THE DEVSJAVA SIMULATION VIEWER: A MODULAR GUI THAT VISUALIZES THE STRUCTURE AND BEHAVIOR OF HIERARCHICAL DEVS MODELS

by

Jeff Mather

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN COMPUTER ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

2003

Table of Contents

Table of Figures

Abstract

This thesis describes the DEVSJAVA Simulation Viewer, a program written by the author to perform visualization of the structure and behavior of hierarchical DEVS models. The viewer meets a number of useful requirements with regards to its visualization and control capabilities, the most novel and important of which are its integrated, boxes-within-boxes-style view of the entire hierarchy of the model's components, and its animation of messages traversing across the couplings between components during each simulation step. The viewer's implementation employs software engineering techniques to extend the DEVSJAVA framework in such a way that there is little modification to the framework's core code, and also such that minimal changes need to be made to a model's source code for it to be displayable within the viewer. The performance of the viewer on models with a large number of components is also reported.

Acknowledgement

The author wishes to thank his thesis adviser, Professor Bernard Zeigler, for taking the time to review drafts of this document and offer suggestions as to how to improve its content and organization. Moreover, the author appreciates Dr. Zeigler's having allowed him the opportunity to work independently on the DEVSJAVA extension described by this thesis, and for granting him the freedom to implement this extension mostly as was seen fit by the author's own judgement.

The author would also like to thank Professor Jerzy Rozenblit and Professor Salim Hariri for their time taken to serve on the author's thesis committee.

Chapter 1: Motivation, Objectives, and Design Requirements

When employing the Discrete Event Simulation (DEVS) formalism through use of the DEVS for Java (DEVSJAVA) framework, the structure and behavior of a simulation model is specified using Java source code. Such a model represents a component which may itself be composed of a connected hierarchy of other component models. This hierarchical, connected component structure is not easily discerned by examining the model's source code. Neither is what the model's behavior will be at any instant during a simulation run, especially when the model is coupled to other such models, because it is only during the running of the simulation that this behavior becomes apparent, at least for non-trivial models.

Therefore, since the source code cannot by itself communicate these aspects of a model to a human modeler, a means was needed to intuitively and automatically visualize a hierarchical model's structure and behavior. Such a visualization system would have to take compiled Java model classes, create instances of those classes, then query those instances as to their structural properties, as well as their state, during the course of a simulation, to be able to know what to display to the user. This implies that the system must also be able to leverage the DEVSJAVA framework to run simulations directly on model class instances.

Specifically, the requirements for the viewing capabilities of the visualization system were as follows:

- To display an integrated view of potentially all the components in a hierarchical model, using an intuitive boxes-within-boxes visual metaphor to represent the components and their positions within the hierarchy

- To display the individual input and output ports of each component in the model
- To display the couplings of the components to one another
- To provide a graphical animation of the messages being sent across the couplings between components at each step of a simulation run
- To provide an abridged, updated-as-changes-occur visual representation of each component's state during a simulation run, as well as on-demand representation of the component's full state when bidden
- To provide all these aspects of the visualization in a timely manner, even for large models with potentially hundreds or thousands of components, and to keep the overhead of the visualization code from producing undue delays when running a simulation
- To provide the ability to run a simulation with only a minimal amount of visualization occurring - specifically, just the updates of the abridged display of the state values of the components involved
- To provide an optional, more limited extent of visualization for any component where, if the component contains any children within the component hierarchy, their display is elided, such that the component appears as a "black box"

Beyond visualization, there were also requirements on the system to allow the user to perform various control functions to facilitate interaction with the model under scrutiny:

- To allow stepping through, or continuous execution of, a simulation run on the model

- To allow execution of simulation runs in a real-time manner, with events occurring according to a wall-clock time-scale, and with the ability to scale these delays as a whole to be longer or shorter to make the simulation run slower or faster
- To allow quickly switching to a different model class to view from amongst classes populating different Java packages
- To allow injection of input values into input ports of components during a simulation run
- To allow drag-and-drop repositioning by the user of the onscreen boxes that represent components of the model, in order to more clearly communicate the model's design to the user and others

This thesis describes how the DEVSJAVA Simulation Viewer, a program written by the author and hereafter referred to as the SimView, extends the DEVSJAVA version 2.7 framework to meet the above requirements. The viewer is available as a portion of the DEVSJAVA 2.7 distribution downloadable from the DEVS website [DEVS], and has been used as an educational tool in the Discrete Event Simulation course offered at the University of Arizona by Professor Bernard Ziegler, the creator of the DEVS formalism.

The rest of this thesis is structured as follows. Chapter 2 provides a summary of the background research that was performed during the course of this thesis's work. This research focused primarily on the other systems that exist for visualization of discrete event simulation models, DEVS-based and otherwise. Chapter 3 briefly describes the functionality offered by the simulation viewer. Chapter 4 gives a quick overview of the design used in constructing the simulation viewer software. Chapter 5 expounds on various challenges that arose when implementing the software to meet the above requirements. Chapter 6 presents data concerning the performance of the viewer, especially on larger-sized models. Finally, Chapter 7 describes the conclusions that were reached about the viewer, as well as enhancements that could be researched in the future to help make the viewer more usable and useful.

Chapter 2: Background and Related Work

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

Section 2.1: DEVS

The SimView was crafted as an extension to the DEVSJAVA simulation framework, which in turn is an implementation of the DEVS formalism. Discrete event simulation in general, and DEVS in particular, were created as a way to more efficiently perform simulations on digital computers by focusing on the events that occur within a model as the simulation progresses, rather than on discrete steps in time or continuous rates of change of simulation variables [TMS]. By only expending computational effort at those times where meaningful changes in a model's state are taking place, discrete event simulation avoids wasting cycles on timepoints that are of no special consequence to the simulation's results.

DEVS goes beyond this underlying tenet to prescribe a means for decomposing a simulation model into a hierarchy of components interconnected through ports. Messages are passed between components across couplings that each span from an output port of one component to an input port of another. Rules are specified as to which components in the hierarchy may be coupled to one another in order to maintain a well-structured design. Specifically, a component may only be coupled to its parent, immediate sibling, and/or children components.

This hierarchy of connected components comprises the structure of a model to be visualized by the simulation viewer. The messages between the components in a model over the course of a simulation, as well as how the state values of the components change, together form the model's behavior, which is also visualized by the viewer.

A more thorough explanation of DEVS design and implementation is given in [Monitor]. While this explanation concerns an older implementation of DEVS in a language other than Java, almost all of its statements are equally true when applied to DEVSJAVA.

Section 2.2: Software Engineering Practices

As the SimView exists as an optional extension to the DEVSJAVA framework, it was important to extend the framework in such a way that no references to the added visualization code would be made by the core framework code. By strictly adhering to this rule, the core framework could still be released on its own as a valid DEVS implementation, albeit one without visualization capabilities. This required certain software engineering design practices to be employed to maintain the necessary lack of mutual dependency between the core code and the extension code. These practices are described in [DP]. The practice that was most put to use in this context was the Template Method pattern, which prescribes placing hooks into base code that may be overridden later by extending code to modify or add to what is taking place within the base code. The Observer pattern was used to attach object instances within the extension code as listeners to events occurring within associated core object instances. The Composite pattern was used to be able to treat a graphical view of one component within a model the same way as the individual views it contains for the component's children. The Decorator pattern was used to extend the core model classes to give them more of an ability to report updates about their internal state.

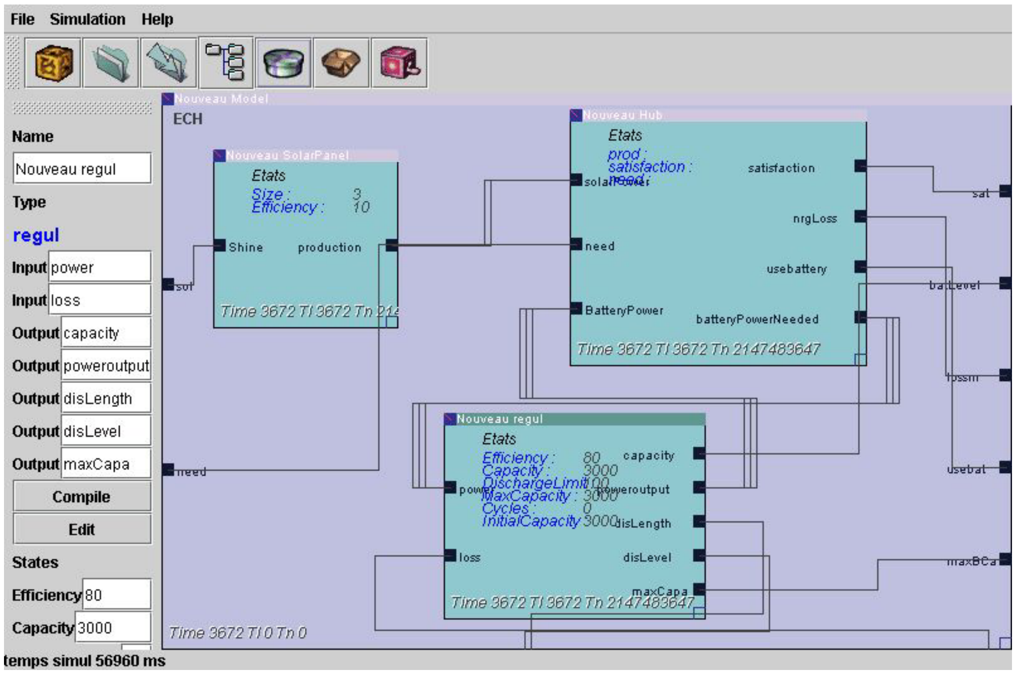
Section 2.3: Other Visualization Systems

In this rest of this chapter, strategies employed by other visualization systems for depicting the structure and behavior of discrete event simulation models will be briefly described. Since the set of such systems that is designed to work specifically with DEVS/DEVSJAVA models is very limited, the discussion will be expanded to systems that support other implementations of discrete event simulation modeling. Some of these were released after the major portion of the implementation work for this thesis had been completed.

Section 2.3.1: JDEVS

JDEVS [JDEVS] is a DEVS modeling environment created by Ph.D. student Jean Baptiste Filippi over the same general time period as that for the development of this thesis. JDEVS is not DEVSJAVA-based, but rather employs its own Java DEVS implementation.

JDEVS's manner of depicting a DEVS model bears similarities to that of the SimView. Components are displayed coupled to one another's ports with links, and textual labels display the component's updated state values as a simulation runs. JDEVS also enables construction of the model within the environment, and stores models in an XML-data file format. Additionally, when a component is selected within the environment's display of the model, edit boxes appear for the user to edit the names of the component's ports, as well as the current values of its state.



Name Nouveau regul
Type regul
Input power
Input loss
Output capacity
Output poweroutput
Output disLength
Output disLevel
Output maxCapa
Compile
Edit
States
Efficiency 80
Capacity 3000
temps simul 56960 ms

Figure 2-1: The JDEVS environment in action

JDEVS does not seem to support the simultaneous, integrated display of multiple levels of a hierarchical model, however, and it also does not display any animation of the messages moving across the links between the components.

Section 2.3.2: simjava

As its name implies, simjava [simjava] is written in Java, and it provides its own, non-DEVS discrete event simulation implementation. While simjava does not support hierarchical models, it exhibits similarities with the SimView as to how model structure and behavior are depicted. simjava "entities" (i.e. simulation model components) occupy rectangular areas within the view window, and are connected to one another by links to their ports. As a simulation runs, simjava displays events that are transmitted between the entities over their connecting links, and those events are represented with short textual descriptions. Text labels may be placed next to an entity to display aspects of the entity's state. Shown in the figure below is an example simjava applet whose simulation model depicts a work farm.

