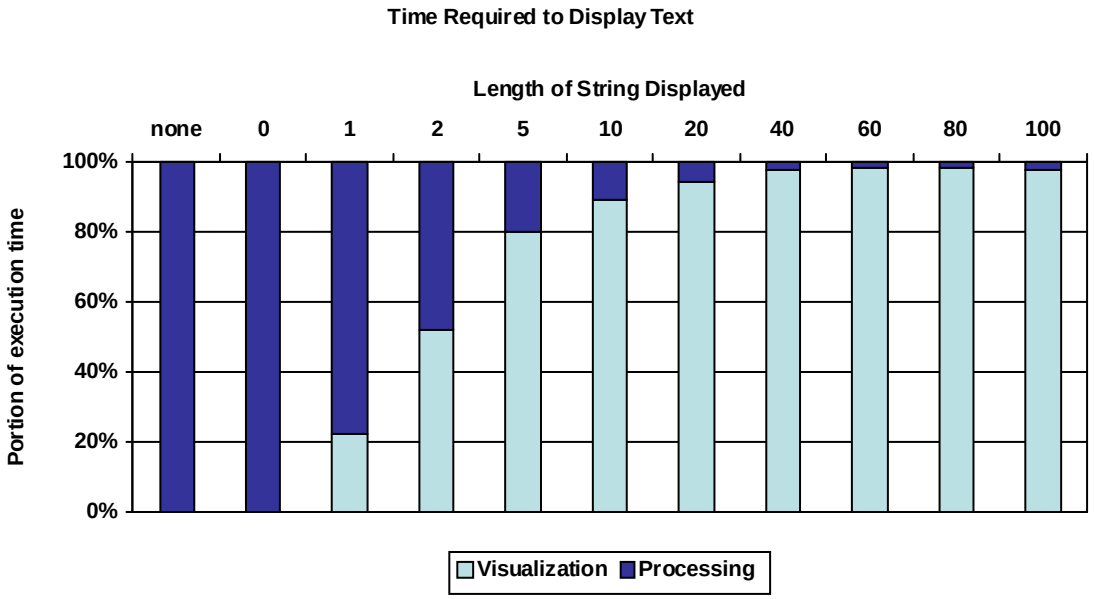


Figure 3: Portion of Execution Time Required to Display Output

The horizontal axis in Figure 2 shows the number of characters displayed per iteration of the loop, and the vertical axis shows the amount of time in milliseconds required to perform that iteration. The first entry on the horizontal axis, “none”, refers to

the case that there was no print statement at all, whereas the “0” entry refers to another case with an empty string (i.e., “”) inside the print statement.

The two values, “Window Visible” and “Window Not Visible” refer to whether the console window that displayed the output had focus (visible), or was hidden behind other windows (not visible). When the output was not being displayed to the user, the program ran at approximately a constant speed regardless of the amount of output. If the output was being displayed in real time, the program slowed down significantly as the amount of output increased. A string of length one or two did not seem to have an effect on the overall execution speed, but at ten characters, the program began to show significant slowing. As the output string length increased, the time required appears to be directly proportional to the string length.

Figure 3 reorganizes this data to show proportionally how much time was spent in processing and displaying the data. The “Window Not Visible” time was used as the processing time, and the difference between “Window Visible” and “Window Not Visible” was used as the displaying time, as visualizing the text required both processing and displaying. We found that as the length of the string increased, nearly all of the time needed to visualize each string was spent in the displaying phase.

This simple test shows us two things: there is a significant time-cost for displaying any output at all, and then there is extra time required to display the output, depending on its length. The program with no output functionality at all ran by far the fastest of any test. When the print statement was introduced, even with a zero-length string, execution slowed considerably. In the next test, when the program displayed a single character, execution slowed even further. With longer strings (two or more),

execution speed then depended on whether the output window was visible or not. If it was hidden, execution speed remained relatively constant. Otherwise, the program took an amount of time directly proportional to the length of the string.

The results of this experiment may have an impact on our approach to studying efficient data visualization techniques. Since this experiment was purely text-based, there may be some differences in results when moving to a graphical environment. However, we expect the overall concepts of this trial to hold for graphical user interfaces that can be used in simulation tools. A GUI that has no output should require the least amount of time. The amount of output to be rendered should have a directly proportional effect on the time needed to complete execution.

2.3 Animations

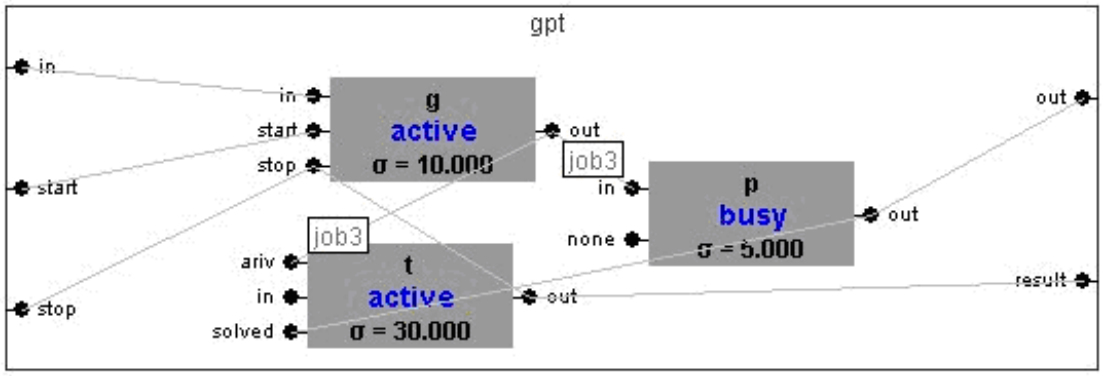


Figure 4: Animated messages in the model view for a Generator-Processor-Transducer model.

Another way to convey model interactions to the user is through animated displays. One example of this is the messages that are passed between components in DEVS-Suite. Whenever this occurs, a small rectangle slides across the screen, following

a predetermined path from the sender component to the receiver component (see Figure 4). Components may transmit multiple messages at a time in a broadcast, and it is possible for multiple components to send multiple messages at the same time. If this is the case, then DEVS-Suite will render every message animation simultaneously. The user may adjust the speed of the animation, but there is no option to toggle specific models on or off; all models will display their messages on the screen, unless the user turns off the animation at initialization.

2.4 TimeView Display in DEVS-Suite

DEVS-Suite's main data presentation interface is called the TimeView. The TimeView belongs to the View layer of the MFVC architecture, and exists to visually organize component activity. Each component selected to be viewed by the user is created in its own window, which we will refer to as a view. These views may be placed inside of a tabbed pane on the main DEVS-Suite window, or created in their own windows.

Each view contains one or more data plots called graphs. These graphs are where the data generated by the simulation will be rendered to the user. Graphs' trajectories may either be event-based or piecewise-constant, depending on the type of variable. It is important to differentiate between our concepts of views and graphs: one view represents one component, and this view may contain multiple graphs pertaining to this component. A view may not contain duplicate graphs or graphs for other components, and at most one view can be created for each component.

The number of graphs contained in a single view can vary. Each variable selected to be tracked by the user must have its own graph. At least one graph must be visible in a

view, and the user may select up to *N* graphs, where *N* is the cardinality of the predefined state variables, user-defined input ports, and user-defined output ports of the component combined.

Since the TimeView is loosely coupled with the Façade layer, it has no knowledge of the Model layer's state. The View layer's purpose is to receive data and display it accordingly, so the TimeView lacks a fundamental sense of time progression. The data contained in the TimeView is a compilation of output collected from the simulation that is stored in an array and then visually arranged on the screen. This arrangement gives the TimeView graphs the look of an oscilloscope.

Each graph contains a two-dimensional area for plotting data. The horizontal axis is subdivided into segments with a length defined by the user at initialization, called the scale, which all graphs on the same view must share. Different views may have different scales. As the view collects data, if there is new data that cannot be plotted on the screen, all graphs on that view will scroll to accommodate the new data points.

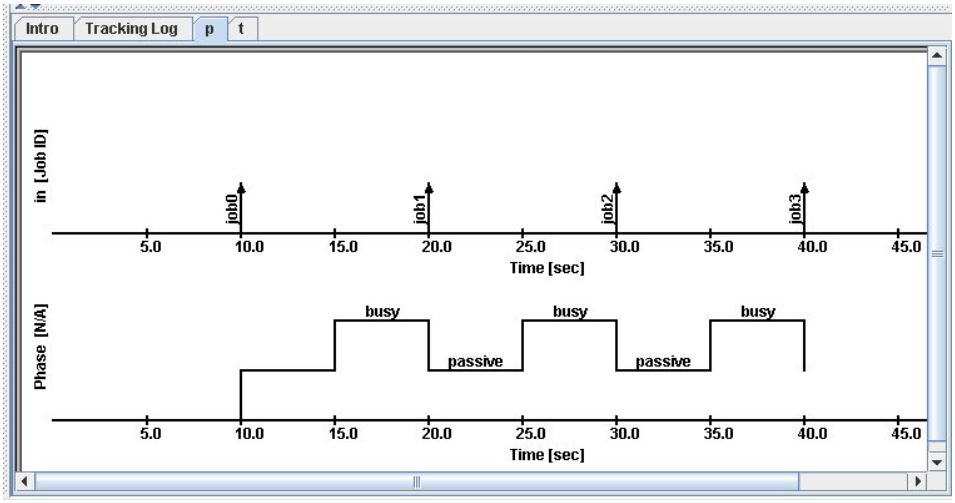


Figure 5: TimeView sample in DEVS-Suite of the Generator-Processor-Transducer model.

While the TimeView (see Figure 5) is a powerful tool for capturing the information generated by one or more components over time, it is not without its share of problems. One of the most prevalent issues with the TimeView window is that when a user views it along with the model display, the TimeView trajectory may lag behind the actual simulation execution. We will discuss how this lag affects simulation time later in this thesis. Another problem that may affect users is that a component must be tracked from start to finish; there are no options to track a new component mid-simulation or remove a component after starting to track it. Since the TimeView is only an instance of the View layer of the MFVC pattern, it is not a critical component to the functionality of the modeling and simulation engines, and can be added or removed without adversely affecting the integrity of the rest of the simulation.

2.5 Trials with DEVS-Suite

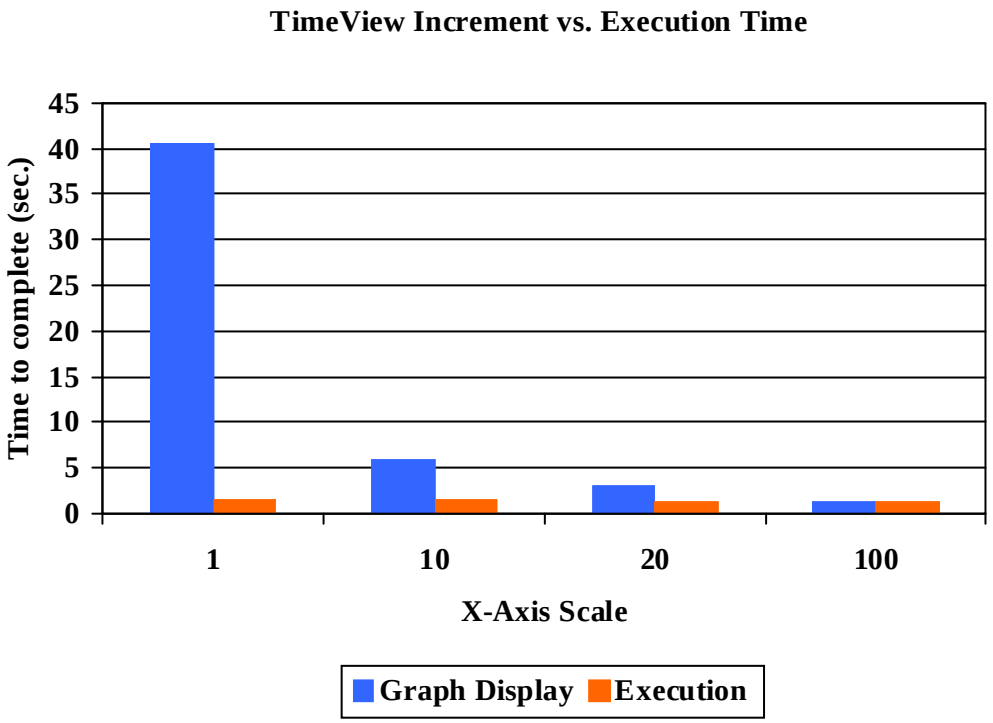


Figure 6: TimeView Increment vs. Execution Time

Now that we have studied the effect of visualization on program execution time in text-based programs, we are moving on to GUI-based programs. The experiment shown in Figure 6 was performed running DEVS-Suite version 2.0, using a manual stopwatch to estimate execution time. This experiment was also run on the same machine running Windows XP Home SP2, with 3 GHz CPU, 512MB RAM, and JVM 1.5. This demonstrates how the X-axis increment in the TimeView window affects how long it takes to fully display all of the information. This TimeView window can be seen in the lower right corner. The X-axis increment determines the difference in values between two consecutive ticks on the graph. As this number decreases, the longer the graph will appear horizontally.

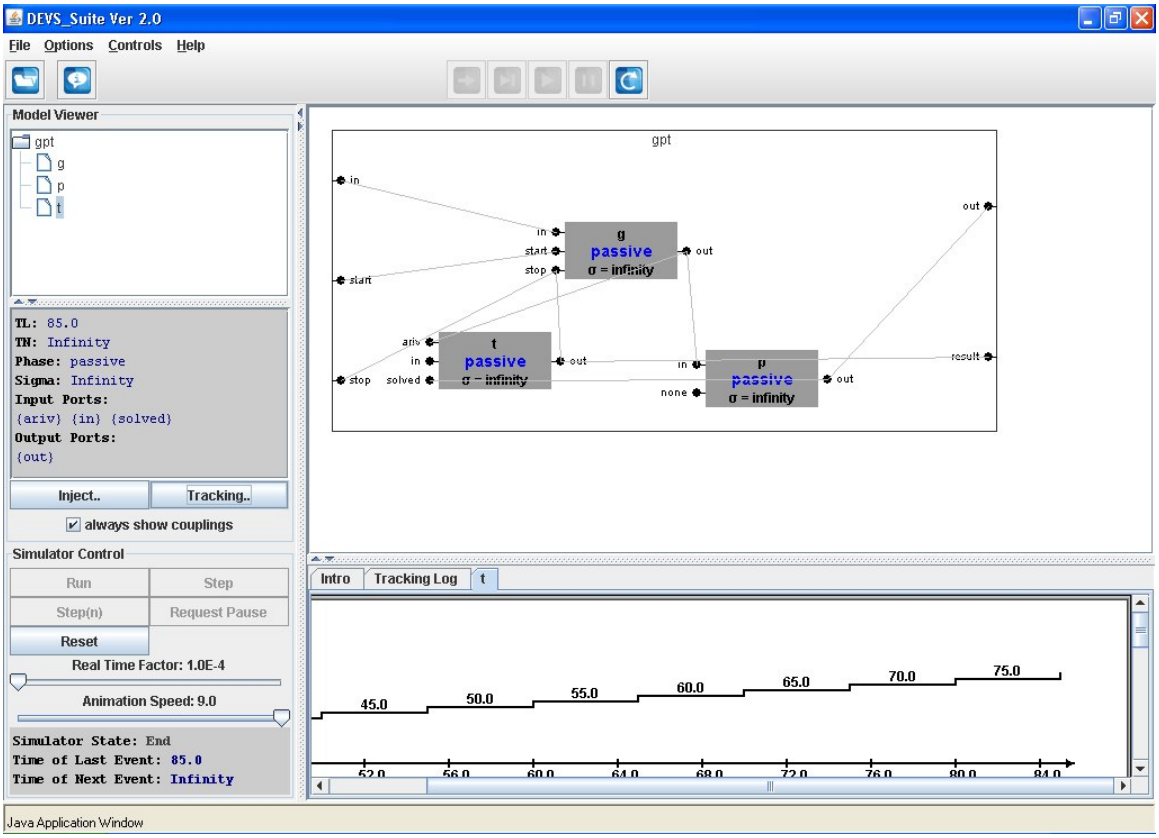


Figure 7: DEVS-Suite UI

Figure 7 shows a screenshot of DEVS-Suite in action. The top-right window contains a visualization of the models used in the simulation. The lines connecting the rectangles are the couplings along which messages travel. These messages are small, yellow boxes containing some amount of text. They travel along the paths at a certain speed, as regulated by the “Animation Speed” slider in the lower-left corner of the screen. In the lower-right corner, the program displays a tracking log and a time-view graph of various states of the simulation. As the simulation progresses, this graph updates with information taken from the model view directly above it. TimeView can display primitive data types such as double and string. The time increment and unit for time and any other

variable can be set by users. Users can monitor, track, and view status of each model component separately as time trajectories [12].

The problem with the TimeView in DEVS-Suite is that it can lag behind the simulation view, depending on the speed of the simulation and amount of information displayed. Adding variables to track does not hinder the execution time at all, but each additional item to graph increases the latency of the tracking window.

Figure 6 demonstrates this: when the increment on the x-axis is small, the time-view graph takes longer to complete. This is due to two factors: horizontal scrolling, and a maximum horizontal scroll speed. When the x-axis increment is very large, the results of the simulation can be fit on a single screen, which does not require horizontal scrolling. If the increment is smaller, the graph will need to automatically scroll to the right as it progresses through the simulation time. This may be a problem if the axis increment is very small, because DEVS-Suite limits how quickly the graph can scroll, as discussed in section 4.2. This restriction imposed by the program is also constrained by the physical capabilities of the system, and demonstrates that due to the nature of data visualization, it is not possible in certain cases to display the information from the simulation in real time.

2.6 Ptolemy

The Ptolemy project is an ongoing endeavor by the Electrical Engineering & Computer Sciences department at University of California Berkeley to study the modeling, simulation, and design of concurrent, real-time, embedded systems. It also provides

visualization support for viewing simulation results through a graphical user interface called Vergil.

Ptolemy is described as having an actor-oriented framework. Objects within Ptolemy are called “Actors” and implement an interface called Executable. The Executable interface provides the abstract semantics of control for the actor [13]. These actors that are used to build the models are separated from the UI packages. The graphical support exists in separate packages with names such as gui, vergil, and plot. There is also support for MoML, an XML-based language that is used to describe Ptolemy models.

Actor-orientation differs from object-orientation in a number of ways. In object-oriented programming, classes provide methods that are invoked sequentially; that is, they receive the control from a calling method, and eventually return that control [14]. In actor-oriented programming, an actor has data (known as its state), but communicates with other actors through its ports.

These ports provide an interface for concurrency and asynchronous communication. Object-oriented programming handles concurrency by using semaphores, monitors, and other low-level communication protocols [14]. Because actors do not have to wait for control to return from another actor, they can continue processing something else while waiting for a response. This built-in concurrency lends itself, and the actor-oriented paradigm, to model-based design.

In addition to these actors, Ptolemy also has a set of components called directors. These directors define component interaction semantics [15]. Many well known interaction types have been implemented in Ptolemy, including Continuous Time and

Discrete Event [16]. The director's job is to manage the order in which the actors execute. Models are not restricted to have only one director. In Ptolemy's strict model hierarchy, it is possible to place directors within composite actors [16]. These directors will only control the execution of the actors within their composite actor. While it is true that actors' communication is not synchronized, directors must be able to control the actors' behaviors to keep the results consistent and execution logically correct.

MoML is an XML modeling markup language intended for specifying interconnections of parameterized, hierarchical components [17]. While this language is used to specify Ptolemy models, it is designed to be programming language-independent. If a simulation tool were developed in C++, for example, it would need to be able to parse MoML files and load the appropriate classes. The MoML document is designed to be a self-contained description of the model, including all components, properties, connections, and hierarchies, but not behaviors. Behaviors are defined in class files written in a specific language, which the MoML file references.

Vergil allows the user to design and execute models in Ptolemy. Vergil is based on a diagram-editing called Diva, which is built on top of existing Java Swing technologies. Diva is a collection of loosely coupled components that employ the principles of MVC to separate data and the presentation of that data [18]. Components in Diva include the infospaces, which act as the model portion of MVC, and the surfaces, which are the view portion of MVC. These surfaces are sometimes simply wrappers for existing visual Java Swing components, but in most cases, there is extra infrastructure added to the components to help them work together in Diva.

There are also classes called “sinks” that exist to visualize data effectively. These components are added to models, but only exist to collect data produced by the simulation. Just like regular components, multiple sinks may exist for a single model. There are three types of data sinks available in Ptolemy II: Generic Sinks, Timed Sinks, and Sequence Sinks. These three groups contain a total of 17 different actor types. Example plotter actors include the XYPlotter, TimedPlotter, ArrayPlotter, which belong to the Generic, Timed, and Sequence Sink categories, respectively.

However, it does not appear that the views are required by the simulator to stay synchronized. The models seem to generate the data to be displayed and push it towards the views. The views then display the information as quickly as possible. Consider an example where a model generates one piece of information for each logical step in the simulation, and sends that information to one of the two graphs associated with the model. If the model were to send one piece of information to the first graph, then a million pieces to the second, then one more piece of information to the first again, the lack of synchronization in the views means that the first graph will plot its two pieces of information before the second graph finishes with its million, despite the fact that some of the data in the first graph was generated after some of the data in the second graph.

Simulations in Ptolemy can be run at one of two available speeds. The first speed, which is the default, is logical time. At this speed, the simulation will run as quickly as possible, subject to the physical limitations of the hardware. The other speed, real-time, is accessible through the View/Run Window menu. In real-time execution speed, the simulation time will stay in sync with wall-clock time. At this point, there are no other speeds available.

Additionally, there is no way to enforce the speed of the views attached to a model in Ptolemy. The execution speed chosen by the user (logical time or real time) only affects the speed of the model, which generates the data; the view itself that displays the information it receives will always run as fast as possible. This is apparent when the user clicks “Pause” and “Resume” in quick succession. When paused, the view will not update with new information. However, once resumed, the view will try to catch up to displaying all of the data it received from the model during the pause, since clicking pause did not stop the simulation itself from continuing.

Another modeling and simulation tool that also supports data visualization is DEVS-Suite. DEVS-Suite uses the same visualization engine as DEVSJAVA, but adds new features, such as the tracking environment and TimeView.

A comparison of two simulation software programs		
Modeling Features		
Feature	**Ptolemy**	**DEVS-Suite**
Interaction Semantics	Provided by Directors, many kinds available, including Discrete Time/Event, and Continuous.	Continuous Time is only option available for component phase/sigma. Discrete Event is only option available for ports.
View Features		
Feature	**Ptolemy**	**DEVS-Suite**
Tracking Options	User must manually add data sinks to model in order to track information.	Tracking options are built into all components. User selects components to watch using a menu.
Data Viewing Options	Sinks' graphs appear in separate windows from the model. Each graph window corresponds to a single data sink.	Graphs appear in same window, but in a tabbed pane. Each tab can display an unlimited number of graphs but from no more than one component.
Scalability	Graph view can be dynamically resized in both dimensions, even mid-execution.	X-axis scale must be set before starting execution, and cannot be changed mid-execution. Y-axis automatically rescales to fit data.
Scrolling Capability	Graphs do not scroll to show incoming data.	Graphs automatically scroll to show incoming data. They mimic the behavior of an oscilloscope.
Control Features		
Feature	**Ptolemy**	**DEVS-Suite**
Pause Functionality	Pressing pause suspends the data from being presented to the user. Pausing does not suspend model execution.	Pressing pause suspends model execution, but the graph may continue to update itself with backlogged data.
Execution Speed	Only logical-time and real-time available.	Logical-time and real-time available, as well as intermediate steps, and sub-real-time speeds.

Figure 8: A comparison of features offered by Ptolemy and DEVS-Suite

TimeView displays historical graphs for users to track various features of a component in the model. This is similar to Vergil's plotter classes, except each component selected to be viewed appears in its own tab, and only one tab can be visible

at a time. Each tab can contain multiple plots, which may include the phase or sigma of a component, if the user selects them.

There are several differences in DEVS-Suite's TimeView and Ptolemy's plotters, as described in Figure 8. In Ptolemy, clicking pause only halts the displays, but does not stop the progress of the simulation. In DEVS-Suite, clicking "Request Pause" will actually stop the simulation itself, until the user decides to continue.

Another major difference between the graphical displays is that Ptolemy's plotters allow the user to dynamically rescale the plot dimensions, whereas DEVS-Suite's TimeView graphs scroll automatically to keep up with incoming data. In Ptolemy, the plotters cannot scroll if a plot point exists outside of the window range. TimeView's scrolling window is often too slow to keep up with the speed of incoming messages, especially if the axis increment set by the user is too low.

The TimeView's graph scrolling is a process that starts automatically when information appears off the edge of the visible range. The speed of the scrolling is constant and cannot be changed by the user. The scrolling behavior cannot be controlled by the user, either. Viewing a different component's tab in the TimeView pane does not affect the speed of the scrolling; the component's graphs continue at the same rate regardless of visibility.

While the TimeViews receive the graph data as quickly as the model can generate it, the TimeView's automatic scrolling feature is what causes the latency between the data generation and data visualization. Sometimes it is possible for the graph to continue scrolling long after the simulation has completed.