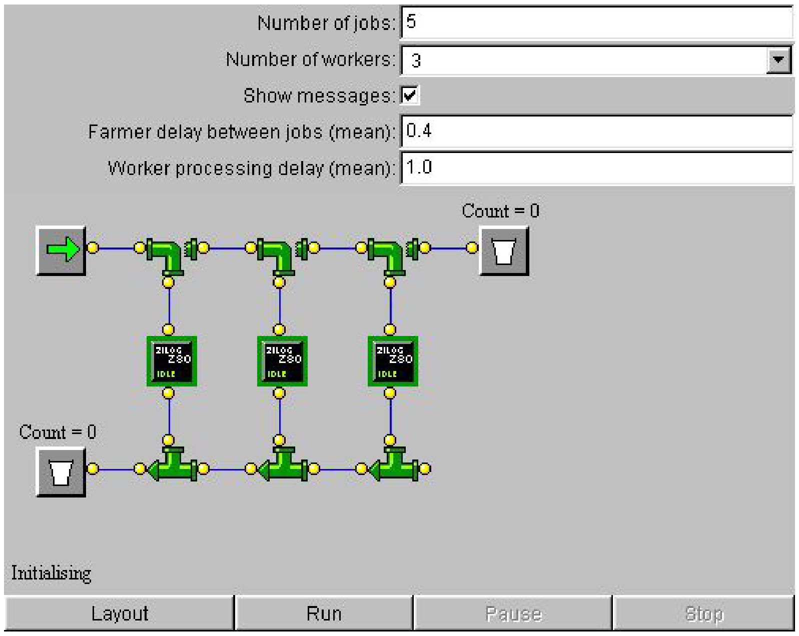
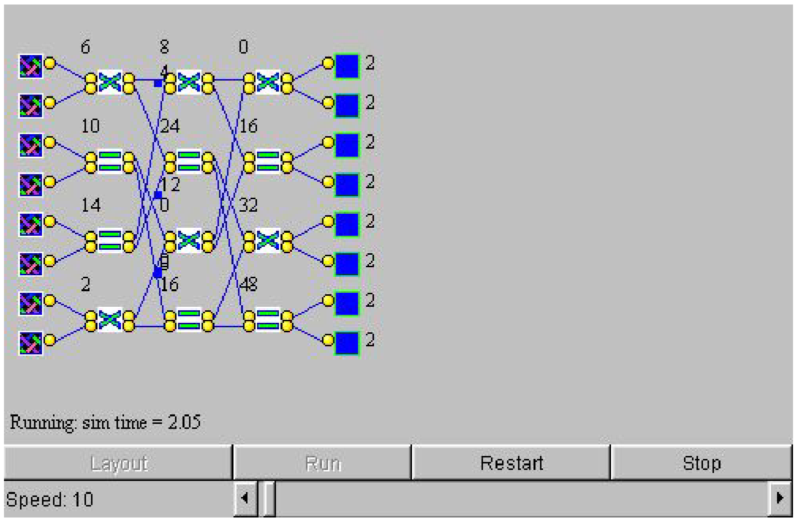

Number of jobs: 5
Number of workers: 3
Show messages: ✓
Farmer delay between jobs (mean): 0.4
Worker processing delay (mean): 1.0

Figure 2-2: A sample simjava applet

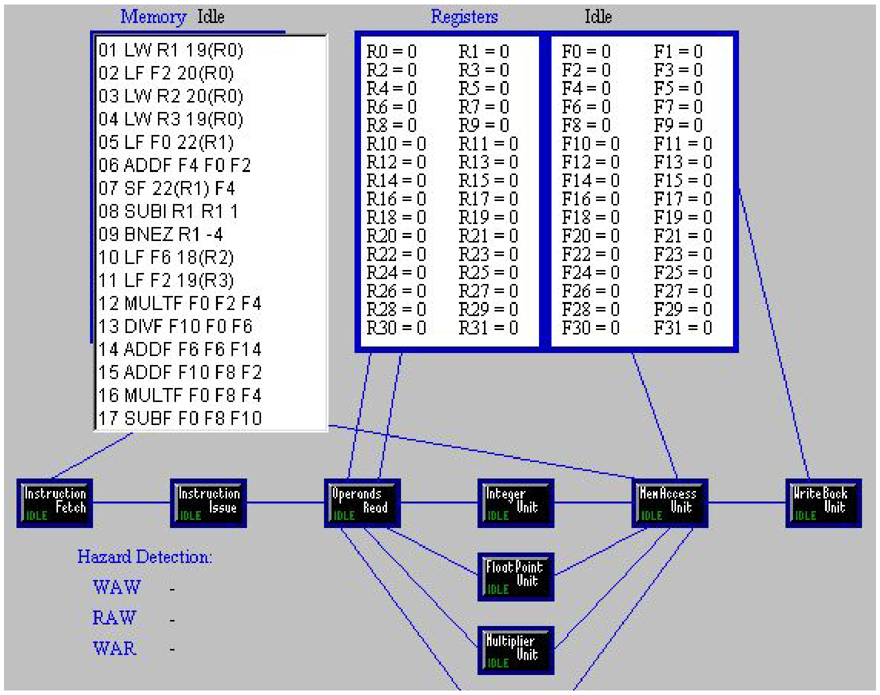
Interestingly, the simjava system lets the model programmer decide what is displayed within the onscreen rectangle that represents an entity in the model, as well as the dimensions of that rectangle. While this represents more work for the programmer, it also provides a great deal of flexibility, as the programmer is free to choose whatever text, graphics, or combination thereof best summarizes the entity's function. The programmer is also free to vary what is depicted over the course of the simulation run. In the figure below of a running simjava model of an omega network, the display of each interior entity flip-flops between a multiplication symbol and an equals sign.



```
Running: sim time = 2.05
```

Figure 2-3: A simjava depiction of an omega network

While, in the following figure, the textual displays for instructions held in memory, as well as the contents of the registers, are updated as the simulation runs.



```
01 LW R1 19(R0)
02 LF F2 20(R0)
03 LW R2 20(R0)
04 LW R3 19(R0)
05 LF F0 22(R1)
06 ADDF F4 F0 F2
07 SF 22(R1) F4
08 SUBI R1 R1 1
09 BNEZ R1 -4
10 LF F6 18(R2)
11 LF F2 19(R3)
12 MULTF F0 F2 F4
13 DIVF F10 F0 F6
14 ADDF F6 F6 F14
15 ADDF F10 F8 F2
16 MULTF F0 F8 F4
17 SUBF F0 F8 F10
```

```
R0 = 0    R1 = 0     F0 = 0    F1 = 0
R2 = 0    R3 = 0     F2 = 0    F3 = 0
R4 = 0    R5 = 0     F4 = 0    F5 = 0
R6 = 0    R7 = 0     F6 = 0    F7 = 0
R8 = 0    R9 = 0     F8 = 0    F9 = 0
R10 = 0   R11 = 0    F10 = 0   F11 = 0
R12 = 0   R13 = 0    F12 = 0   F13 = 0
R14 = 0   R15 = 0    F14 = 0   F15 = 0
R16 = 0   R17 = 0    F16 = 0   F17 = 0
R18 = 0   R19 = 0    F18 = 0   F19 = 0
R20 = 0   R21 = 0    F20 = 0   F21 = 0
R22 = 0   R23 = 0    F22 = 0   F23 = 0
R24 = 0   R25 = 0    F24 = 0   F25 = 0
R26 = 0   R27 = 0    F26 = 0   F27 = 0
R28 = 0   R29 = 0    F28 = 0   F29 = 0
R30 = 0   R31 = 0    F30 = 0   F31 = 0
```

Figure 2-4: A simjava applet wherein entities are represented with large textual displays that update

And below, the entities corresponding to cache entries light up green as they are filled during a simulation run.

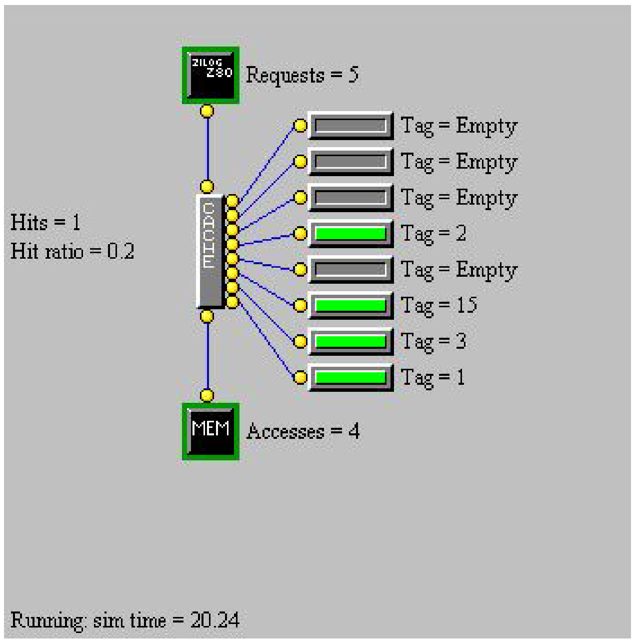


Figure 2-5: A simjava depiction of a memory cache

The above figures illustrate that allowing such customization of an entity's onscreen appearance can help the human viewer of a simulation to better understand what is occurring during a simulation run. Although it is not discussed elsewhere within this thesis, it is possible to override what is displayed for components in the SimView, however to do so would require a level of savvy with certain parts of the environment's source code, as well as fairly extensive experience using the Swing GUI component API. It may be useful in the future to make this ability far easier for the user to exploit, as well as to properly inform the user of its existence in the first place.

Another useful feature of the simjava package is its ability to concisely contrast over time the state of multiple entities by displaying that state within a timing diagram that is really a Gantt chart, as is shown below. Note the use of color to display what periods of time each entity's state equals a particular value.

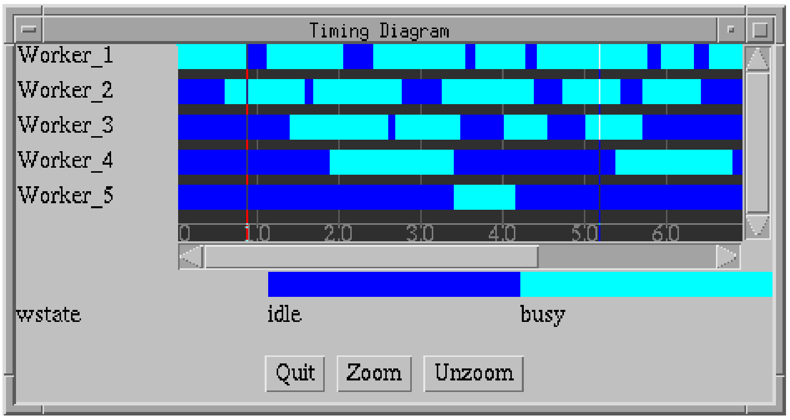


Figure 2-6: A simjava timing diagram

Section 2.3.3: GoldSim

GoldSim [GoldSim] is a commercial package whose forte is modeling continuous systems, but yet also allows the inclusion of discrete events at arbitrary timepoints. It provides built-in, prefabricated elements that the user connects graphically with links to create a model. There are data and function elements to choose from. The user may also link external programs to a model to provide functionality not present in the pre-supplied elements.

GoldSim supports hierarchical modeling but shows the children of only one node in the tree of elements within a view pane at the same time. As is shown in the figure below, a tree GUI component is used to allow quick navigation to any desired node in the tree.