2.7 Testing the DEVS-Suite TimeView

Figure 9 shows a list of factors that influence the execution speed of DEVS-Suite, and which can be controlled by the user. In the case of hardware limitations and other software running on the machine concurrently, these are out of the user's control.

The user does have control over visualization, the TimeView scale, which variables to monitor, and the speed of the execution. In the case of visualization, the user can select to view or not view the animation and tracking windows. If viewed, the user can select to track any or all of the variables and ports associated with any or all of the components in the model. The user can also define the scale of the TimeView window for each component. There is also a slider the user can adjust to modify the execution speed. These are the extent to which users can control the execution speed of DEVS-Suite.

Factor	**Controllable?**
Hardware	No
Behavior of other software	No
DEVS-Suite Visualization	Yes (On/Off)
TimeView axis scale	Yes (Any positive integer)
Variables to monitor with TimeView	Yes (any or all variables/ports provided by component)
Simulation speed	Yes (via Real Time Factor)

Figure 9: Factors on a computer system that may influence the execution speed of DEVS-Suite

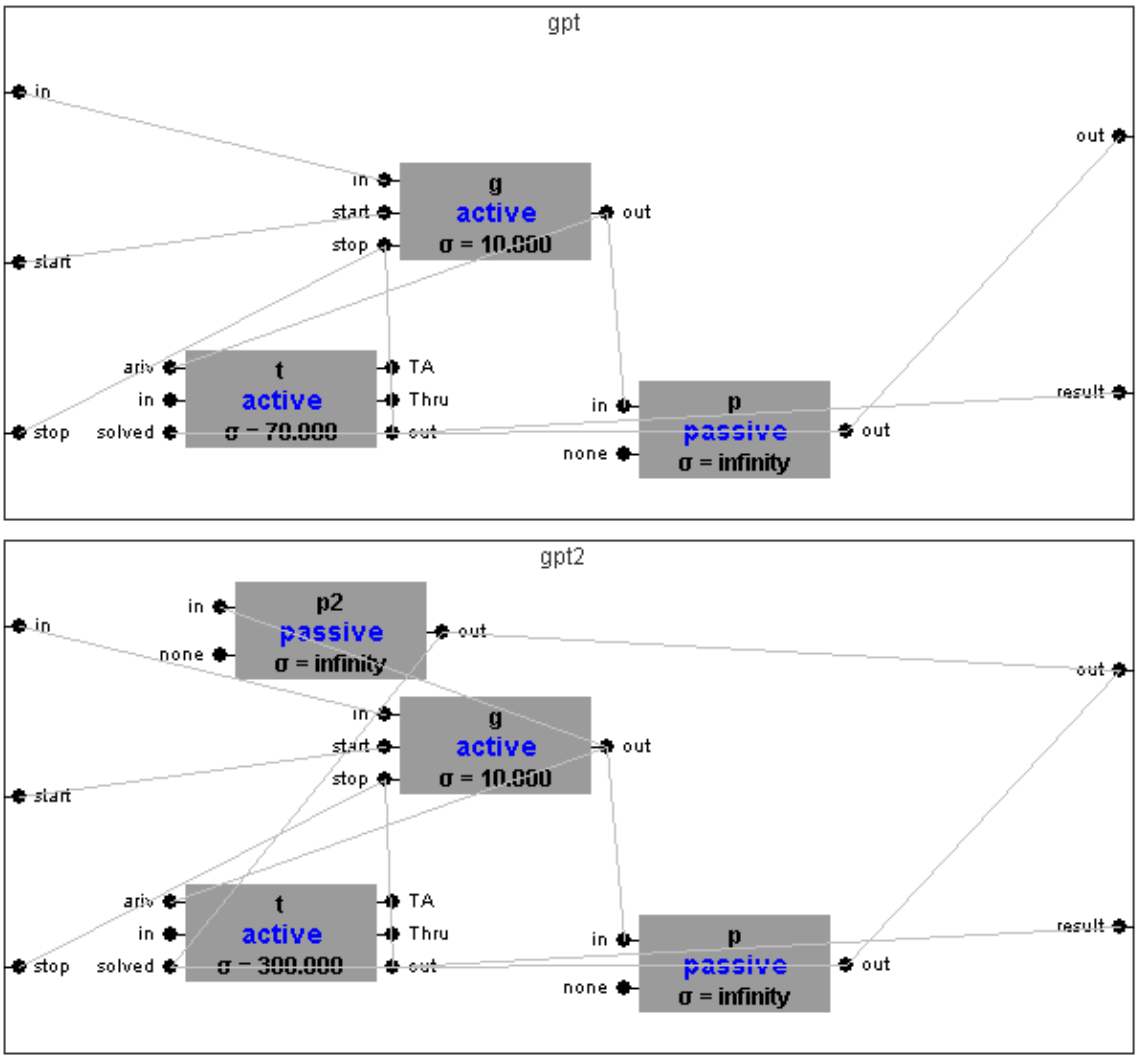


Figure 10: Models “gpt” and “gpt2” in the SimView

To test the behavior of the TimeView plots in DEVS-Suite, we constructed a test model specifically designed to reveal any flaws in the views. This model, called “gpt2” (see Figure 10), was derived from the “gpt” (Generator, Processor, Transducer) model. The difference between the two is that gpt2 has a second processor that only processes the first job created by the generator, and takes ten times longer than the original processor to finish the job. By the time the second processor finishes its job, the first one

will have already produced several output messages, possibly causing the TimeView windows for the two processors to fall out of sync with the animation window or each other.

For this experiment, the gpt2 model was executed in logical time without pausing between steps. The simulator only monitored the activity on the "out" output ports for both processors. The only change made between trials was altering the scale on the graph's x-axis.

Experiment	**gpt's scale**	**gpt2's scale**	**Result**
1	10	10	Both views scrolled in sync. P2's message was out of the range of the graph at first, but when it came into range, the message was plotted.
2	10	40	P2's message appeared (at t=105) after only two messages on P1 were plotted (t=15, t=25). P1's messages then continued to display normally, in sequence. (See Figure 12)
3	40	10	P2's message appeared after P1 finished graphing its entire results (through time=200). P2's graph then kept automatically scrolling to time=200, making P2's message disappear. (See Figure 13)
4	1	1000	P2's message did not appear on the graph at all.

Figure 11: TimeView behaviors given variable component scales

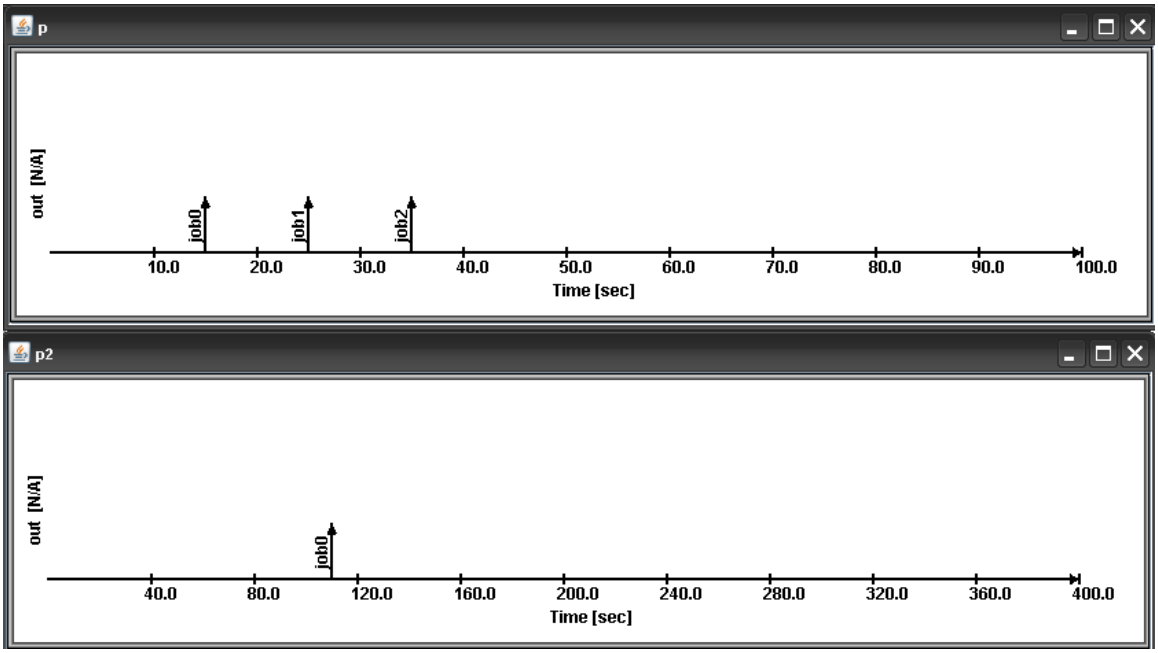


Figure 12: Illustration of Experiment 2 from Figure 11.

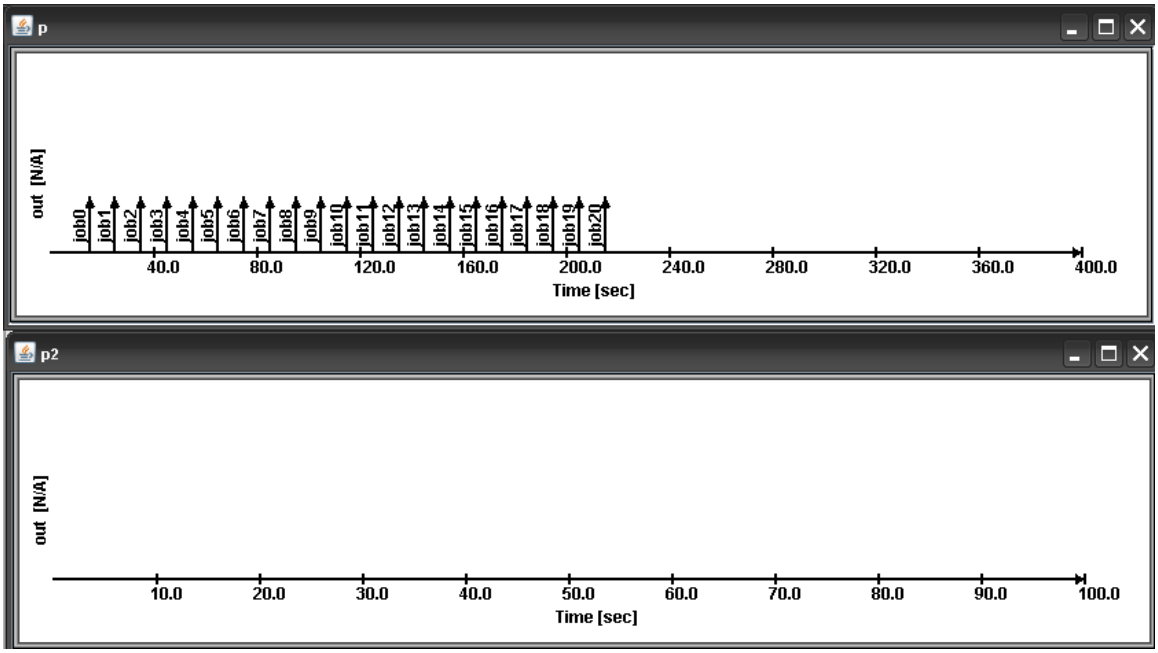


Figure 13: Illustration of Experiment 3 from Figure 11.

From these experiments, we can draw two conclusions about the behavior of the DEVS-Suite TimeView graphing functionality. Regarding its synchronization with the

animated display window, there does not appear to be any mechanism or logic in place to enforce the TimeView and model animation to stay in sync. Both tasks run as quickly as possible and ignore each other's progress. They only communicate through messages from the model to the view about what to graph. This is what leads to the TimeView lagging so far behind the animated display window in some cases.

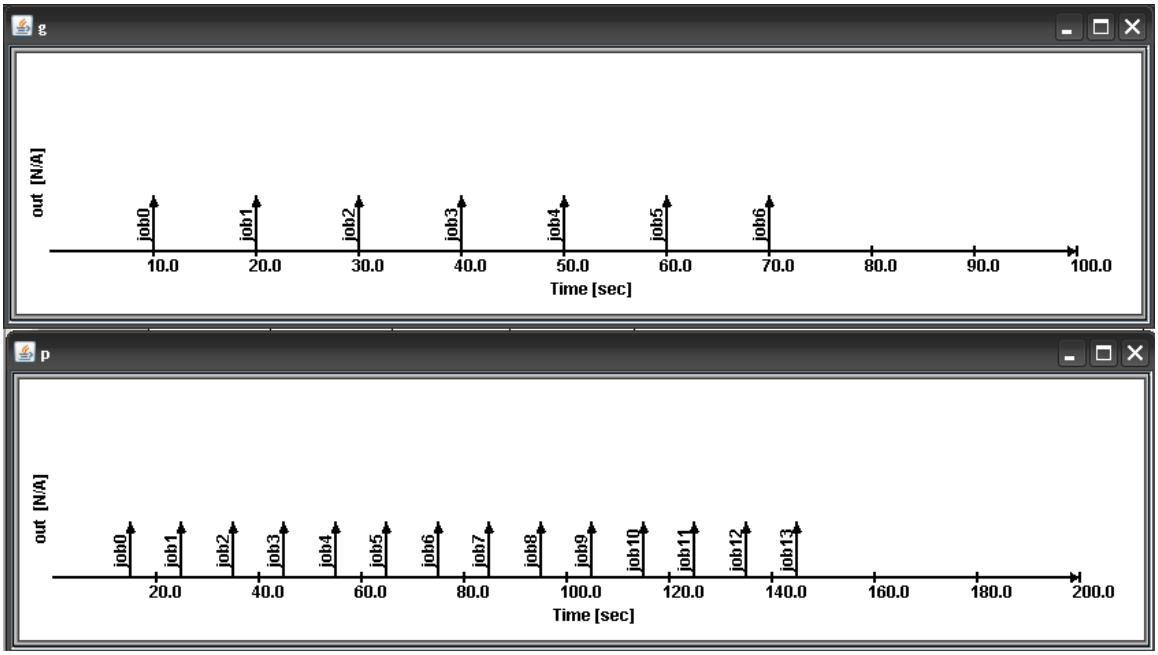


Figure 14: Example of a physical synchronization in two TimeView graphs.

As for the TimeView's ability to synchronize its own graphs, there appears to be a physical synchronization, but not a logical one. What this means is that for any two components, their plots will be visualized in sync with respect to their position on screen rather than when the plotted events occur. Suppose there are two graphs: G, which has a time-scale of 10; and P, which has a time-scale of 20, as in Figure 14. This means any given point on P's graph will represent a point in time twice as far from the logical start of the simulation than the same point on G's graph. Furthermore, suppose both graphs

transmit a message to display every 10 time-units, starting at 0 for G, and starting at 5 for P. Since the TimeView graphs are physically synchronized, these messages will be plotted in order with respect to their physical position on the screen (i.e., number of pixels). P's first message at time=15 will be graphed slightly before G's first message at time=10 even though it occurred later, because P's time=15 corresponds to G's time=7.5. Next, P's messages at time=25 and time=35 will appear, then G's message at time=20.

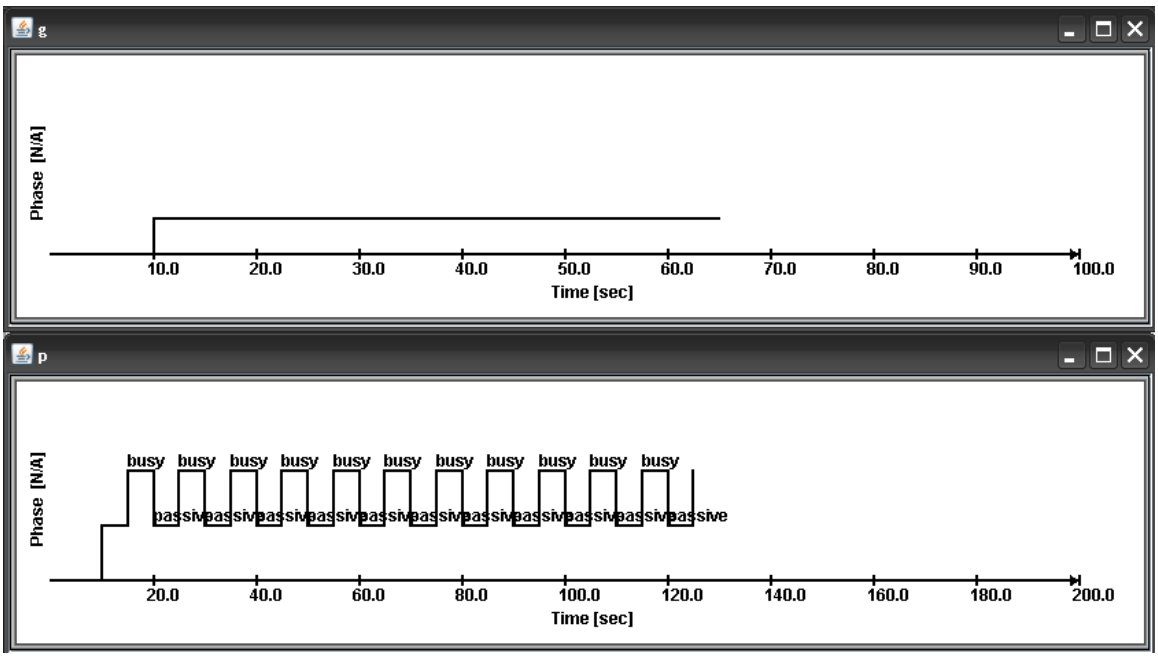


Figure 15: Physical synchronization with piecewise-constant variable phase.

Piecewise-constant variables such as phase or sigma produce similar results (see Figure 15); the graphs are updated with respect to physical screen location rather than logical time. This method of synchronization is only useful if all graphs on the screen have the same time-scale. If the scale varies between components, messages will be displayed out of order, as we have shown in Figures 12 through 15.

3 Approach

In order to add synchronization to DEVS-Suite while maintaining the simulator's validity, we had to add new components as well as modify existing code. During the process of adding synchronization, we encountered several other issues that needed to be fixed before we could ensure proper simulation behavior. This section details our modifications to the existing DEVS-Suite source code, and their implications on the software architecture.

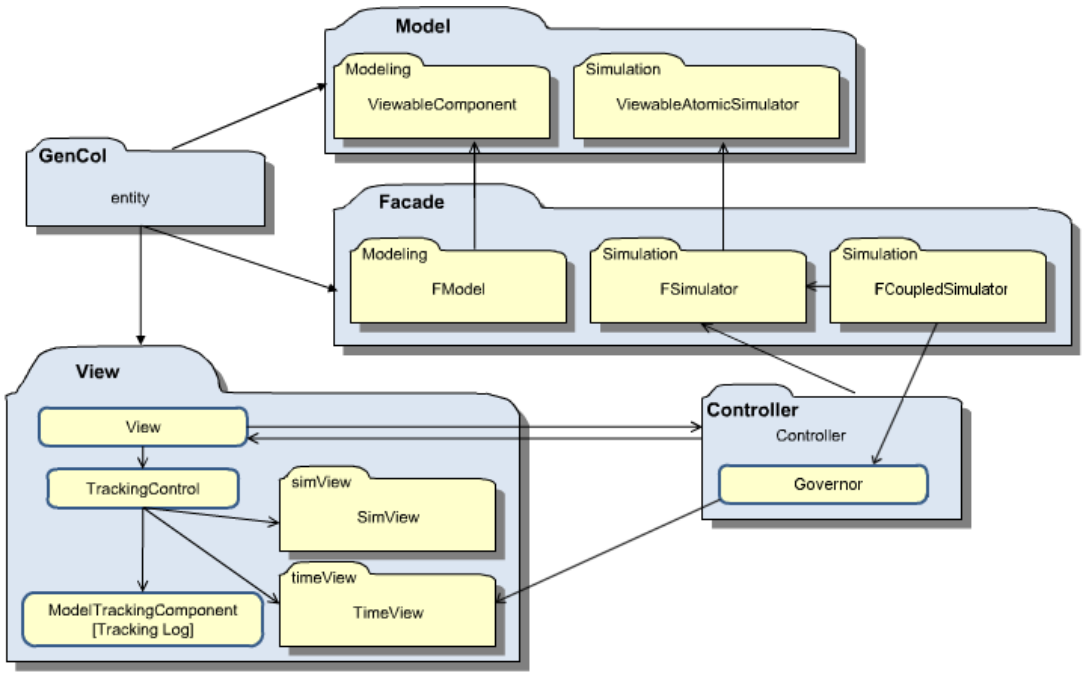


Figure 16: Package diagram of the DEVS-Suite simulation environment.

Our initial task was to determine the critical components that drove the data generation and visualization features. In the MFVC architecture, the View receives its data from the Façade, and interacts with the Façade via the Controller as illustrated in Figure 16 [5, 12]. We decided that our changes needed to be made within the Controller layer. Within the View layer, we located the objects responsible for containing the data

received from the Model, and within the Façade, we located the methods responsible for transmitting the data to the View.

After determining which components were important to the visualization, we had to define the interactions between those components to ensure proper synchronization without compromising the simulator integrity. Our approach was to suspend the simulation while the View had data that had yet to be visualized. In this section, we will show how this procedure was effective in synchronizing the simulation with the views.

3.1 The Governor Class

In order to synchronize the graphical data displays and the data models used in the simulation, we introduced a new static class called Governor. The governor interacts between the façade and views to accomplish this. Immediately before advancing the simulation time to the time of the next event, the central coordinator inside the coupled simulator calls the Governor's syncGraphs method (see Figure 18). Since the simulator is pushing data to the views, the Governor needs to ensure they are keeping up with the information. The purpose of syncGraphs is to stall the execution of the simulator until the views have finished displaying the data given to them (see Figures 19 and 20). When the simulation is complete, the Governor will remove all of its references to the views (see Figure 21).

A simulation may have any number of views associated with it, and each view contains at least one graph. Graphs are the components that display the information to the user. Each graph contains three arrays: prev, current, and next, which hold the incoming events. The current array contains the events that are within the bounds of the scrolling

graph, prev contains events that have scrolled off the side of the graph, and next contains events that have yet to appear on screen.

Events sent from the simulator to the graph are placed into the next array, and moved to the current array depending on the values of the graph display boundary variables beginTime and endingTime. These two variables respectively define the values at the extreme left and right ends of the graph. The graph is scrolled by incrementally adjusting these two values. A third variable, curTime, restricts the speed at which events move from the next array to the current array and onto the graphical display.

As discussed earlier, if the simulation gets ahead of the views, two very noticeable problems arise in the graphs. The first, and more obvious of the problems, is that a view may continue to scroll its graphs to accommodate the generated events long after the simulation has finishing executing. This is due to the limited incrementing speed of the endingTime variable. These graph boundaries increment at a fixed time interval by an amount proportional to the graph scale. The graphs already contain all of the events in their next arrays, but are waiting for the endingTime variable to increase so they can move the events into the current arrays.

The second problem appears when two views are created with different x-axis scales. If the simulation pushes too many events into the graphs' next arrays, events will appear on the screen with respect to their physical positions on the screen. This is caused by how the curTime variable increments. Since it increases by an amount proportional to the scale of the view, two views' curTime variables will move across the screen at the same rate regardless of the x-axis scale.

In synchronizing the views and model with the governor class, both of these limitations are removed. By suspending the operation of the simulator when any graph had events that needed to be graphed, events appeared on the graphs in the order the models generated them, and never fell behind the simulation execution. Below is the algorithm that allowed this synchronization to happen:

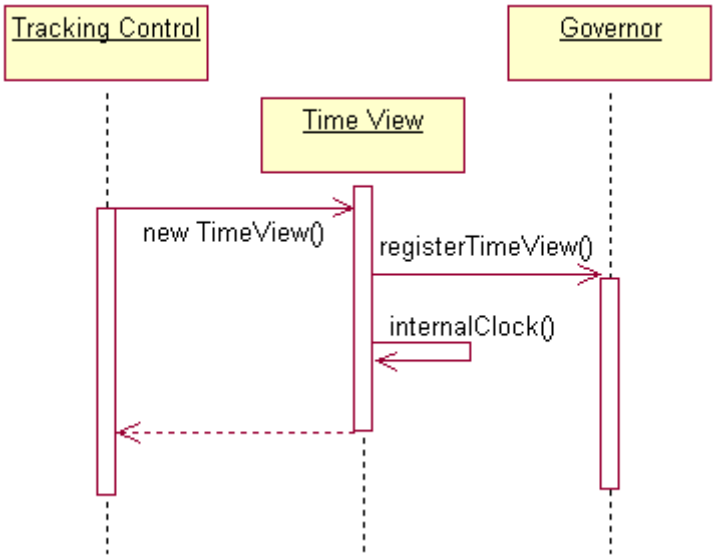


Figure 17: Sequence diagram of a time view's registration with the Governor class. When a new Time View object is created, the constructor passes a reference to itself to the static Governor object, and then invokes its own internalClock() method to initialize the Time View update timer.