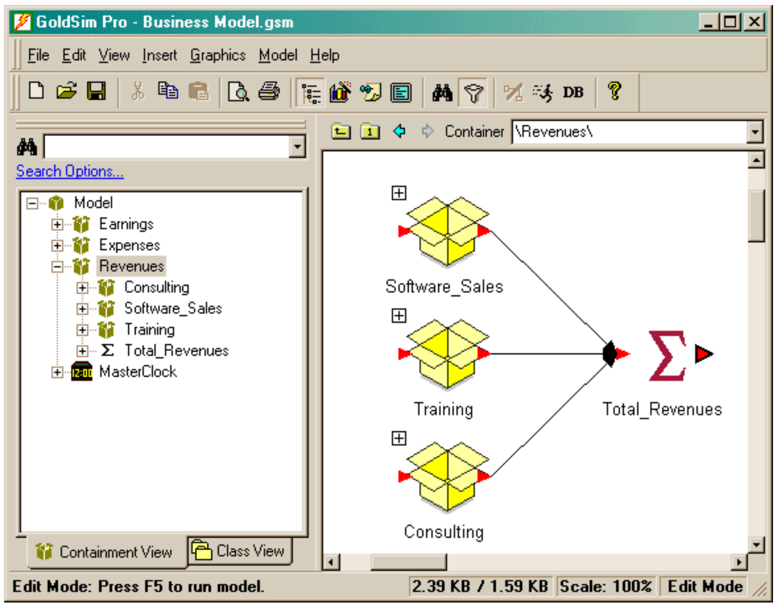


Figure 2-7: GoldSim's approach to hierarchical modeling

Given that models in this system are built mainly from pre-tested elements (with simple connections between them), as well as the fact that users of this system are more likely to care about the simulation results than the process of computing those results, no real effort is made in GoldSim to depict component interactions as the simulation progresses. Rather, emphasis is placed on providing strong support for various ways of graphing and charting the output data.

Being a commercial package, GoldSim has many helpful features that befit its status as a for-sale program rather than a research project, including: being able to specify the type of units used for any data value; automatic unit conversion of data values passed between model elements; the ability to import simulation input data from a spreadsheet or database; being able to record a revision history for each model; allowing the selective deactivation of any subtree of elements within the model's element hierarchy; annotation of a model's structural diagram with texts, pictures, and hyperlinks, for documentation purposes; creation of a "dashboard" interface for a model so that those unfamiliar with

GoldSim may easily execute a simulation on that model; and the ability to run simulations in a distributed manner across a group of machines.

Section 2.3.4: Ptolemy II

Ptolemy II [Ptolemy] is a modeling and simulation environment produced by Professor Edward A. Lee and a large group of graduate students under his direction at the University of California at Berkeley. It sports a very polished user interface and an expansive feature list. Its primary stated objective is to support the use of heterogeneous methodologies for simulation within the same model, from continuous to discrete event to many other domains, both well-established and experimental.

Vergil is the name of Ptolemy II's graphical user interface for the design of models and the execution of their simulations. Vergil provides easy-to-use model construction facilities, allowing the user to drag model components (which it calls "actors") from a hierarchical library of categorized such components, and to drop them onto a canvas containing a block diagram of the model. Custom actors may be created using Java source code.

Links are created between components by dragging from a port of one component to a port of another component. The Vergil interface utilizes an underlying block diagram editing interface called Diva which does a very effective job of drawing the links as connected segments that are orthogonal to the view edges, using curved corners to resolve ambiguity when two or more links overlap. Its approach in this regard is simple yet very attractive to the eye.

Vergil saves considerable viewing space within its model-view by representing components as mere icons in boxes, with small, unlabelled arrows attached to indicate ports. This is possible because the environment does not display state values attached to components as a simulation is run. Neither does it attempt to animate what is being passed between the components over their connecting links, although it does have an animation mode that highlights which components are sending or receiving transmissions at any particular time during the simulation.

Vergil facilitates the design of hierarchical models, but only shows the children of one node within the component hierarchy at a time within the same view pane. Multiple view panes may be created to allow the user to edit multiple parts of a hierarchical model simultaneously.

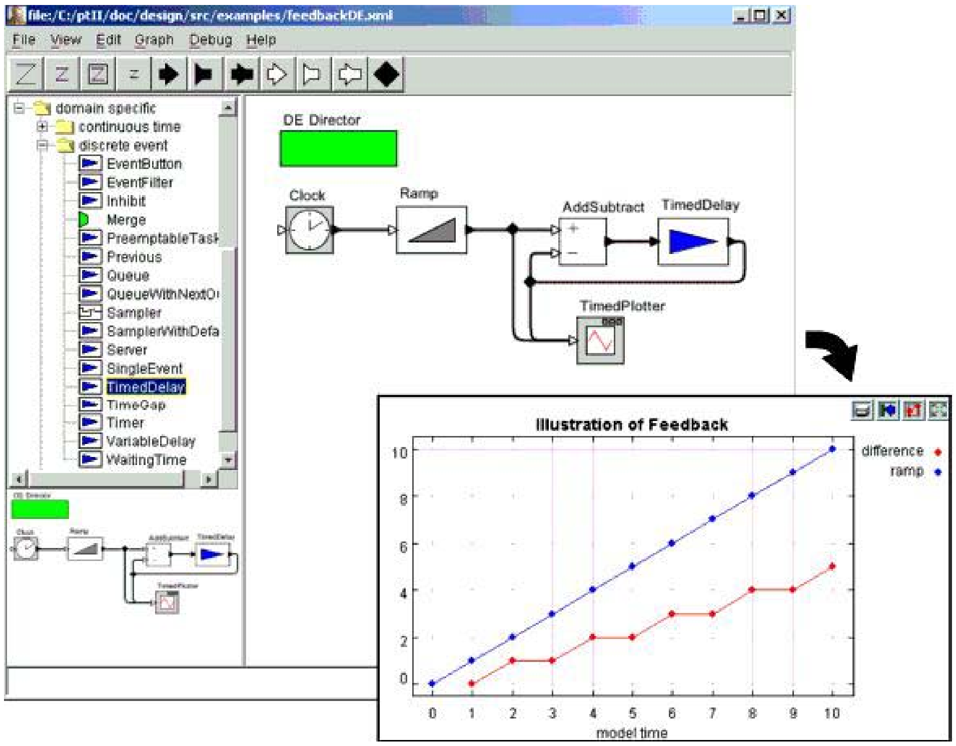


Figure 2-8: The Vergil model designer, along with a plot that might be produced during a simulation using the model shown

Vergil features extensive plotting and console display support for output values, with such plots and displays represented as components connected to any suitable output port of a component.

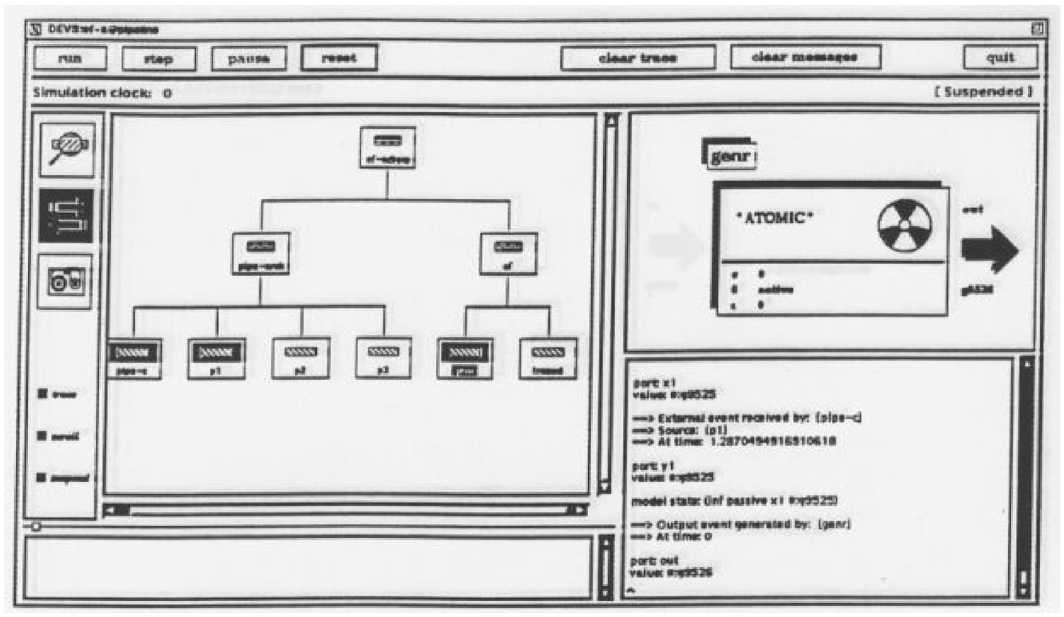
The environment also offers the ability to pan across, as well as zoom into and out of, a model whose display size is larger than the current view size. Any portion of the model's display may be zoomed into, and the lower left corner of the design window contains a scaled-down image of the entire model, over which a red outline box may be dragged to view that portion of the model within the normal display area.

The author concludes that Ptolemy II through Vergil offers excellent capabilities for constructing discrete event models (and models of many other simulation methodologies, as well as mixed models), performing simulations on those models, and viewing the results of those simulation runs. While the environment is not specifically DEVS-compatible, it would be an enticing prospect to leverage the efforts of Ptolemy II's many creators by implementing a DEVS extension to the package that would provide alternative discrete event support to that currently supplied with the program.

Beyond this, while the Vergil interface is very impressive in terms of its operation and its functionality, it will be seen that in comparison there are novel visualization features presented by the SimView beyond mere DEVS compatibility. In particular, the SimView may do a better job at communicating the operation of a hierarchical model during simulation time. However, being a far more comprehensive simulation package, the Vergil/Ptolemy II combination performs many functions and contains many features that are absent from the SimView. This was in large part by design, as the SimView's objectives are far more narrowly focused.

Section 2.3.5: The DEVS Monitor

The DEVS Monitor's first billing is as a simulation model debugger. However, it is obvious that in providing this functionality, the program is also in fact a visualization environment for model structure and behavior. It displays a graphical view of the structure of a hierarchical DEVS model, and allows simulations to be run on that model, either continuously or one step at a time. If a simulation is being stepped through, the user may inspect the state values of any component in the model, and may also modify them. In either running mode, the program can display a trace of the behavior of any of the model's components, including input and output events, as well as the internal state transitions resulting from those events. Besides such traces, the program also continuously highlights the next component that will undergo an internal state transition, and graphically displays at each step the events arriving at and issuing from each component (although, in a separate window panel than the model-view). The program can also selectively elide the display of trace messages for components of a digraph whose behavior the user believes to be correct, to prevent such messages from adding clutter to the trace output.



```
port: x1
value: #<q9525

---> External event received by: (pipe-c)
---> Source: (p1)
---> At time: 1.2870494916510618

port: y1
value: #<q9525

model state: (inf passive x1 #<q9525)

---> Output event generated by: (genr)
---> At time: 0

port: out
value: #<q9526
```

Figure 2-9: The DEVS Monitor main window

The DEVS Monitor displays the hierarchy of components in a model as a top-down tree. The problem with this style of depiction from the viewpoint of the SimView's requirements is that it would cause a large amount of overlap between the onscreen drawings of the couplings between sibling components, since the y-ranges of those components will usually coincide to a large degree. The depiction style works in this case because the DEVS Monitor does not attempt to display any couplings whatsoever.

As the program is almost ten years older than the SimView, it is based on an earlier DEVS implementation than DEVSJAVA, one written in the Scheme language.

Section 2.3.6: Less Similar Systems

There are other simulation environments featuring model visualization that were not described here because their particular domain, approach, and/or scope were too different from that of the SimView for them to contribute pertinently to this discussion.

These include the Multimodeling Object-Oriented Simulation Environment (MOOSE/OOPM) [MOOSE], ATOM3-DEVS [ATOM3], OMNet++ [OMNeT++], Omola/OmSim [OmSim], CD++, the Collaborative DEVS Modeler [CDM], and the commercial software packages AweSim [AweSim] and Simscript [Simscript]. As with the DEVS Monitor, both MOOSE/OOPM and ATOM3-DEVS display hierarchical simulation models using a tree graph with the normal textbook-style downward flow of components from the root to the leaves. ATOM3-DEVS's main purpose is to allow graphical creation of coupled DEVS models, from which Python source code is automatically generated. The present release version of the Collaborative DEVS Modeler is also geared towards model construction and supports collaboration by multiple remote users in the modeling process.