For each time view created by user

Register time view with governor

Initialize internal clock that will check for new event data received

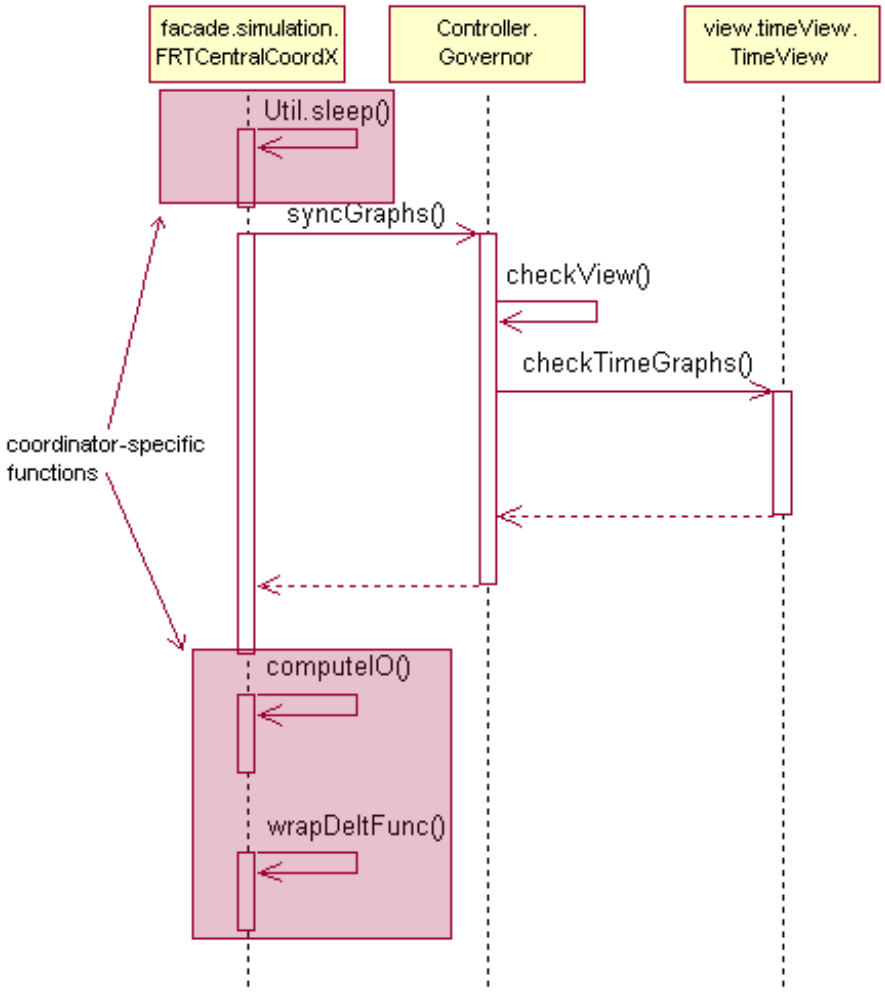


Figure 18: Sequence diagram of how Governor enforces synchronization by interacting with coordinator. After the coordinator completes its waiting period between steps, it passes execution control to the Governor class's syncGraphs() method. If the Governor is enabled, this method will then call checkView() intermittently until the method returns true. This checkView() return value is dependent on checkTimeGraphs(), which is responsible for checking each individual Time View object that is registered with the Governor (see Figure 17). Once all Time View objects have plotted all of their event

data, the Governor returns execution control to the coordinator, which continues processing the simulation normally.

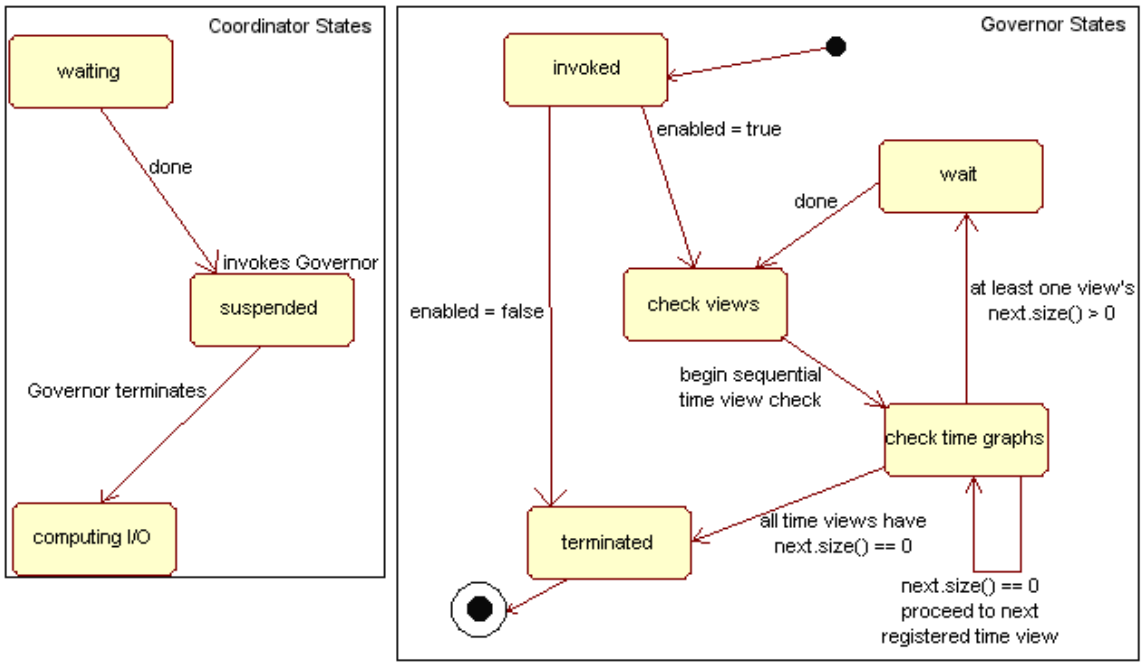


Figure 19: A state diagram of the system as the coordinator calls the Governor.

When the Time Views have data yet to plot, the Governor will cycle through the “check views”, “check time graphs”, and “wait” states until the Time Views catch up, effectively synchronizing the model and view components of the simulation.

```
In central coordinator:

After pausing graph, call Governor.syncGraphs()

Governor.syncGraphs():

While checkView() is false

        Sleep for a certain amount of time

Governor.checkView():

For each registered view, if the view's checkTimeGraphs() is false,

        Return false

Return true

CheckTimeGraphs():

For each graph within the view,

        If next.size() > 0

                Return false

Return true
```

Figure 20: A generic algorithm for describing synchronization enforcement.

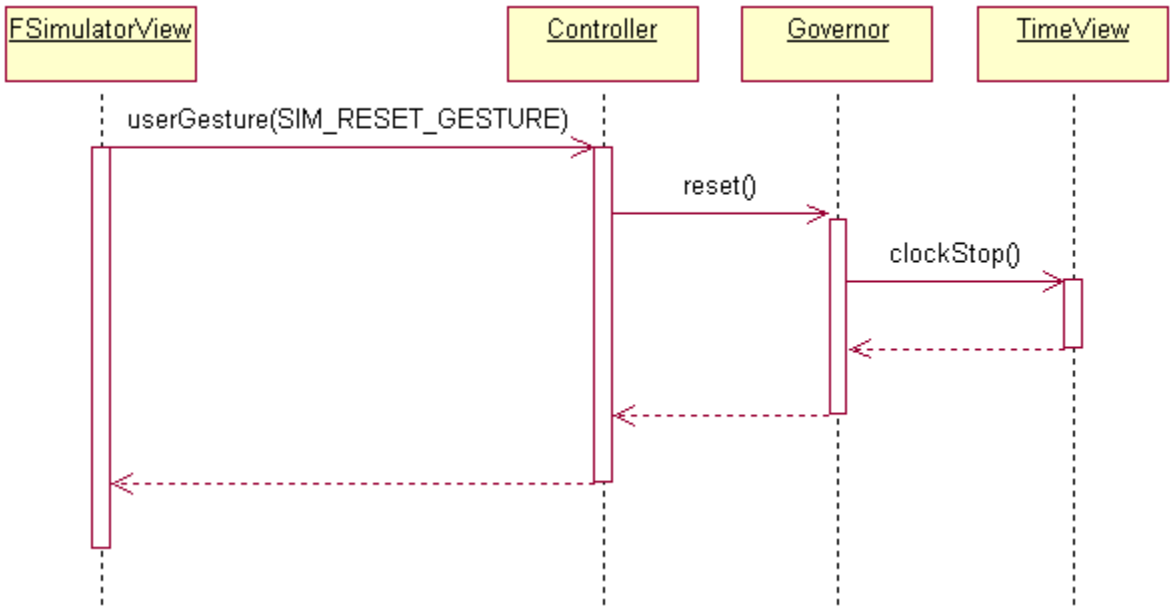


Figure 21: Governor unregisters TimeView objects when the user resets the simulator. The GUI sends a gesture to the Controller, which interprets the gesture and invokes Governor.reset(). This method is responsible for clearing its list of registered Time View objects, and setting the Governor to the disabled state.

3.2 Interface Alterations

To more effectively work with the DEVS-Suite user interface, we made a few minor adjustments to help facilitate this research. The three modifications made include adding a slider to control the speed of the time views, adding a checkbox to control invoking the Governor, and making the time views appear in separate windows rather than in a tabbed pane.

The first two changes were done to the interface to give the user more control over simulation execution. A slider added to the control panel lets the user decide how fast the views associated with the simulation should refresh. The slider is labeled “Time

View Update Speed" and has values ranging from 1 to 1000, which represent the how many times per second each view refreshes. When this value is higher, it will reduce the time needed to completely plot all of the points given to a time view. Since each graph scrolls by a fixed amount each time it is refreshed, when the time between refreshes is decreased, it may decrease the time spent waiting for the graph to finish plotting the points. When the update speed is decreased, it allows the user to view the contents for a longer period of time before the data scrolls off the left side of the graph.

A checkbox was added near this slider to allow the user to enable or disable the Governor as needed. It is unchecked by default, but the user may decide to enable the Governor at any time, even during execution. Changing the state of the Governor mid-execution affects the simulation and views upon the next simulation step. When disabled, the simulation will continue without checking to see if it is still synchronized with the time view, which may cause the simulation progress far ahead of the views, depending on how quickly information is generated. When enabled, it may have no effect if there are no views linked to the simulation, but if there is at least one component that has a backlog of events or data to plot, the simulation will immediately halt and allow those views to catch up before proceeding with further simulation. This feature can be useful if a user wants to use the Governor for certain experiments, but not for others, since enforcing synchronization can potentially slow down simulation by a significant amount.

The third change concerns how time views appear to the user. Before, time views were added to a tabbed pane on the main window. This was space-efficient, but did not allow the user to visually track more than one component at a time. Multiple graphs within a view could be seen, but all graphs had to belong to a single component. Now,

each time view is assigned its own window. Users may now view as many time views concurrently as their screen allows. This change was necessary to ensure that all views were running in time with each other.

3.3 Complications arising from design choice

Implementing the Governor effectively synchronized the execution of the simulation and the animated displays, but introduced a new complication: under certain circumstances (see Figures 22 and 23), the simulation would fall into an infinite loop, causing the DEVS-Suite program to lock up and become unresponsive.

The cause of this was a combination of the values of the variables curTime, endingTime, and the time at which the most recent event occurred, which we will denote as nextTime. The variables curTime and endingTime were responsible for producing the scrolling effect on the graphs. They increment at values that were based on the graph's scale, and it was possible for curTime to exceed endingTime as well as the time at which the next event to be displayed occurred. There are two conditions that must be met in order for a graph to cause this error:

- No value of curTime belongs to the range [nextTime, endingTime] as it increases by the increment. More formally, there is a value of the integer *k* such that:

  increment * k < nextTime < nextTime + 1 = endingTime <

  increment * (k + 1).
- nextTime > increment.

Scale	Increment	curTime	Next.time	endingTime	Result
20	4	12	15	16	OK
20	4	24	25	26	Halted
10	2	108	110	111	OK
21	4.2	109.2	110	111	Halted
26	5.2	109.2	110	111	Halted
30	6	108	110	111	Halted
110	22	110	110	111	OK
550	110	110	110	111	OK

Figure 22: Determining the conditions under which the DEVS-Suite simulator will halt.

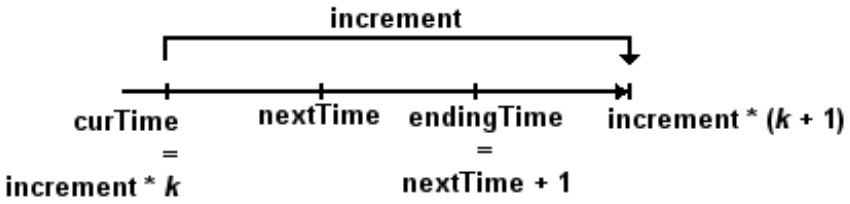


Figure 23: Graphical representation of the conditions for a simulator halt.