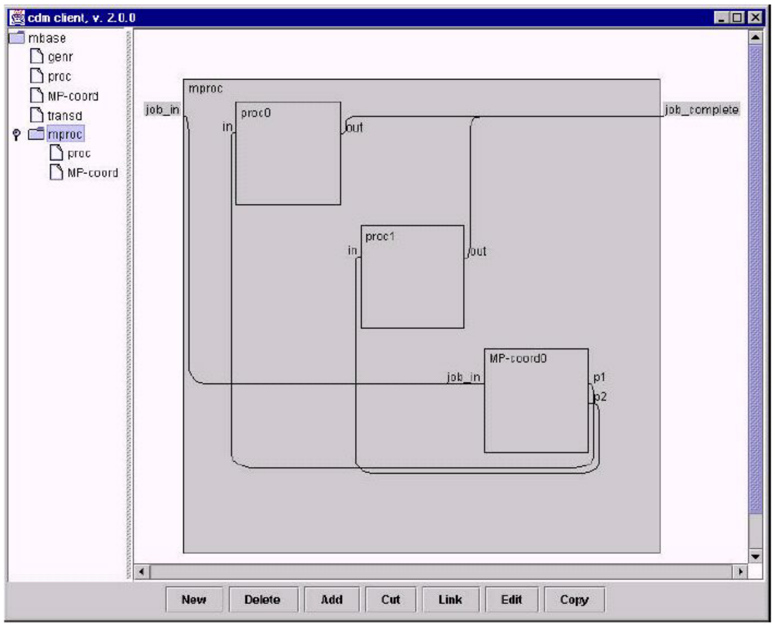
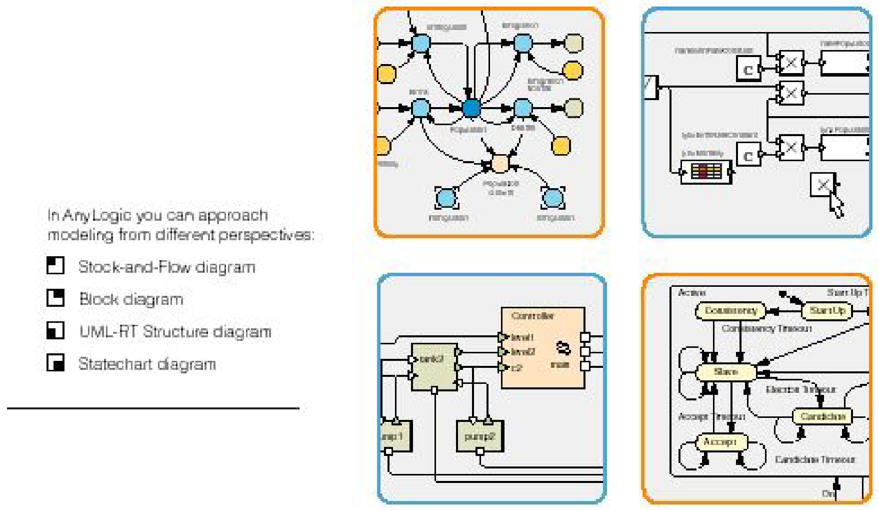


Figure 2-10: The Collaborative DEVS Modeler client program

The commercial package AnyLogic [AnyLogic] could not be properly evaluated due to its high cost. Its online brochure states that the program is capable of displaying hierarchical simulation models in both block diagram and UML-RT Structure diagram formats, as shown below.


In AnyLogic you can approach modeling from different perspectives:

- ◰ Stock-and-Flow diagram
- ◳ Block diagram
- ◱ UML-RT Structure diagram
- ◲ Statechart diagram

Figure 2-11: From the AnyLogic online brochure

Chapter 3: Functionality

This section briefly describes what the simulation viewer does, to give the reader an understanding of what was accomplished, as well as a suitable context with which to associate aspects of a later discussion about the challenges faced during the implementation phase.

The figure below shows the viewer in action, visualizing a hierarchical DEVS model on which a simulation is currently being run. What model is being visualized is selectable from the drop-down boxes at the top of the window. In the figure, the model is *HierarModel* from the *GenDevsTest* Java package.

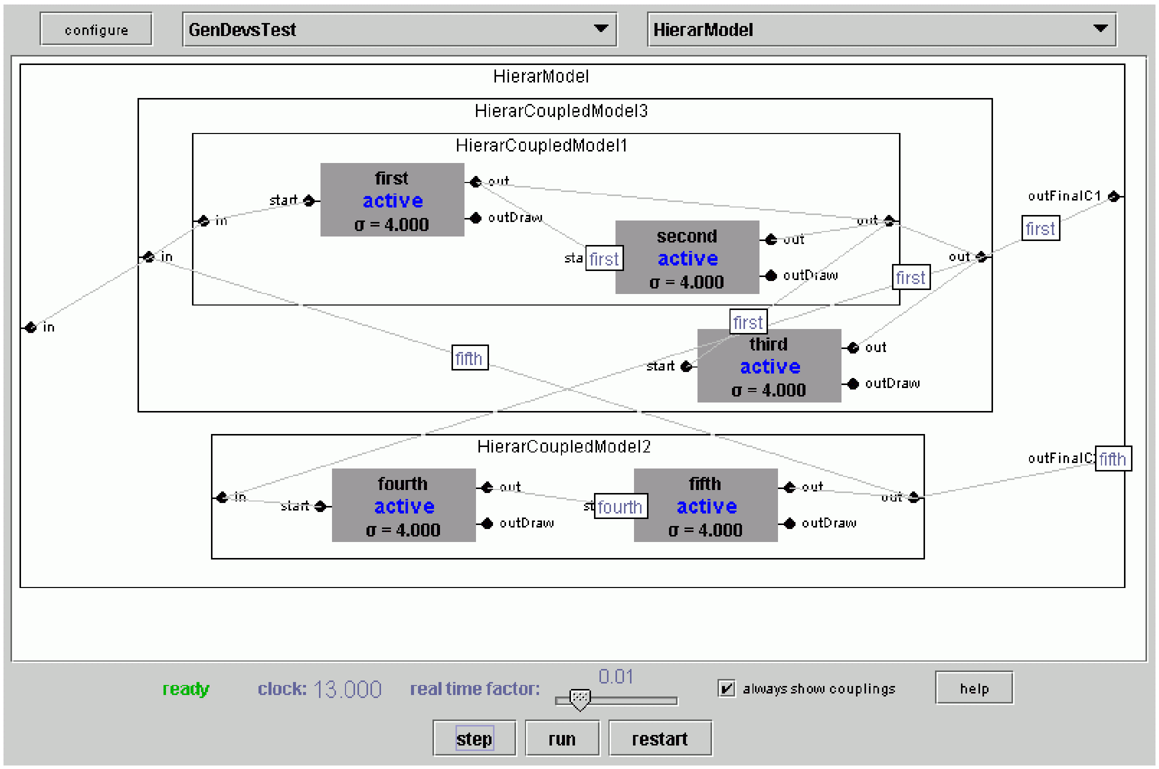


Figure 3-1: The simulation viewer visualizing the structure and simulation-time behavior of a hierarchical model

In the main model-view window, the *HierarModel* instance is depicted as the outermost box-outline, with one inport, *in*, and two outports, *outFinalC1* and *outFinalC2*. *HierarModel* contains other components, namely *HierarCoupledModel3* and *HierarCoupledModel2*. *HierarCoupledModel3* in turn contains another model, *HierarCoupledModel1*, as well as an atomic DEVS component, *third*. In the viewer, atomic components such as *third* are represented as solid boxes with inports on the left and outports on the right.

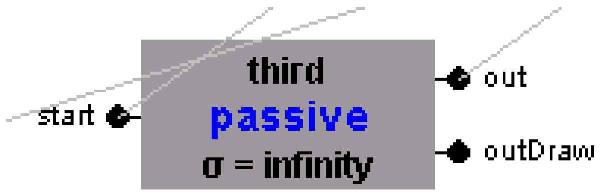


Figure 3-2: The depiction of an atomic component, *third*

Also shown in the model-view area are gray lines that connect outports to inports within the model; these depict couplings between components. When a simulation is run from within the environment, messages that components send to one another across these couplings are visualized as small boxes that display the message content and which travel along the couplings on the path from the message sender to the message receiver. In Figure 3-1, examples of such boxes are displayed, such as the *fifth* message that is traveling along the coupling between the *out* port of *HierarCoupledModel2* and the *in* port of *HierarCoupledModel3*.

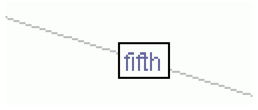
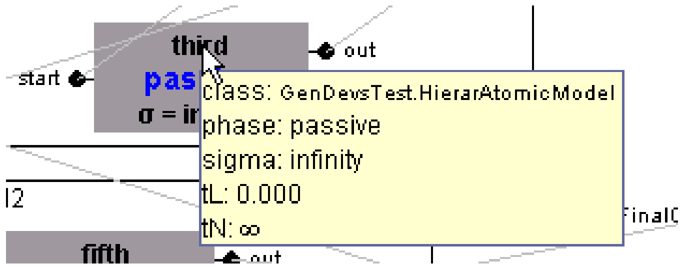


Figure 3-3: A message shown in transit across a coupling

As just alluded to, the user of the viewer is not only able to view the static structure of a hierarchical DEVSJAVA model, but can also view certain aspects of the behavior of the model during the course of a simulation. As shown in Figure 3-1, a button is present for advancing the execution of the simulation by a single event step at a time. When viewing such a step, the boxes representing messages being sent between the components of the model are visible in the view. In contrast, the toggle-button for running the simulation lets steps continue to execute one after another with no user intervention and no message boxes moving about. This lets the user more quickly advance the simulation to the desired time-point of interest.

If the model being viewed is geared for real-time simulation, then whether the execution is being performed in step or run mode, events will occur only after delays expire that are meant to simulate the passage of wall-clock time. The *real-time factor* slider can be used to speed up or slow down the environment's notion of the duration of wall-clock time by the selected factor, which varies by powers of ten from the baseline value of one.

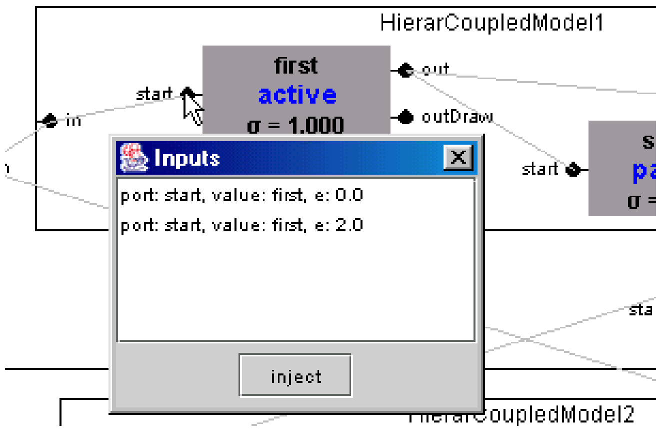
At any event-step within a simulation execution, aspects of the current state of any component may be viewed. For an atomic component, both the component's phase and sigma (i.e. time until next internal event) values are shown within the component's box in the view. When the mouse cursor is held over that box, a more complete view of the component's current state is presented in a tool-tip window. The code for the component decides which of its values to display in a such a manner, and how to present those values textually within the tool-tip. For a digraph (i.e. composite) component, such a tool-tip view of the component's state is made visible when the mouse cursor is held over the display of the component's name within the component's outlined box.



class: GenDevsTest.HierarAtomicModel
phase: passive
sigma: infinity
tL: 0.000
tN: ∞

Figure 3-4: The tool-tip display of an atomic component's entire set of state values

There is also the capability for a user to click on any inport of a component in the simulation being viewed and choose from a list of input values that may be injected into that inport. The value of the input, as well as the time to wait beyond the current simulation time to inject that value, are displayed for each list entry. It is an error for the user to choose an input whose wait-time would put it beyond the simulation's current time-of-next-event.



port: start, value: first, e: 0.0
port: start, value: first, e: 2.0

Figure 3-5: Bringing up the list of test inputs for a component's input port

How the components of a model are positioned within the model-view area of the simulation viewer window is determined by the model's code. Any digraph component gets to determine at least two aspects of its layout behavior. One is its own size, and the other is the positioning of its children components within its own bounding box. In turn, any of these children which are themselves digraph components get to determine their own size and the position of their children within themselves. A nice property of this system is that a digraph's layout and size need only be specified once for itself, and this specification may then be reused every time that component is composed within another digraph. Note that the size of atomic components is automatically determined by the program, and as they contain no other components, they have no internal layout to specify.

To make it easier to specify size and layout properties for digraph components, mouse-dragging behavior is used within the model-view to let the user position each component in a model, as well as to size those that are digraphs. When the user is done viewing the model, the program modifies the source code of each digraph whose size or internal layout was changed, such that the next time the source files are recompiled and the resulting classes are loaded into a new instance of the viewer program, they will possess the updated size and layout properties.

Given that a hierarchical DEVSJAVA model can consist of hundreds or thousands of components, the viewer provides a means of "clipping" the tree of components represented in the model-view to a desired level of detail. Namely, any digraph in the model can be told to display as a "black-box", meaning that none of its subcomponents or internal couplings will be shown. In essence, then, this collapses the subtree anchored at the digraph down to a single node, eliminating most of the clutter caused by that digraph.

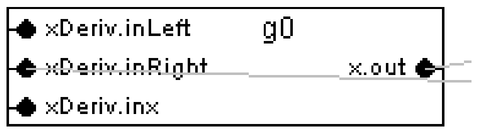


Figure 3-6: An example depiction of a black box component

Similarly, there may be components in a model whose display would add nothing to the user's understanding, perhaps because they perform only some sort of auxiliary function that is secondary to the model's true purpose. The *CellGridPlot* is an example of such a component, because it is used to provide a plot for an outport value of another component. Displaying the *CellGridPlot* components in a model would take up valuable screen space, while only cluttering the view, as they do not contribute to, or alter, the model's operation. Therefore, such components can specify that they should be hidden from the view, and in so doing the environment will also leave out all couplings connected to those components, as well as the display of any messaging involving them.

Chapter 4: Design

As mentioned before, the SimView had to be designed as an extension to the DEVSJAVA package. Existence as a pure extension implies that there must be no references to the visualization code from within the core code. An additional objective was to keep to a minimum changes within the core code necessary to support the SimView code. DEVSJAVA is intended to be a reference implementation of the DEVS formalism, which says nothing about visualization capabilities, so code added to the core that lies outside the confines of what DEVS specifies would cloud that intention.

Another important design goal was to avoid forcing the user to have to make substantial changes to a model's source code to use that model with the SimView. Such changes would represent a time-consuming burden that might discourage use of the viewer. They might also render the model incompatible with the core DEVSJAVA distribution. It would be best if models could be used as is, without any modification, but as will be discussed later on, this good intention conflicts with the arguably more important design goal above of keeping SimView-related changes to a minimum within the core code.

The next chapter will discuss the hardest challenges faced in meeting the requirements outlined in the introduction while also satisfying the above design objectives. We go on here to present a few UML diagrams that depict aspects of the high-level structure of the program's design. When reading the next chapter, it may be helpful in some cases to refer back to these diagrams to determine where within the larger picture the current topic focuses.

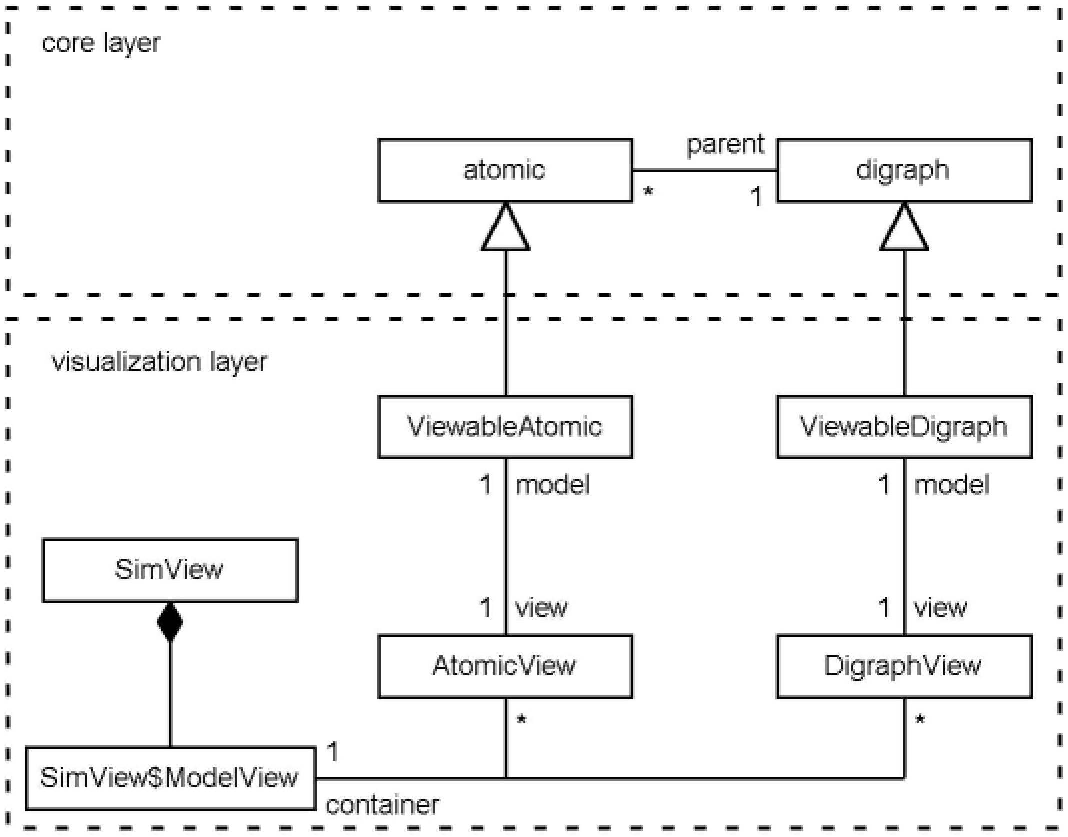


Figure 4-1: The relationships between extended model classes and their associated view classes

This first diagram shows how while the model classes in the core package are extended for the purposes of visualization, their extended forms themselves do not represent the views that are in seen in the program's model-view. Instead, associated with each extended model class instance is a separate view object with which it communicates and which represents the visualization GUI component that is added to the program's model-view container of such views. This particular subject is discussed in more detail in the next chapter.

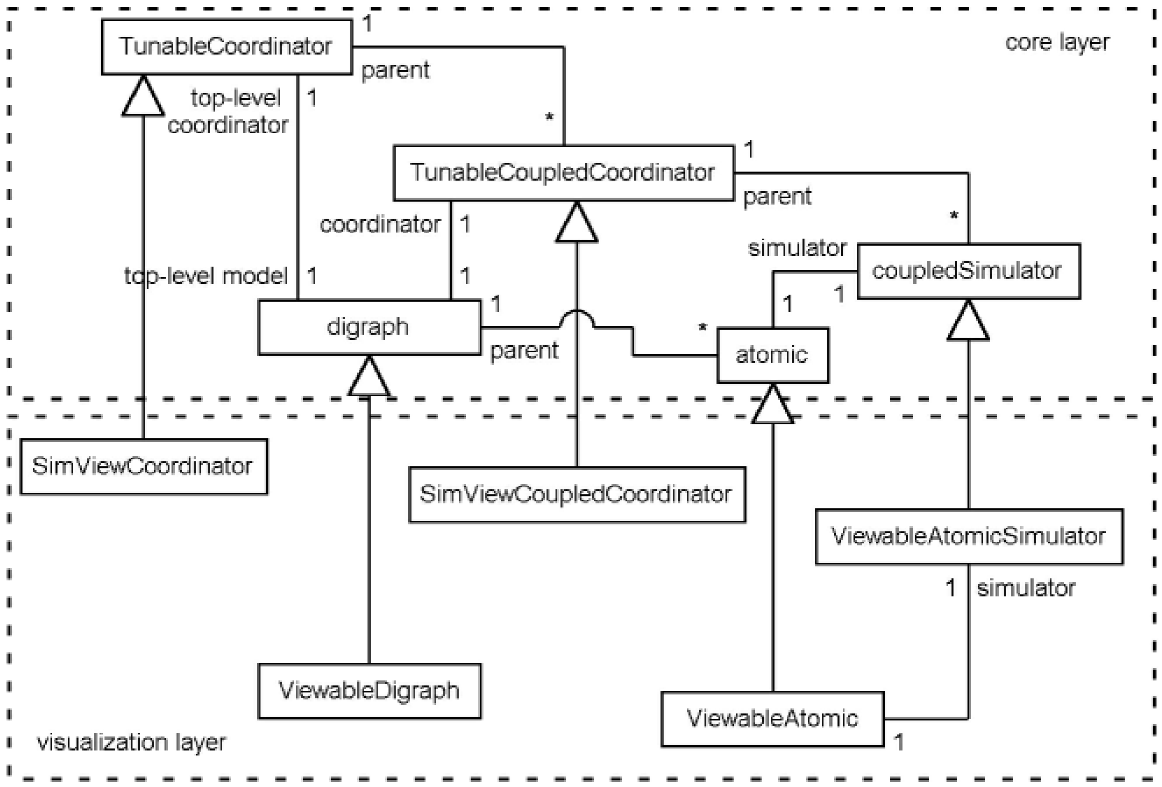


Figure 4-2: The visualization layer's various extensions of core simulation classes

This next diagram concerns the hierarchy of simulation components that is built by the DEVSJAVA framework to perform the actual simulation on the corresponding hierarchy of model components. Note how each of the classes involved in the core layer has a corresponding extension that was created in the visualization layer. Also note that most of these extensions use the associations with other classes they inherit from their superclass, rather forming a new association on their own with a different extending class (the simulator-of association between ViewableAtomicSimulator and ViewableAtomic being an exception). The need for these simulation class extensions is covered in the next chapter.

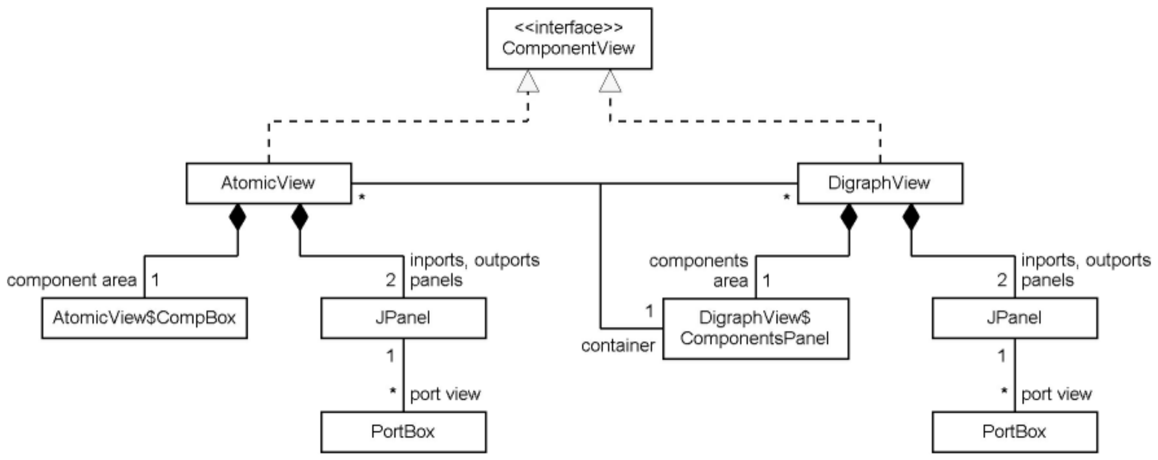


Figure 4-3: The structural decomposition of the component-view GUI classes

This final diagram concerns the structural composition of the visualization layer's GUI view components that are associated with their corresponding model components. Atomic views are decomposed on the left, digraph views on the right. Note that a digraph view's main drawing area may contain other atomic and digraph views. Both kinds of views implement the ComponentView interface so that the program's model-view area may treat them uniformly.