Before the governor was introduced to enforce synchronization, this problem existed in DEVS-Suite, but did not cause the program to become unresponsive. Without synchronization enforcement, the simulator simply pushed all of the events generated to the views and did not have to wait for the infinite loop created by the time variables. This caused the views to remain blank while the simulation continued normally.

We have determined four distinct solutions to prevent this lockup from affecting the program. One way we have already mentioned is to simply not enforce the governor. This causes the simulation and views to fall out of sync, and will not allow the data to appear on the time views, so we had to select another option.

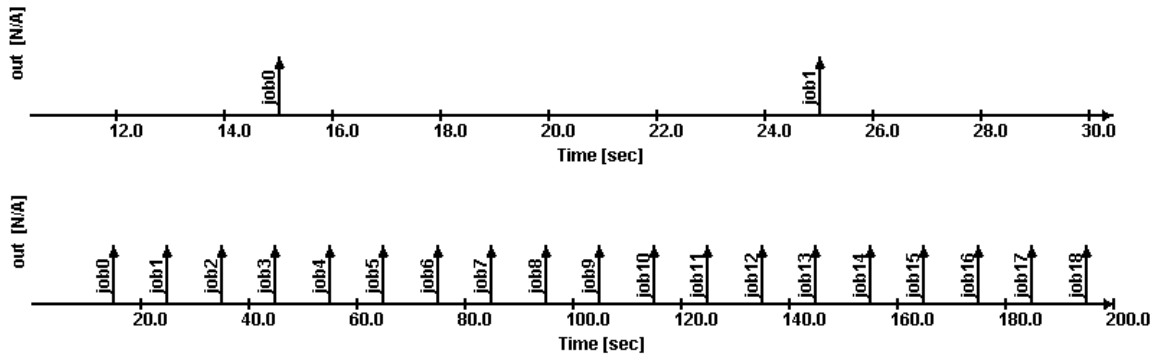


Figure 24: The same component tracked in TimeView graphs with scales 2 and 20.

The second possible way to avoid the lockup would be to make the x-axis scale very small. This option has been available to users from the very beginning, and was often used when the time view graphs did not behave as expected. The downside to this option is that if the scale is very small, the graph takes a very long time to display all of the data it receives, and cannot contain as much information on the screen as a graph with a larger range. This is exemplified in Figure 24, where we tracked the p component of the gpt2 model with two different scales. On the top graph, the scale is 2, so only two events can be plotted on the screen. The bottom graph has a scale of 20 and can hold many more events. In some cases it may not be suitable to only display the last two events on the screen, so we needed to develop another way to prevent this lockup error.

A third option is very similar, and involves tweaking the source code. The increment by which the graphs scroll is defined as one-fifth the x-axis scale. If that fraction were decreased to one-tenth or one-hundredth, it would solve most of these lockup issues. Even though the graphs would be able to display the same amount of data as before, they would still require a great deal more time to complete their graphs, and the increment defines how quickly they can scroll or update. Another important point is that

even if these two options were employed, it would not protect against a model with sufficiently small event times. If nextTime is 220 and the increment is 12, then reducing the increment to 4 will solve the problem, but if nextTime is 73, the problem occurs again. It would be ideal to have a solution that worked in all cases and did not compromise the efficiency of the time views.

Our selected solution to fix the lockup problem caused by introducing the governor was to add a condition into the function responsible for updating the graphs. In the updateTime function, there is a check to see if curTime is less than endingTime. If so, the function updates the graphs and, if needed, scrolls the graph region by incrementing the beginTime and endingTime variables. A second condition was added to check if the next array contained any items, and if either condition was true, the graph was updated as needed. This condition acted as a fail-safe in the few cases when large increments to curTime caused it to exceed endingTime even though there were still more events to plot.

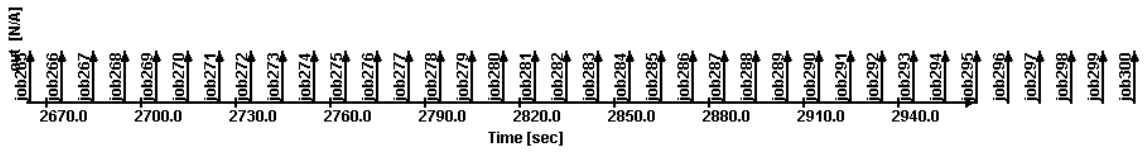


Figure 25: An incomplete TimeView graph.

Another issue related to the variables curTime and endingTime is the problem of incomplete graphs. Under certain conditions, some graphs may plot points outside of the boundaries defined by the x-axis (see Figure 25). Data points, such as events, and continuous plots, such as sigma or phase, simply continue past the right side of the graph. This causes views to have unnecessary horizontal scroll bars, and makes the graph plot data without providing the x-axis for reference.

Incomplete graphs are caused when curTime and endingTime are incompatible. When curTime exceeds endingTime, it makes an if-statement fail, causing the graph to quit scrolling temporarily. Events and data to plot are still coming in from the façade, so the graph falls behind. If this happens only a few times, it usually is not noticeable, but when it occurs many times, these short pauses add up and the discrepancy between the end of the plotted points and the end of the graph becomes very obvious. Incomplete graphs are more common when multiple views are employed simultaneously.

The solution for this problem involved removing one of the conditions of the failing if-statement. Two conditions must be true in order for a graph to scroll: curTime must be strictly less than endingTime, and curTime must be greater than timeEnd, the variable representing the value of the end of the axis. When curTime increased past the end of the graph, it was time to scroll to the right. By simply removing the condition that curTime must be less than endingTime, we were able to prevent the graph from plotting too far to the right.

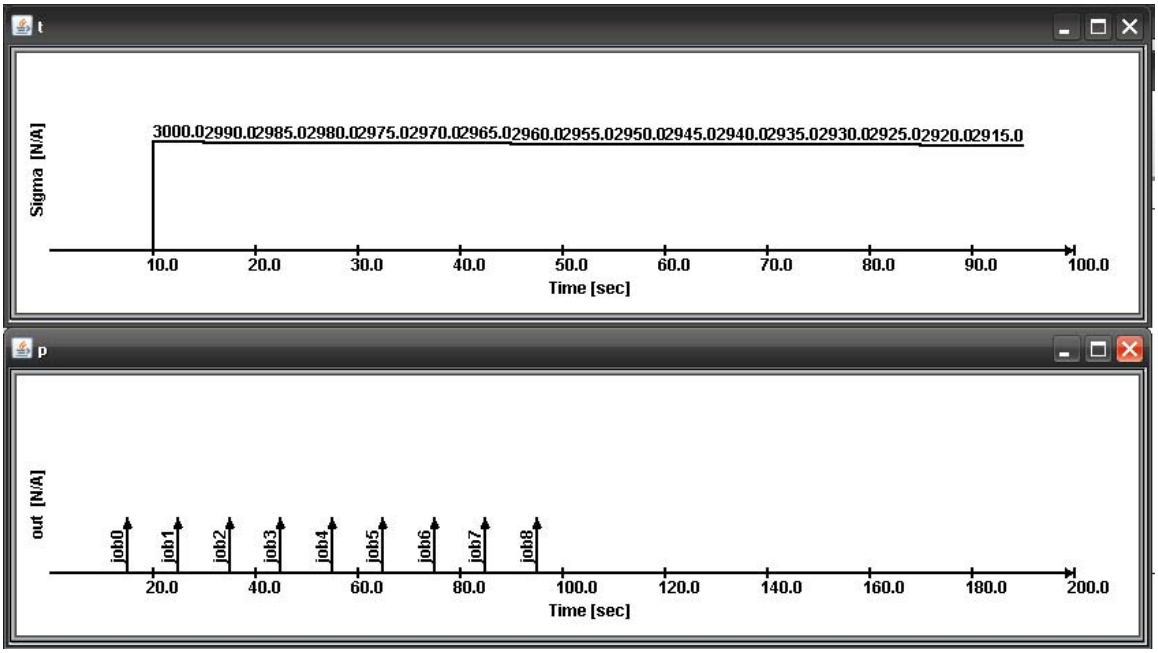


Figure 26: Synchronized TimeView windows with different scales.

With the governor in place, all time views will now stay in sync with the simulator. This can be demonstrated by making all views appear in separate windows instead of a single tabbed pane (see Figure 26). Regardless of the number of views, the governor will ensure that all views are up-to-date before letting the simulation proceed.

However, this synchronization comes at a cost: the time it takes to complete a simulation is almost always limited by the speed of the views. More precisely, the weakest link in the synchronization process is often the graph scrolling. Before adding in the governor, the feature that caused the graphs to take so long to complete was the rate at which the graph scrolled to incorporate events that occurred outside the visible bounds of the graph.

Fixing the lockup issue had an unexpected side effect: graphs now display all data they receive, regardless of the scale of the x-axis. Under normal circumstances, the graph

will plot a short, vertical line for incoming events, or a piecewise constant line graph for state variables such as phase and sigma. The graph will also plot a label next to events and on top of horizontal lines for state variables. As the scale increases, the graph becomes more compressed towards the y-axis, and the labels overlap each other. At a certain point, the graph becomes compressed into a single vertical line on the far left side of the axis, and labels cease to appear.

It is possible to determine the point at which the labels will cease to appear. In the code that determines the visual properties of the view, there are several variables of importance within the TimeGraph class: graphXstart and graphXend define the starting and ending positions of the graph within the view display, labelIncrement is the difference in values between each label on the x-axis, and numLabels defines how many segments are to appear on the graph. Normally, graphXstart and graphXend are set to 30 and 780 respectively, to define a plot area 750 pixels wide. The variable labelIncrement is 10 by default, but can be changed by the user input when a view is instantiated. Finally, numLabels is fixed at 10; the user has no control over how many graph segments appear on the time view.

The key variable in the class that determined whether the labels would appear is xIncrement, which is computed as

(graphXend - graphXstart)/(labelIncrement * numLabels).

This variable represents the number of resolution of the graph in pixels per x-axis unit. A very small number means the graph is very compressed and able to display a lot of data on the screen at once, whereas a larger number means less data will be on the screen at the same time, but will be easier to see.

Under the current version of DEVS-Suite, if xIncrement falls below 0.1, the graph will behave differently. The plotted lines will be compressed to a single line, and no labels will be applied to the data. Assuming the user cannot alter the variables in the program other the graph scale, this means the scale must not exceed 750 units, which is coincidentally the width in pixels of the plot within the graph.

3.4 Simulation cleanup

After a simulation has completed and all data has been plotted to the time views, the user must either reset the simulation or choose another model to load. When a user performs either of these actions, there will be no visible remains of the previous run; it will be as if the user had just loaded the program.

However, if the user opted to display any time views during the previous run, the threads that prompted the graphs to update constantly would persist. Java's garbage collector would not stop these Timer objects from running, so after each simulation, these threads continued to run in the background, consuming CPU resources. Wasting CPU time on these rogue Timer objects caused subsequent simulations to slow down considerably without contributing anything useful to them. Finding a way to solve this problem would provide an efficient and more consistent environment for subsequent simulations in DEVS-Suite.

The Governor was used to terminate these threads once the simulation was over. When the user clicks the Reset button on the interface, a gesture is sent to the controller, which then interprets the gesture and performs certain related functions. We added in an extra function call to the controller's reset routine that would prompt the Governor to halt

all Timer threads related to time views. Since the Governor contains an array with references to every time view, it was able to shut the Timer objects down safely before clearing the array.

3.5 Implementing Synchronization in other Simulation Tools

Software developers can add a Governor component to other simulation tools as long as those tools use the MFVC architecture. Adding synchronization is still possible with the MVC architecture, but our implementation specifically utilizes functionality of the Façade.

First, locate the simulator protocol responsible for executing the model and generating events, and determine which classes are involved in the simulation process. This protocol should be located in the Model and accessible via the Façade. Simulation will be suspended by Governor to enforce synchronization as described in Section 3.1. The Governor may need to be adapted to the protocol of the simulator.

Second, determine which components are responsible for displaying simulator output. These are not limited to graphical displays; anything that will receive data from the Model is applicable here. The Governor must have references to the visualization objects in order to track their progress. Each time a new display is created, it must be registered with the Governor. Also, it is critical to ensure that views are self-driven. Each individual view must have its own thread responsible for updating its displays.

Third, find how data is collected in the View to be displayed. In order to effectively synchronize the simulation with the displays, a scheme needs to be devised to

track how much data has yet to be visualized. In the case of DEVS-Suite, data is moved between arrays inside of each individual TimeView as it is displayed on the screen.