Chapter 5: Implementation as an Extension to the DEVSJAVA Framework

This section will detail the most difficult challenges that were faced when implementing the software to meet the requirements presented in the introduction.

Section 5.1: Providing Access to the State of DEVSJAVA Components

The main structural entities in a DEVSJAVA model are the components that comprise that model. As implied up to this point but not explicitly stated, in DEVSJAVA the classes representing these components are called *atomic* and *digraph*. The biggest difference between the two classes is that *digraph*s may contain other *atomic* and *digraph* components, while *atomic*s may not. Much of what needs to be visualized to properly depict the structure and behavior of a DEVSJAVA model consists of a substantial portion of the internal state of the instances of these classes. The vast majority of this internal state, however, was not exposed to outside classes. This was by design, as the encapsulation of such state follows good object-oriented programming practice. Now, however, there was a reason to expose this state, at least to the visualization code, and the question arose as to how best to do this.

Clearly, adding public methods for accessing the desired state values to these component classes that are part of the core DEVSJAVA implementation would not be a wise decision. As the visualization package would be the only consumer of these methods, their presence would needlessly complicate the public interfaces of these classes for all other clients, and would unnecessarily violate the encapsulation of the data being accessed.

Another tack that could have been taken, but that for the most part was not, would have been to attach one or more listeners to each instance of a component class. When a state value of interest within the component changes, the component then would inform the listener of that change. The listeners in this case would be elements of the visualization code, which would store the updated values, as well as display them as necessary. An advantage of this method, which is also shared by the method described before this one, is that it would help to facilitate the avoidance of having to modify existing DEVSJAVA model source code to make those models visualizable. The visualization code itself could be entirely responsible for gleaning the necessary internal state from the core component classes. A big disadvantage of this method is that it would require that at every place in the core component code where a particular state value is modified, that the appropriate listener be called with an update. Done improperly, code implementing this requirement could add a lot of clutter to the core DEVSJAVA implementation, and the likely possibility of not always remembering to call a listener every time code is added that modifies an internal value could very easily lead to the visualization code having an inconsistent view of a component's state.

A third method, the one that was chosen for the simulation viewer implementation, is to create classes within the visualization package that extend *atomic* and *digraph* whose (nearly) sole purpose is to provide public methods to access the desired internals of their parent superclasses. Thus, the classes *ViewableAtomic* and *ViewableDigraph* were created within the *simView* package to provide this functionality.

The main advantage of this method is that the code that exposes the components' data is localized outside of the core DEVSJAVA implementation. The major disadvantage of this method is that it requires at the least that any model to be visualized must be derived from one of these two new classes, rather than their parent classes *atomic* and *digraph*. This fact has certain negative implications. Extant (i.e. pre-viewer) DEVSJAVA models now require modification in order to be visualizable within the simulation viewer. Conversely, those models written with the viewer in mind must later be modified in order to work with a core-only DEVSJAVA implementation. It is not possible for the visualization code to simply create *ViewableAtomic*s and *ViewableDigraph*s on its own in place of existing *atomic*s and *digraph*s, because the creation of these components is coded within the initialization code of each digraph using *new* statements, which cannot be overridden.

These considerations demand, then, that the modifications made to make a component either visualizable or core-compatible be extremely limited in scope, preferably to the point where only the *extends* clause of the component's class needs to be changed between *atomic* and *ViewableAtomic*, or *digraph* and *ViewableDigraph*, and vice-versa. Otherwise, the time required to make the necessary changes might deter the user from employing the viewing environment in the first place. In the end, this goal was achieved – changing the component class's *extends* clause is the only mandatory requirement to make instances of that class appear within the viewer. While there are methods within the *Viewable*… classes that may be overridden to refine the display of a component, such as to specify how best to lay it out on the screen (in the case of a *ViewableDigraph*), what state values to show in the component's tool-tip, and what color the component should be (in the case of a *ViewableAtomic*), these overrides are optional and may be done long after the time the component is first displayed within the viewer.

Section 5.2: Separation of Component State and Display Concerns

It makes sense to leave to each DEVSJAVA component the responsibility of determining how it should be displayed within the model-view, as this keeps such component-related functionality within the code for those components, and not strewn elsewhere. Beyond display appearance, there are other aspects of representation within a GUI, such as listening for mouse events, that are also best delegated to the components themselves.

One path that could have been taken to add the aforementioned display and other GUI-related capabilities to DEVSJAVA components would have been to simply put all the functionality directly into the *ViewableAtomic* and *ViewableDigraph* classes, and treat them as first-order GUI components. However, this would have obscured the main purpose of those classes, which as stated before is to expose the internal state of their parent classes *atomic* and *digraph* for display in the viewer.

It was decided that it would be better to create separate classes whose instances would serve as the GUI components in the model-view, and these instances would then query the *ViewableAtomic* and *ViewableDigraph* classes for the necessary state to display. *AtomicView* and *DigraphView* are the classes that resulted from this thinking. Furthermore, the relationship between each correspondingly-named pair of classes is bi-directional, as a *Viewable…* instance makes calls on its associated *…View* instance to inform it of changes in state within its particular DEVSJAVA component.

Section 5.3: Addition of Hooks Into the Core DEVSJAVA Simulation Workings

To perform a simulation on a hierarchical DEVSJAVA model, the DEVSJAVA framework creates a like-structured hierarchy of simulator and coordinator objects. Each of these objects is responsible for conducting the simulation workings of its associated structure-side component in the model. Simulator objects get assigned to atomic components of the model, while coordinators are assigned to the digraphs. There is a top-level coordinator (or simulator, when the model is simply an atomic component) that is associated with the model itself which presides over the simulation-side hierarchy of objects performing the simulation.

Analogous to the situation where it was necessary to provide external access to state values within the structure-side components, there are many events that occur during the operation of the simulators and coordinators which contain information needed by the viewer to perform its job, yet these events were in no way exposed to outside classes, as they had been considered to be internal workings only. Examples of such events would include that a component's phase value has changed, that the elapsed simulation time has increased to a new value, that a component has issued a message on one of its ports, that a message has passed from one component to another across a coupling between them, and that the simulation has completed a certain number of event-step iterations. Not only are there such events, there are also control points within the code of these objects at which the viewing environment needs to be able to influence what occurs, such as having the opportunity to tell a real-time simulation to wait for a certain amount of time to elapse before injecting an input into a port of a component, or being able to decide what type of simulator or coordinator object to construct at each position in the simulation-side hierarchy of objects.

Here again, a way was needed to provide the visualization package with access to internal workings of the core DEVSJAVA implementation, without adding to that core any references to the visualization package's code. Three different solutions to this problem were considered. The first was to allow the attachment of a listener to the simulation-side objects, where the code of those objects would report events of interest to the listener at various points of its execution. The listener, in this case, would have been some element of the visualization package, although from the core's view, it would have been making calls on some generic listener interface of its own specification. The main problem presented by this method was that the simulation-side code needed to not only report events, but also to allow itself to defer control at certain spots where the visualization code would have need to do things its own way. It is not expected of a listener to exert such control, as it is by nature supposed to be just listening for events.

The second approach, therefore, would have extended the first by attaching to the core simulation code an object that is not only a listener, but also would also be expected to exert some control when called upon. The question was, from the point of view of the core, what to call the interface of this object to give its presence a well-understood meaning in the absence of the visualization package for those developers working with just the core code? The presence of this object would have been entirely to the benefit of the visualization package, but to call it, say, a *SimViewProxy* would have in the opinion of the author been too explicit a reference to the visualization package's existence, even though the *SimViewProxy* class would have been defined in the core, rather than the simulation viewer package.

It was decided, then, that the best course of action would be to add in “hooks” to the core code at the event and control points, and then extend the simulator and coordinator classes from within the visualization package to override those hooks with methods that either acted upon some event or exerted control during the midst of the executing parent method. These hooks were each documented as being necessary to fulfill the needs of an extension package to the core DEVSJAVA framework.

What allows this strategy to be used is the manner in which the simulation-side hierarchy of simulators and coordinators is built before a simulation on a model is to take place. DEVSJAVA dictates that at each level of the hierarchy, every coordinator specifies what type of coordinator or simulator each of its children objects will be. Additionally, a nice property of DEVSJAVA is that to create a simulation execution instance one must simply create an instance of the top-level coordinator (or simulator) to assign to the model, and then the process of filling out the simulation-side hierarchy begins and runs until the leaves of the hierarchy’s tree structure are populated by simulators. Therefore, the simulation viewer, when it wishes to execute a simulation on a model, can assign one of its extensions to a core coordinator (or simulator) to the model, and for each of the model’s components, the extended coordinator will create further instances of the extended coordinators and simulators, and so on, down the hierarchy. In this way, it is ensured that every relevant event- and control- oriented hook in the core simulation classes is properly overridden with logic from the visualization package.

Section 5.4: Keeping the Display of Message Passing Between DEVSJAVA Components Efficient

During a simulation run, DEVSJAVA components communicate with one another by sending messages across couplings that connect them. Such communication is never directly specified, however. A component can only specify on which output port it would like to send a message, and similarly, it cannot choose from which other components it will receive messages on any of its inports (taking into account that direct feedback loops are not permitted). It is up to the containing digraph of that component to couple the inports and outports of the component either to its own inports and outports, or to those of the other children components it contains. These restrictions help to ensure proper componentized design for every hierarchical model.

Even considering these facts, it is possible to imagine a scenario in which every component that helps to compose a model could at each event step send one or more messages to every other component in the model, meaning that a number of messages on the order of the square of the number of components in the model could be traveling around at each event iteration of the simulation. It was therefore vital that the presence of additional code to display the movement of the messages not add a significant amount of processing overhead when the simulation is being run with such display turned off to reach a certain point in the proceedings. It is understood that if such display is desired, the amount of time necessary to display the messages' movement animations to the user will dominate, and as the user is viewing such displays to either understand or debug a model, the delay is beneficial rather than detrimental.

Let us examine how the display of the messages traveling between the components at each event step was implemented, to illustrate how the visualization code does not produce any more than a constant amount of processing overhead per message displayed. To begin with, at some time during the simulation iteration for the current event step being processed, the coordinator (or simulator, although for this discussion we assume it is a coordinator) object for an associated model component determines what messages that component will output (on which ports) for that step. Now the coordinator's class type is one of those from the visualization package that extends from a core coordinator class as described earlier. Therefore, it provides an override for the hook in the core class that resides at the point just after the output messages have been determined. This override passes the output messages along to a listener attached to the extended version of the coordinator, which turns out to be the model-view portion of the viewer display where the messages will be depicted. The very first thing the model-view does with this notification is to check to see if the simulation is currently being run one step at a time (i.e. waiting for the user to say when the next step should be run), because if instead the simulation is running with no such pause between steps, we do not want any display of messages to occur. It is important that this check is performed here, before any further work is done with the message, to eliminate needless overhead that would keep the environment from performing well while in "run" mode. For the same reason, another check is performed here, this one to see if the source component is a *ViewableAtomic* or *ViewableDigraph*, because if the component is not viewable from within the viewer, there is no way to depict the message as issuing from it. It may be surprising to the reader to learn that a check is not performed at this point to see if the source component is hidden, either due to being within a containing black-box component, or because the component's code simply specifies that it is to be hidden. This is because, even though part of the path the message will take during this event-step will be hidden, we still want to properly store the path, as there may be parts of it that consist of two coupled, non-hidden components, and we want these legs of the message's journey to be depicted. So, we cannot just skip creation of the path altogether simply because the source component is hidden.

If the simulation is being run in "step" mode, the model-view must now start to keep track of the path the message is taking across couplings on the way to its destination component. The message will eventually be displayed traveling across these couplings from the source outport to the destination inport. The model-view therefore creates a path object to hold the ports visited along the path specified by the couplings. Each port is considered to be one step along the path. The outport of the source component is considered to be the first step, so a step object representing this port is added to the path object. The path object is then cached for later retrieval.