Fourth, after collecting all of this information, the Governor needs to be adapted and added to a specific simulation engine. A single instance of the Governor class is used. In Java, this can be achieved by declaring the contents of the Governor as static. During a simulation, a scheme needs to be devised to pass execution control to the Governor temporarily in order to synchronize the Model and View, and then return control in order to continue the simulation (see Figure 18).

Finally, after the Governor has been implemented, a suite of tests need to be devised and carried out to ensure that synchronization is working properly. The first test is to test a model with no visualization, with and without the Governor. If the implementation is successful, there should be no significant difference in execution times with the Governor enabled and without. Next, test a model with and without the Governor, but for each simulation enable a view for one component. There should be a significant increase in execution time for the simulation when the Governor is enabled. If there is a difference between the execution time between the simulator and display, the simulation time will increase to match the display time when the Governor is enabled, as this is evidence that synchronization is being enforced. Additionally, similar results will be obtained when multiple views are added to a model.

Each test of the synchronization should be executed multiple times, recording execution times to build a sample data set. Running each test multiple times is advisable because the execution time for a single simulation can vary significantly. Hardware

constraints, operating system processes, and other programs running simultaneously may all have an impact on the execution time of a single execution of a test.

4 Results and Benchmarks

Once the Governor was in place to synchronize the graphs with each other and the simulator, we did performance testing on the entire system to see how well it ran. To do this, we needed a way to measure the time taken by the program. Rather than rely on a manual stopwatch we developed a special class called Stopwatch that could be used to track the time elapsed from an arbitrary time point in the execution.

Below is a list of trials we ran on the DEVS-Suite system, using the efp and gpt2 models under various program settings. The time to completion is measured in the following ways: for the simulator, it is the time for the model to passivate (all components have a tN of infinity); and for the views, it is the time to plot all data points.

Trial Notation	SimView	Governor	TimeView (scale=10)	TimeView (scale=30)
O				
S	X			
SG	X	X		
ST	X		X	
SGT1	X	X	X	
SGT2	X	X	X (Multiple)	
SGT3	X	X		X

Figure 27: Definitions of notations used in result tables.

In Figure 27, we define a notation that we will be referencing in the tables that describe the results of our experiments. For each model and test condition, we employed a number of different combinations of simulator features that could potentially impact the simulation time. These features include the SimView, Governor, and TimeView. A mark

in the SimView column means we enabled the visual model depiction (see Figure 4). A mark in the Governor column means we enabled synchronization, otherwise it was disabled. A mark in the TimeView column means that we tracked and plotted one variable from one component during the experiment, with an x-axis scale of 10 for SGT1, or a scale of 30 for SGT3. For the row SGT2, we tracked multiple variables from multiple components.

Our basis for ordering these feature combinations as such is to determine which conditions create a significant difference in execution time for the simulator or the visualization. We believe that this order represents increasing levels of required processing; each added component should require more time to complete.

For each combination of model, simulation length, and feature set, we applied a consistent methodology. Each experiment was run ten times, clicking “Reset” after run. The model was not reloaded and nor program terminated between runs. After collecting the ten data samples, we removed the minimum and maximum values, and computed the mean and standard deviation from the remaining eight samples.

Model name = “efp”, simulation length = 200		
Trial	**Mean Simulation Time (sec.)**	**Standard Deviation (sec.)**
S	3.098	0.216
SG	2.903	0.0833
ST	5.382	0.144
SGT1	6.667	0.0649
SGT2	12.963	0.810

Model name = “efp”, simulation length = 1000		
Trial	**Mean Simulation Time (sec.)**	**Standard Deviation (sec.)**
S	16.650	0.757
ST	25.346	1.334
SGT1	30.539	1.390

Model name = “efp”, simulation length = 3000		
Trial	**Mean Simulation Time (sec.)**	**Standard Deviation (sec.)**
S	65.655	0.720
ST	74.241	2.067
SGT1	119.404	1.259

Model name = “gpt2”, simulation length = 200		
Trial	**Mean Simulation Time (sec.)**	**Standard Deviation (sec.)**
O	0.414	0.0637
S	0.619	0.140
SG	0.521	0.0463
ST	0.700	0.0656
SGT1	5.751	0.149
SGT3	1.914	0.0234

Model name = “gpt2”, simulation length = 1000		
Trial	**Mean Simulation Time (sec.)**	**Standard Deviation (sec.)**
O	2.064	0.0610
S	2.501	0.121
SG	2.430	0.0842
ST	3.214	0.114
SGT1	27.620	1.108
SGT3	9.953	0.381

Model name = “gpt2”, simulation length = 5000		
Trial	**Mean Simulation Time (sec.)**	**Standard Deviation (sec.)**
O	25.514	0.100
S	26.637	0.697
SG	25.658	0.132
ST	28.854	0.356
SGT1	145.328	1.493
SGT3	64.064	0.466

Figure 28: Simulation times for various models, lengths, and GUI features.

The tables in Figure 28 show the effect of various experimental conditions on the simulation time. The next tables contain identical conditions, but the values will now represent the visualization time. For these, the trials “S” and “SG” have been removed, as they have no measurable visual component. Figures 29 and 30 show how visualization time changes under different experimental conditions.

Model name = “efp”, simulation length = 200		
Trial	**Mean Visualization Time (sec.)**	**Standard Deviation (sec.)**
ST	6.779	0.0971
SGT1	6.717	0.0634
SGT2	12.894	0.818

Model name = “efp”, simulation length = 1000		
Trial	**Mean Visualization Time (sec.)**	**Standard Deviation (sec.)**
ST	31.965	0.916
SGT1	30.628	1.378

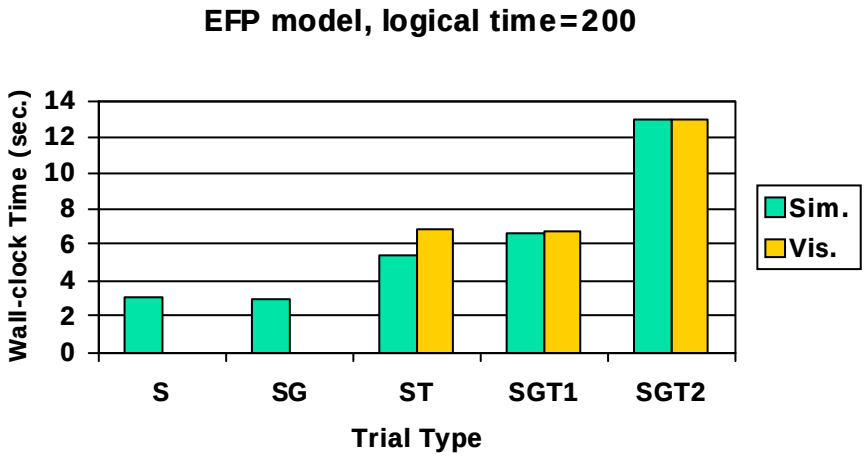
Model name = “efp”, simulation length = 3000		
Trial	**Mean Visualization Time (sec.)**	**Standard Deviation (sec.)**
ST	114.658	1.481
SGT1	119.510	1.249

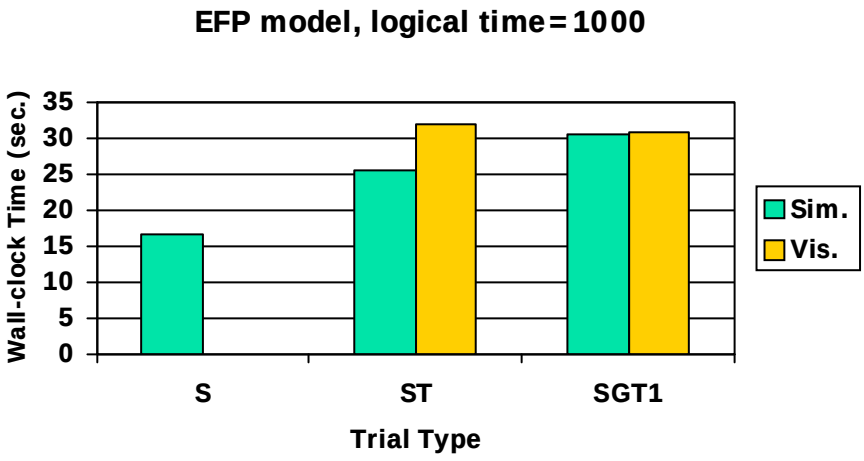
Model name = “gpt2”, simulation length = 200		
Trial	**Mean Visualization Time (sec.)**	**Standard Deviation (sec.)**
ST	5.722	0.0311
SGT1	5.971	0.158
SGT3	1.934	0.0236

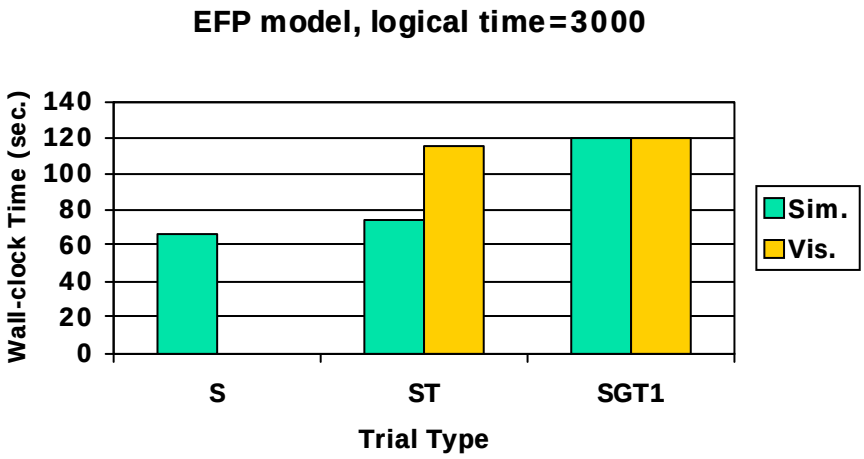
Model name = “gpt2”, simulation length = 1000		
Trial	**Mean Visualization Time (sec.)**	**Standard Deviation (sec.)**
ST	27.846	0.304
SGT1	27.833	1.092
SGT3	10.002	0.382

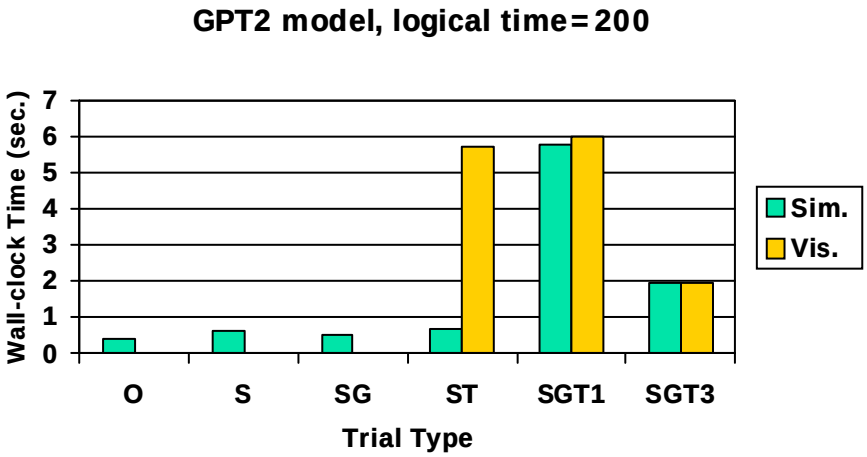
Model name = “gpt2”, simulation length = 5000		
Trial	**Mean Visualization Time (sec.)**	**Standard Deviation (sec.)**
ST	155.008	1.491
SGT1	145.577	1.495
SGT3	64.125	0.470

Figure 29: Visualization times for various models, lengths, and GUI conditions.




EFP model, logical time=1000
Wall-clock Time (sec.)
35
30
25
20
15
10
5
0
S
ST
SGT1
Trial Type
Sim.
Vis.


EFP model, logical time=3000
Wall-clock Time (sec.)
140
120
100
80
60
40
20
0
S
ST
SGT1
Trial Type
Sim.
Vis.


GPT2 model, logical time=200
Wall-clock Time (sec.)
7
6
5
4
3
2
1
0
O
S
SG
ST
SGT1
SGT3
Trial Type
Sim.
Vis.