Subsequently during the computation of the current simulation interval, the message will be passed from the source component to the component at the other end of the first coupling in the message's path, and then onto the component at the end of the next coupling, and so on, until it reaches the destination component. At each point in the process where a core simulator or coordinator (again, we assume here it is a coordinator) determines what component is at the other end of one of its own component's outport couplings, there is another hook that is overridden (which can happen since, once again, the coordinator is an instance of a class from the visualization package that extends a core coordinator class). This hook reports to a listener that a message has crossed a coupling, and as before the listener is in this case the viewing environment's model-view.

After the model-view performs checks to see if the component at the destination side of the coupling is of a viewable type (i.e. *ViewableAtomic* or *ViewableDigraph*, as before) and to ensure the simulation is being run in step mode (if it is not, no further processing is done by the model-view), a step object is created for this step of the message's movement along its path, and the message's path object is retrieved from the cache for such objects. At this point, a copy of the path object is created, including a copy of the order of the steps within the path object. The steps themselves are not copied, but are merely referenced again by the new path object. The path object copy is then used instead of the original. This is so the original can be left as-is, which is necessary because the message path might have branched at the source side of the coupling just traversed, meaning there will soon be other notifications that the message has arrived at the destination end of each of the other branches. The approach taken here is that a separate complete path is to be stored for each branch taken by a message. Animations will be performed concurrently for all the paths the message takes.

There are some optimizations that could have been performed here to avoid unnecessary path copying, but were not. Making a copy of the path after every coupling has been traversed could often be avoided by checking to see if any branching was done at the source end of the coupling. Also, maintaining a complete path for every branch taken by a message could have been avoided by instead storing the tree of paths taken from the source outport to every destination inport, and during the resulting animations making sure to create new message display elements for each branch at a branch point, and then visually sending them on their way across their branches. These optimizations, however, would have complicated the extending visualization code, and the assumption was made (although not proven) that in the average case, the number of couplings traversed by a message during an event step along a path contained within a hierarchical model would very likely not exceed a fairly small constant. This assumption is what lets us say that the above work does not add any more than a constant amount of processing time to each step traversal notification.

Only after the above copying of the path object is performed is the new step object added to the path represented by the path object. Now that the path has at least one step in it, we can start the display of the message movement animation in a separate thread, confident that by the time the onscreen message reaches the end of the first coupling, the rest of the path's steps will have been filled in by the simulation thread.

An important point implied by the above description is that the visualization code does not just listen for the message's final destination (for the current event step) to be computed, and then spend time computing the intermediate steps along the path on its own (as this path information is not saved by the core code to be reported along with the destination), which would have made the interfacing with the underlying core code much simpler. Rather, a hook was placed deeper into the process at the point where each step along the path is computed, which saves this work from having to be wastefully redone by the simulation viewer code. The simulation viewer package, in effect, extends the core code by performing the caching of the path information it needs to animate the message's traversal. As this path information is built as the path itself is being built, if we can avoid adding any more than a constant amount of running time to the addition of each step in the path, we know that we have not increased the asymptotic running time of the simulation code with visualization versus that having no visualization

Section 5.5: Providing the View of a Hierarchical Model

The model-view panel of the viewing environment provides a novel depiction of a DEVSJAVA hierarchical model. While the hierarchical structure of the model implies that what is really being presented is a tree of components, what is displayed does not resemble a tree at all. Rather, a visual metaphor of enclosing boxes is employed. Each node in the tree is represented as a box which contains the boxes of its children components within the tree. The outermost box displayed is therefore the model itself, and as we move from a containing box into a box it contains, we are moving down the component tree. The leaves of the tree, which represent the atomic components in the model, are displayed differently, as solid boxes, to indicate their status as leaves, although it will be suggested later on that for uniformity's sake this should be changed to no longer happen.

Each enclosing box within the model-view is a *DigraphView*, which is a Java Swing component. The boxes contained within this box are themselves *Digraph*- and *AtomicView*s and are added directly to the parent view, which employs no predefined layout. So, the model-view leverages the existing Swing component containment system to handle the display of components within other components. This provides other benefits, as well. When the position of a box within its enclosing box is changed by dragging the enclosed box around, the dragged box's own enclosed boxes move around with it, and they maintain their positions relative to the dragged box's origin. Also, the underlying Swing code handles the determination of which component within the tree the mouse cursor is currently overlapping, to know to display that component's tool-tip to provide the user with more information about that particular aspect of the model's state.

What also makes the viewer's depiction of a hierarchical model novel is that the subcomponents of any node in the tree may be elided from the display, resulting in the ability to deliver to the user a totally integrated, completely tailored view of whatever subset of the component tree's elements is desired. This is important, because it means the user can see exactly those parts of the model that are of current interest, and the interactions between those parts, without having to switch between windows and/or mentally correlate the elements of two or more views with one another.

The tailoring of what parts of a model are and are not depicted within the display is specified in a hierarchical way, to keep every component's hidden or shown status sensitive to the context in which it is being displayed. That is, each digraph at some level within a hierarchical model not only is allowed to override this status for each of its contained components, but can also delve into those components to override the status of any of their subcomponents, and so on. Therefore, a component that is normally hidden by its parent component may be exposed for display by some other component that decides to contain the parent.

There is currently a suboptimal aspect to the viewer implementation that keeps this flexibility from being as useful as it should be, however, and this will hopefully be remedied in the near future. This has to do with the fact that as subcomponents are selectively hidden or shown within a parent component, the screen real estate needed to depict that component increases or decreases. Unfortunately, right now when the user manipulates the size of the parent, and the positioning of its subcomponents within it, to account for this change in overall size, the updates are made directly to the parent's source code. This makes sense only when the parent happens to be the hierarchical model itself (i.e. when the parent is the top-level component in the model). Otherwise, what should be done is to have the resizing and repositionings be specified as overrides at the level of the model itself. In other words, if the model is going to delve deep into its own hierarchy to specify what is and is not shown, it needs to provide its own specifications for how big, as well as where, the affected components are shown, and that particular "view" of the affected subcomponents should apply only to the context of that model. As it stands now, such changes are made to the source code of the subcomponents themselves, and therefore affect every context in which those components are utilized. This is obviously a bad situation.

Section 5.6: Screen-Efficient Couplings Display

It can be seen by referring back to Figure 3-1 that couplings between components in a model are displayed as a thin, gray, straight line between the two ports that are coupled. A fancier way to depict couplings would have been to determine paths for them such that they would not overlap any of the components being displayed. Often, when this approach is taken, the paths chosen contain only segments that are orthogonal to the axes of the program's window. However, it is much more difficult to implement such a scheme, and moreover, given what was said above about how the model-view can potentially provide a depiction of the entire component tree of the model, it could be asked, is it worth the screen real estate that would have to be consumed to make room for coupling paths that do not overlap the components and, for the most part, each other?

The decision made was that the direct-line-between-ports approach that was implemented is unobtrusive enough as to not significantly detract from the user's comprehension of what is being displayed, and that whatever benefit that could be gained from the fancier approach would not have been worth the major coding and maintenance work that would have to have been expended. It is also the opinion of the author that the direct-line approach makes it quicker to ascertain to which component a message is destined as a simulation is being run, rather than having to follow a twisting path of connected line segments as it makes its way around intervening components.

The choice of the simpler method for displaying couplings implies there will be situations where a coupling and a component overlap, and the decision was made that in all such cases the coupling should appear in front of the component. However, utilizing the standard Java Swing model of hierarchical GUI component containment presents a problem in this regard. While under this model components are allowed to overlap and will display as such, there is no means to specify the order in which they are drawn, such that the couplings would always be drawn after the simulation model's components are drawn, so that they appear to be in front. The workaround was to declare the model-view itself to be not just a normal Swing *JPanel*, but a Swing *JLayeredPane* which allows other components to be added to it at different perceived depth levels, where the levels are drawn in reverse order. A separate couplings panel (of the model-view's size) was added to the model-view panel at a lesser depth level than the panel that holds the *Atomic*- and *DigraphViews* themselves. The couplings are drawn within this transparent panel only after the underlying component views are drawn.

Similarly, the message-boxes that travel across the couplings make use of this layered-pane functionality to be displayed always in front of the both the couplings and the simulation model's components.

Chapter 6: Performance

As stated previously, it was an important design goal to extend the DEVSJAVA framework with visualization capabilities in such a way so as to prevent the user from experiencing major delays when running simulations due to the overhead of the added code. It is often the case that the user needs to be able to run a simulation up to a given point within the viewer, so that the model's operation within a timeframe immediately subsequent to that point may be studied. The presence of unnecessarily added delay while doing this, when a model's simulation may already be slow due to the presence of hundreds or thousands of components in the model, may make the visualization environment intolerable to use.

With these facts as a major concern, informal timings of the simulation runs of several (mostly) large-size models were taken, both within the confines of the visualization environment as well as without visualization. "Running" here denotes that the simulations proceeded through until the given time point with no user input required between event-slice iterations. The results are presented in the following table.

Model name	Number of components	Number of components visible	Simulation time reached	Running time without visualization, N (sec)	Running time with visualization, V (sec)	Slowdown factor (V-N) / N
package devsIntegration:						
ShockMulti	141	21	880	22.7	24.7	0.09
Acoustic	64	61	100	31.2	56.6	0.81
Harmonic	10	5	1000	8.7	20.8	1.39
DiffuseMulti	351	51	6	53.0	49.4	-0.07
package Continuity:						
vehicleSpaceMixed	32	30	100	8.8	20.9	1.38

Table 6-1: Comparison of simulation running times, with and without visualization

From the data, it can be seen that for models where onscreen updating of multiple state values is continuously occurring (e.g. the Harmonic and VehicleSpaceMixed models), the simulation viewer does indeed add a noticeable delay to the proceedings. However, this is to be expected, because rendering the updated values to the screen takes time, and these updates do serve a valuable purpose in that they can inform the user when a particular point of interest in the simulation may be getting close. On the other hand, for models which contain so many components that the only ones that are shown (to prevent overwhelming the user with too many details) are top-level black-boxes (which do not display any state), such as the ShockMulti and DiffuseMulti models, continuous visual updates of state do not occur, and in these cases the viewer performs about the same as running the models on their own with no visualization code attached.

Chapter 7: Conclusions and Further Research

The SimView fully achieves the requirements outlined during the introduction of this thesis. To summarize the most important of them, the SimView takes DEVSJAVA model classes as input and provides a novel visualization of their structure and simulation-time behavior. This functionality is provided as an extension to the DEVSJAVA framework in such a way that there are no explicit references to the visualization package's code from within the framework's core classes. Furthermore, the presence of the visualization workings does not impose an undue performance penalty when running simulations, even for large models.

No other visualization system for any discrete event simulation formalism that was evaluated by the author during the course of this thesis's research provides the SimView's integrated, components-within-components-style view of the entire hierarchy of components in a model at once. Neither do any of them animate the movement of messages along the paths of couplings between the components, at least not for hierarchical models (simjava does provide this for flat models). Finally, none of the DEVS-based systems provide a means for injecting input values into inports of components during a running simulation.

Software engineering design patterns were used to add these visualization capabilities to the DEVSJAVA framework without making it dependent on their presence. Such techniques could asssist in the proper addition of visualization support to any existing software system, in that modifications to the base code are kept at a minimum, and the base code is prevented from having to make references to the visualization code.

The Java Swing GUI-component containment system, together with Swing's *JLayeredPane* GUI-component container, were also leveraged to provide a substantial portion of the functionality necessary to achieve the components-within-components style view of a hierarchical model presented by the SimView. The novel manner in which this Swing functionality was employed could be used in any situation requiring that an intuitive, editable block diagram of a system be displayed.

This thesis concludes with the following provision of many ideas that should be researched to make the SimView more informative, more interactive, and easier-to-use while performing visualization of hierarchical model structure and behavior.

Section 7.1: Automated Plot Support

While the simulation viewer does a good job of visualizing the behavior of a model during a simulation at any given instant of time, it is often more useful to study the data generated across the course of the simulation, and this is currently an area of weakness for the viewer. It is a cumbersome process to specify that a plot be shown of some value of interest over the time of a simulation's execution. A *CellGridPlot* component has to be added to the model and coupled to an output of the component in charge of computing the value to be plotted. The plot window that results is separate from the main viewer window, and thus must be positioned on the screen either by the user or programmatically (amongst all the other plot windows and the main window). Presently, the plot window is not even a non-modal dialog window, meaning that bringing the viewer window to the foreground will not bring the plot windows along with it.

It would be far better if the user could demand a plot for any component inport, outport, or state value over time from within the simulation viewer interface itself. These plots could be added to the right side of the model-view area of the viewer window, stacked vertically so that their respective time axes would be in alignment. Which plots were selected could be saved to a program data file to be remembered for the next time the particular model is loaded into the viewer. The user could also specify when a plot is no longer needed, so that it may be removed from the display.

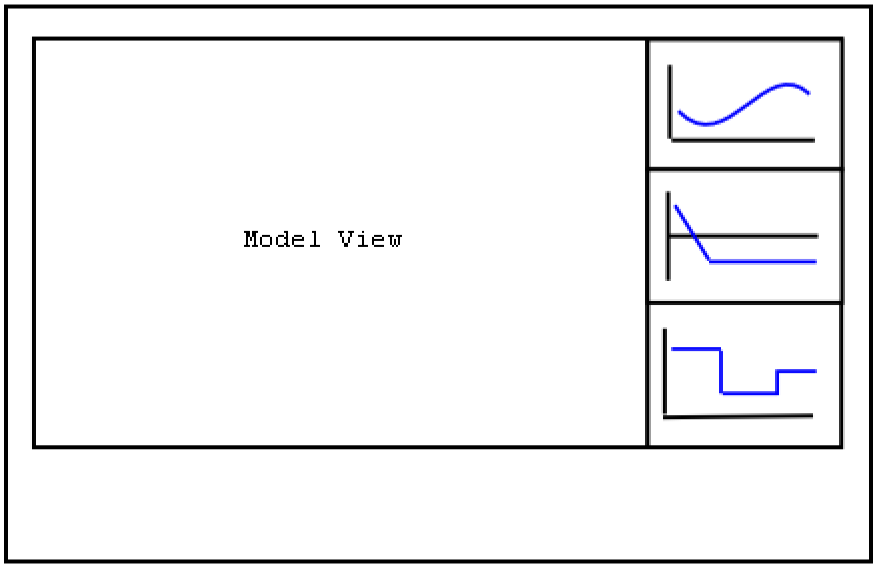


Figure 7-1: Plots aligned by time axis and contained on right side of the model-view area

It would also be beneficial to be able to plot string-valued state variables versus time as step values that are automatically assigned by the program. The step values would be labeled on the y-axis of the plot, as in the following figure.

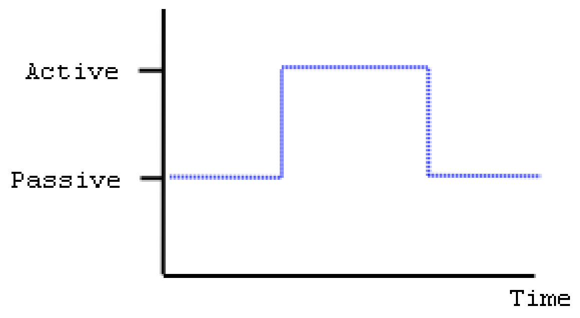


Figure 7-2: Labeled step values for string-valued state variables

The ability to save plots and/or the numerical data they contain for later review and analysis might also prove beneficial.

One issue that might arise in implementing this form of plot support is, how to display a current plot of an as-yet unplotted simulation variable with as much recent history as other, already existing plots. A user running a simulation might decide the value of such a plot at a certain recent point in time might answer questions about the data seen in the already-visible plots. However, since no plot was demanded for the other simulation variable, the program will not have stored any data points for that variable up to the current time in the simulation. It is likely true that for models beyond a certain small size, storing all the data for all the simulation variables at each event slice would be too memory-consuming.

Proposed are two solutions to this problem. One is to allow re-running the simulation up to the current (or a preceding) event in the previous run. The simulation would run to that point with no visualization whatsoever and no delays for real-time simulation. That way, the user could quickly get back to the event of interest in the simulation, with the plot values being stored and plotted for the new plot along with those for the other plots.

Another potential solution would be to store all the data values for any simulation variable for which a plot has been previously requested. This way, even though the first time the user asks for a plot for a particular simulation variable, there will be no history to plot, the data for that variable will be kept from then on, so that for any subsequent request for a plot, recent data will be available for display, without the user being forced to rerun the simulation to that point.

Section 7.2: Model Construction Capabilities

Right now, the only aspect of a model that a user can change through the simulation viewer interface is the layout of the model's components within the model-view. Obviously, a nice capability would be for the user to be able to visually assemble a model within the program from a palette of existing models, and to be able to edit that model's composition subsequently. The program would have to generate and modify Java source files corresponding to the models being created and edited. The benefit of such an approach would be more rapid (and less tedious) model creation and refinement.

Section 7.3: Injection of User-Entered Input Values Into Component Inports

Currently, the list of input values that appears when the user clicks on an inport of a model in the model-view does not include any option for letting the user specify some value not contained in the list. Adding such an option and then allowing the user to specify an arbitrary value into a text-field (as well as a wait-time) would permit far easier experimentation with various stimuli on the inport, saving the user from having to add a new line in the component's source code for each experimental input, compile the class, restart the viewer (to get the new class loaded), and rerun the simulation to the point where it was before. As with the automated plot support enhancement discussed above, giving the user more freedom on what can be done without having to make changes to the model's source code greatly speeds the ability to perform what-if trials on a running simulation.

Section 7.4: Reloading of Model Classes Between Views

As alluded to previously, any changes made to the source code of a model while the viewer is running will not show up in the viewer until it has been restarted, if that model is or has been viewed during the current execution of the program. Therefore, changes to a model's layout, the plots it produces, and the list of input values that may be injected into its inports must all be followed by an annoying and time-consuming program restart. The reason for this is that, to the best of the author's knowledge, there is currently no straightforward way to reload classes within a Java VM. However, the author has come across mention of support for this feature possibly being included in a future release of Java. Taking advantage of any such fruition of this support would likely produce a noticeable time savings for the frequent simulation viewer user.

Section 7.5: Launching a View for a Black-Box Component of a Model

When debugging a model, it would often be nice if there was some way to drill down into a black-box component and see its internal structure and operation, especially when trying to determine whether the problem lies within the component or somewhere else within its parent. Double-clicking on a black-box component could cause a new model-view to appear within a popup window, displaying the component's structure. Furthermore, as the simulation is being run on the parent model, the behavior within the black-box component could be displayed within the new model-view, in sync with the parent behavior in the main viewer window. In some respects, this would be analogous to stepping into a function when running a Java program debugger. The main idea would be to keep the user from having to change the component to no longer be a black box, restart the viewer, observe the component's internal behavior, then undo the change and restart again to restore the component's black-box status.

Likely, no similar motivation exists for purposely hidden components, as they are assumed to be hidden due to their being mundane, tried-and-tested elements whose visibility would only clutter the user's understanding of the model.

.

Section 7.6: Display of Non-Viewable Components

A common use of the simulation viewer is in taking a model class developed by another user and employing the viewer to gain a better understanding of the model's composition and operation. If it is a model class not written with visualization in mind, it is likely that some (or all) of its components will not be *ViewableAtomic*s or *ViewableDigraph*s, and thus will not display within the view. If the model class is complex, it may be a time-consuming task to carefully search through its code to determine what components of what classes it contains, and to update those classes so as to make their components visible.

Just because a component is not derived from *ViewableAtomic* or *ViewableDigraph* does not mean the viewer cannot at least visually represent that component as a plain box with the component name and class information displayed. Couplings to and from the component could also be rendered, although on the component's side of each coupling, there would be no notion of the port to which the coupling is connected (as the component's ports would themselves not be shown). The benefit would be that the presence of the component's name and class in the display would inform the user of its existence in the model, and also quickly let the user know which classes need to be modified to support normal viewing capabilities.

Section 7.7: Allowing the User to Drag the Entire View Using the Mouse

A nice ease-of-use feature of Ptolemy II's Vergil user interface is its ability to allow the user to pan around the view of a model too big to fit in the view window by dragging an outline box around a scaled-down image of the view. This feature would be very time-consuming to implement within the SimView. However, right now the viewer relies on awkward scrollbars to serve a similar purpose. An easy-to-implement compromise might be to allow the user to drag the entire view around within a document by dragging with the mouse within the view when a hand-shaped mouse cursor is visible.

Section 7.8: Letting the User Quickly Run a Simulation up to a Certain Timepoint

There are often times when a simulation has to go through some number of iterations before actually hitting the point of the interest to the human viewer. Using the "run" mode of the viewer to advance past the uninteresting iterations might still incur unnecessary delays for the user, however, as in this mode changes in state values are still updated on the screen, and the user has to watch the progress during each run to make sure the point of interest is not inadvertently passed. It would be much better in this case for the user to be able to simply specify a simulation time value that the simulation execution must reach, and have the simulation run to that point with absolutely no updating of values, or even logging to the console, if possible.

Section 7.9: Indicating Coupling Direction

In DEVS, a coupling between two ports is unidirectional in the way in which messages may travel across it. However, this is not clearly indicated in the SimView's depiction of such couplings. Especially when there are many couplings in the display, it would be visually helpful to the user to indicate each coupling's direction, perhaps either through the use of a certain pattern when drawling the coupling's line, or by placing one or more small arrows along its length.

References

[TMS]	Zeigler, Bernard P., Herbert Praehofer, Tag Gon Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Second Edition*. Pages 66, 69, 129. Academic Press, San Diego, 2000.
[DEVS]	Project coordinator: Professor Bernard Zeigler. Project website: http://www.acims.arizona.edu. University of Arizona.
[Monitor]	Hsu, Irving Shang-Yi. *DEVS Monitor: An X-Window System-Based Debugger for the DEVS-Scheme Simulation Environment*. Master's Thesis, Electrical and Computer Engineering Department, University of Arizona, 1992.
[DP]	Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissdes. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Massachusetts, 1995.
[Ptolemy]	Project coordinator: Professor Edward Lee. Project website: http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm. Department of Electrical Engineering and Computer Science, University of California at Berkeley.
[simjava]	Project implementors: Fred Howell, Ross McNab. Project website: http://www.dcs.ed.ac.uk/home/hase/simjava. Division of Informatics, University of Edinburgh.
[GoldSim]	Product website: http://www.goldsim.com/Tour/Tour0.asp. Golder Associates, Inc.
[JDEVS]	Project implementor: Jean Baptiste Filippi. Project website: http://spe.univ-corse.fr/filippiweb/appli/debut.htm. University of Corsica.
[AweSim]	Product website: http://www.pritsker.com/awesim.htm. Frontstep Corp.
[ATOM3]	Project implementor: Jean-Sebastien Bolduc. Project website: http://moncs.cs.mcgill.ca/people/eposse/devs_in_atom3. School of Computer Science, McGill University, Montreal.
[AnyLogic]	Product website: http://www.xjtek.com/products/anylogic. XJ Technologies Company, Ltd.
[MOOSE]	Project coordinators: R. M. Cubert, P. A. Fishwick. Project website: http://www.cise.ufl.edu/~fishwick/moose.html. Computer and Information Science and Engineering, University of Florida.

[Simscript] Product website: http://www.simprocess.com/products/simscript.cfm. CACI Products Company.

[OMNeT++] Project implementor: András Varga. Project website: http://whale.hit.bme.hu/omnetpp. Department of Telecommunication, Technical University of Budapest.

[OmSim] Project coordinator: Sven Erik Mattsson. Project website: http://www.control.lth.se/~cace/omsim.html. Department of Automatic Control, Lund Institute of Technology.

[CDM] Project coordinator: Professor Hessam Sarjoughian. Project website: http://www.acims.arizona.edu/SOFTWARE/software.shtml. Electrical and Computer Engineering Department, University of Arizona.