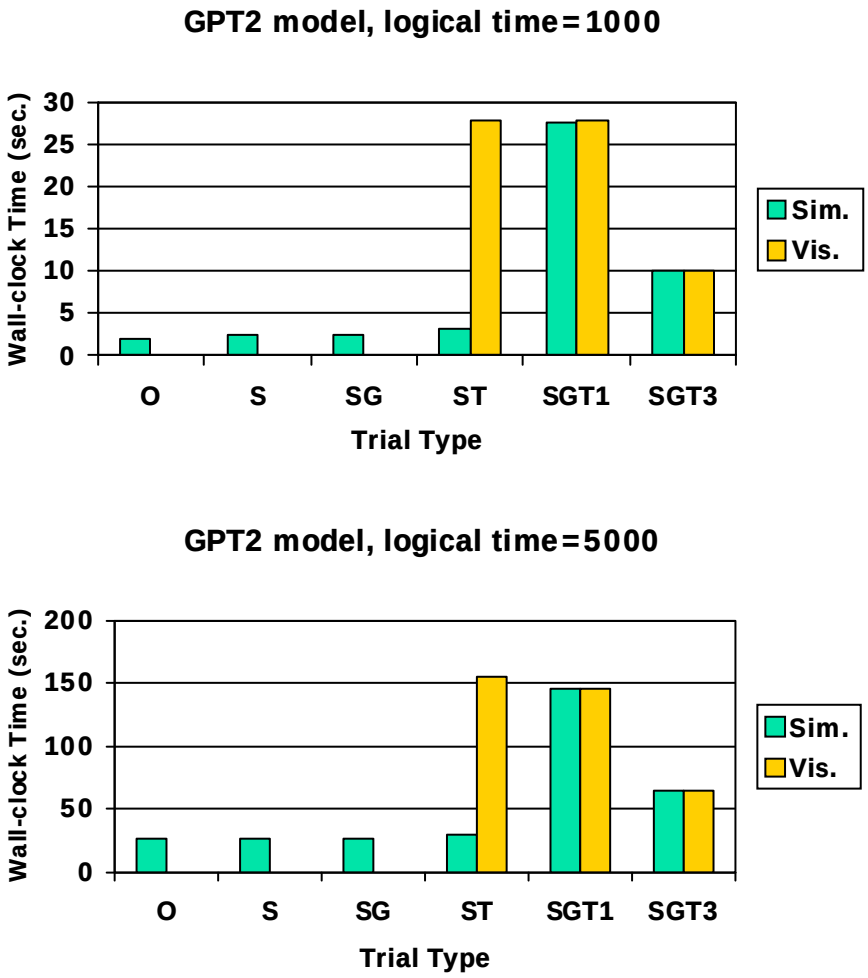


Figure 30: Comparisons of simulation and visualization times between efp and gpt2 models.

One noticeable issue with these trials is that the actual time taken to complete a particular task can vary significantly. If a simulation was conducted immediately after initializing DEVS-Suite, it ran significantly slower than subsequent simulations after clicking “reset”. After one or two runs, the simulation speed increased and remained much more consistent. We have no control over this factor as it is likely a side effect of

dynamic memory allocation in Java. We have accounted for this in our experiments by throwing out the highest and lowest run times before calculating the mean.

During these trials, the computer was allowed to function normally, operating all programs and background processes that it would during normal use, including networking connections, Eclipse, and browser windows. However, there was no user interaction while the experiments were in progress. There will always be something running in the background intermittently, such as the operating system trying to reorganize memory, or more process-intensive applications such as a development environment. What is important is how long one time view takes to finish relative to the others.

Since the simulator generates data for the time views to show the user, it is clear that the simulator should terminate before the views do. One can estimate the amount of time needed by a view to finish graphing all of its data, under certain conditions, with the following formula.

$VT = ST + E + I$, where:

VT := "View Time" = amount of time needed for view to complete.

ST := "Simulation Time" = amount of time needed for simulation to complete.

E := "Event" = ceil(delta-t_{final} / ($scale$ / 5)) * $cycle$ = "event time", or the time needed to plot the last event generated by the model.

$delta\text{-}t_{final}$:= change in simulation time between the second-to-last and last events in the model.

scale := x-axis increment as defined by the user when creating the time view.

cycle := time between refreshes of the time view displays.

I := “Idle” = time spent by CPU on other applications, drawing the graph to the screen, etc.

Furthermore, this formula only applies under the following conditions. First, in order for this to be accurate, it may only be applied when the Governor is being enforced. Second, the time view being measured must have data to visualize at the last time point of the simulation.

We can draw a number of conclusions from these experiments. Using the efp model, we have shown that the Governor does not introduce a significant change to simulation time if no TimeViews are active. Adding a TimeView to display simulation data does introduce a significant change in raw simulation time. Enabling synchronization constrains the simulation to run only as fast as the TimeView can display data, but does not introduce any change in execution time for the visualization. Additionally, the more components or variables are tracked by the user, the longer they all take to complete.

The gpt2 model experiment reinforces these conclusions. Not only does the Governor effectively synchronize data generation with data visualization without introducing processing time overhead, the Governor class also behaves correctly for any scale model. Regardless of the number of components tracked or the length of the execution, synchronization was enforced for the length of the experiment.

In testing the gpt2 model, we also adjusted the TimeView scale and noted its effects. When adding a new component to be tracked, the user may specify a value for the x-axis scale, or use the default of 10. We increased this value to 30, which allowed the graph to plot more data and scroll less. We found a significant decrease in the time needed to plot all of the data generated when the scale was increased. In section 3.3, we discussed how the TimeView graphs scroll by a constant amount based on the graph's scale. In increasing the scale, we increased this amount, thereby decreasing the amount of time needed to scroll the graph to the end of the data.

While testing the gpt2 model, we also studied the effects of disabling the SimView feature of DEVS-Suite. SimView is responsible for visualizing the model and displaying the current states of the components. When this feature was removed, the simulation time exhibited a significant decrease in processing time for short simulations. However, once the simulation length was too large, as in our case with a length of 5000, there was no significant benefit to be gained from disabling the SimView. We believe this relates to the transducer component's ability to effectively manage large quantities of collected events, but this problem is beyond the scope of this study.

5 Conclusions

Solving the problem of synchronizing the model and view components of a simulation system has led to many discoveries about software architecture. We have taken a look at the MFVC software design, which is what DEVS-Suite uses as its foundation. In adding the Governor class, we were able to establish a link between the simulator and the views. The simulator used the Governor to check if all views were up-to-date with their plotting, and if so, allowed the simulation to proceed for one step. This process enforced synchronization between the model and view of the system.

Synchronization is a useful tool for model verification and subsequent validation. For example, if a user wanted to watch a set of components to make sure they were sending and receiving messages properly, it would be easy to set up a handful of time views to track the components. Without synchronization, the data would appear on the views nearly all at once without regard to simulation time. When the user enforces synchronization by enabling the Governor, the user can see when data is generated by each component with respect to each other. If one component sends a message to another, and both of them have time views associated with them, the user should be able to see that transaction instantly on both of the time view graphs. The user doesn't have to wait for the graphs to catch up to the simulation, and there will be no confusion when events occur relative to each other, because the simulator slows down to accommodate for the time-consuming process of visualizing the data.

Synchronization is not useful in all situations. If a user has already validated their model and is concerned with the execution speed of their simulation, synchronization would not be useful to them. The trade-off of having the views and simulation stay

together is that data visualization in inherently slower than data generation. We have shown that enabling the Governor restricts the simulator to running only as fast as the program displays data to the user. Depending on the specific parameters of the views, this could have a variable, but consistently negative, impact on the performance.

Implementing the Governor also revealed a number of issues that needed to be addressed before the program could run smoothly. The issues with plotting data became important when those problems interfered with the simulation. We had to address the graphing problem when a specific condition regarding the graph variables caused the simulator to lock up in an infinite loop. The Timers that would not stop after resetting the simulator were not critical to the execution, but were a nuisance in that they caused poor performance in subsequent simulations. Once these were all addressed, the Governor performed its function perfectly.

The changes specified in this thesis have been applied to DEVS-Suite, and are available on the web. Version 2.1.0 of DEVS-Suite supports synchronization between the model, SimView, and TimeViews. The latest version of this project can be downloaded at the following URL: https://sourceforge.net/projects/devs-suitesim/

6 Future Work

Now that DEVS-Suite can effectively synchronize the Model and View layers, we plan on making this version of the simulator more accessible to others in the Modeling and Simulation community, as well as more feature-rich. Our plans for simplifying accessibility include making a stand-alone, web-based version of DEVS-Suite, and integrating this version with a visual modeling tool.

There are several auxiliary functions that DEVS-Suite currently lacks. First, TimeViews created in separate windows can only be used for one simulation. If the user resets the model or loads a different one, all TimeView windows cease to function and must be closed manually. Second, there are no capabilities to scroll backwards on a TimeView graph. We have determined that no data sent to a view is deleted, so with proper user controls, a user could manually scroll through the data collected to review the simulation results. Third, we are unable to estimate the wall-clock time required to complete a simulation. We have discovered that the actual time needed to run a single simulation varies greatly, so having a way to deterministically predict simulation time would be useful.

We will use Java WebStart [19] as a way of distributing DEVS-Suite through the web. WebStart technologies will allow users to download and launch the program automatically, without having to launch through an IDE such as Eclipse, or locate executable files. If users access the program via WebStart, and updates made to DEVS-Suite on the web will take effect immediately; users will not have to check for updates themselves.

Furthermore, we plan to integrate DEVS-Suite with CoSMoS (Component-Based System Modeling and Simulation), a visual model development environment. Users will then be able to develop their models in CoSMoS and simulate them in DEVS-Suite. Not only will this simplify the model development process, but it will encourage users to make more logically correct models. By removing users from the low-level coding, users without Java programming proficiency will be able to develop models for use in DEVS-Suite.

Bibliography

[1] *Ptolemy II home,* <http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm> (15 January 2009).

[2] *Simulink Home,* <http://www.mathworks.com/products/simulink/> (15 January 2009).

[3] Zeigler, B.P., H. Praehofer, and T.G. Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, Second Edition. Academic Press.

[4] Mather, Jeff. *The DEVSJAVA Simulation Viewer: A Modular GUI That Visualizes the Structure and Behavior of Hierarchical DEVS Models*. 2003.

[5] Kim, Sungung. *Simulator for Service-Based Software Systems: Design and Implementation with DEVS-Suite.* 2008.

[6] Sarjoughian, Hessam S., and Ranjit K. Singh. *Building Simulation Modeling Environments Using Systems Theory and Software Architecture Principles*. Advanced Simulation Technology Symposium, 2004.

[7] Singh, Ranjit, and Hessam Sarjoughian. *Software Architecture for Object-Oriented Simulation Modeling and Simulation Environments: Case Study and Approach*. 2003.

[8] Chen, Yu and Hessam Sarjoughian. *A Component-based Simulator for MIPS32 Processors.* Simulation Transactions, 2009.

[9] Sarjoughian, Hessam, Yu Chen, and Kevin Burger. *A Component-based Simulator for MIPS32 Processors.* 2009.

[10] Hansen, Stuart and Timothy V. Fossum. *Refactoring Model-View-Controller*. Consortium for Computing Sciences in Colleges, 2005.

[11] Mitchell, Aine and James F. Power. *An Approach to Quantifying the Run-time Behaviour of Java GUI Applications.* 2004.

[12] Kim, Sungung, Hessam Sarjoughian, and Vignesh Elamvazhuthi. *DEVS-Suite A Simulator For Visual Experimentation and Behavior Monitoring.* SpringSim Conference, 2009.

[13] *Ptolemy FAQ.* <http://ptolemy.berkeley.edu/ptolemyII/ptIIfaq.htm> (12 February 2009).

[14] *Ptolemy PPT*. <http://ptolemy.eecs.berkeley.edu/presentations/04/overviewptolemyOMG.ppt> (12 February 2009).

[15] *EE249 Presentation* <http://ptolemy.eecs.berkeley.edu/presentations/03/EE249presentation.ppt> (12 February 2009).

[16] Asrigo, Yanwar. *Reading Report: Modeling and Simulation of Heterogeneous System in Ptolemy.* <http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/projects/repository/Yanwar%20Asrigo/readingReport.pdf> (12 February 2009).

[17] Lee, Edward A., and Stephen Neuendorffer. *MoML — A Modeling Markup Language in XML — Version 0.4*. University of California at Berkeley, 2000.

[18] *DIVA: Dynamic Interactive Visualization.* <http://embedded.eecs.berkeley.edu/diva/about/papers/thanksgiving.html> (12 February 2009).

[19] *Java SE Desktop Technologies.* <http://java.sun.com/javase/technologies/desktop/javawebstart/index.jsp> (20 March 2009).

[20] Booch, Grady. 2007. *Object-Oriented Analysis and Design with Applications.* Second Edition. Addison Wesley.

[21] Bass, Len, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice*. Second Edition. Addison Wesley.

[22] Schmidt, Douglas, Michael Stal, Hans Rohnert, and Frank Buschmann. 2000. *Pattern-Oriented Software Architecture*. Volume 2. John Wiley & Sons, Ltd.