A Hierarchical State Machine Editor for Eclipse

By

Michael Gold

Supervised by Dr. Tony White,

Associate Professor, School of Computer Science

COMP 4905: Honours Project

Carleton University

December 2004

Abstract

This project involved the development of an Eclipse plug-in to provide a graphical editor for hierarchical state machines. Its goal is to learn about the design of a modular system based on plug-ins, and to gain experience developing Eclipse plug-ins.

This report describes the project and the theory behind it. First, background material on hierarchical state machines will be presented. Second, Eclipse's plug-in architecture will be described. Next, design and implementation details will be discussed. Finally, the paper will conclude by describing the results achieved and presenting several ideas for future work on this topic.

Acknowledgements

I would like to thank Professor White for his guidance throughout this project.

I would also like to thank Professor Morrison for providing the LaTeX template used for this report.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern computer programs are generally event-driven systems, responding to events such as user input and network traffic. Traditionally, applications have used flag variables and conditional statements to determine the correct handling for an event. In complex programs, this can make the control flow difficult to predict, resulting in buggy software.

A **finite state machine** (FSM) is another way to represent an event-driven system. A set of possible states is defined, and the system must be in one of these states at any given time. Transitions between states are also defined. When an event occurs, the system will look for an appropriate transition based on the current state and the type of event. Finite state machines are commonly represented using diagrams. Given such a diagram, it is easy to predict the system's reaction to various types of events.

In complex systems, the number of states in a finite state machine can grow exponentially, causing the system to become unmanageable. A **hierarchical state machine** (HSM) is an extended version of a finite state machine in which each state can contain multiple sub-states. Similar states can be grouped together in order to share common transitions and code, simplifying the overall design of the system.

The **Eclipse** platform is an Integrated Development Environment (IDE). It is

capable of editing many different types of files, compiling and debugging code for several languages, and integrating with version control systems. Eclipse is written in the Java Programming Language and can be extended by writing plug-ins.

This paper describes the design and implementation of a graphical user interface for editing hierarchical state machines, written as a plug-in for the Eclipse platform.

1.1 Motivation and Objectives

The primary objective of this project was to develop a plug-in for the Eclipse IDE that provides a graphical user interface for creating and editing hierarchical state machines. The ability to simulate events and display the resulting state of the system was also a desired feature.

One personal goal was to apply my academic knowledge and experience to create a useful piece of software. Another goal was to learn about the design of modular systems based on plug-ins, and Eclipse's plug-in architecture in particular.

Chapter 2

Hierarchical State Machines

2.1 Background

The concept of a hierarchical state machine was formalized by David Harel in a 1987 paper called "Statecharts: A Visual Formalism for Complex Systems", as a method of describing complex reactive systems. A **reactive system** is one that is largely event-driven, functioning by responding to stimuli. He notes that there is a major problem in the design of these systems, which is "rooted in the difficulty of describing reactive behaviour in ways that are clear and realistic, and at the same time formal and rigorous, sufficiently so to be amenable to detailed computerized simulation" [Harel, 1987].

Harel stated that while a state machine is a logical model for an event-driven system, a single, flat state machine can become unmanageable when many states are involved. He noted that such a machine will often contain sets of states with similar behaviour, which could be logically grouped into a **superstate**, producing a hierarchical state machine. He combined this idea with the concepts of **orthogonality** (the ability for multiple independent states to be active concurrently) and broadcast communication, calling the result a **statechart**. The scope of this project is limited to hierarchical state machines, rather than statecharts in general.

2.2 ROOM

The **ROOM** (Real-time Object-Oriented Modelling) language was described in detail by Bran Selic, Garth Gullekson, and Paul T. Ward in their 1994 book "Real-Time Object-Oriented Modeling". The authors described "two fundamental drawbacks to using a general-purpose methodology for systems development": the inadequacy of the modelling language, and implementation difficulties. Concepts such as state machines and concurrent processes generally do not map well to traditional programming languages, and are error-prone to implement since the model and its implementation are treated as separate items [Selic et al., 1994].

The ROOM language allows an entire program to be formally defined in graphical form, with enough detail that it can be compiled and executed. A ROOM model describes an application's components, their behaviour, and the manner in which they communicate; a type of hierarchical state machine called a **ROOMchart** is used to describe their behaviour.

The ROOMchart is based on Harel's concept of a statechart. It contains a hierarchy of states, connected by transitions. When a message is received, the component's state is changed by executing an appropriate transition.

2.3 Operation

Like a traditional state machine, a hierarchical state machine is typically displayed in graphical form, with rectangles representing states and arrows representing transitions. States are labelled using names assigned by the user, and transitions are labelled by the events which trigger them. Hierarchy is indicated by drawing states (rectangles) inside other states.

A hierarchical state machine can be thought of as existing entirely within one top-level state, called the **root state**. This state and every other state with substates must have exactly one active substate at any given time. It is also necessary to

designate one substate as the initial state; whenever a state is made active (except through a history transition) the initial substate is also activated, and this continues recursively.

2.3.1 Message Handling

A state machine can receive messages, and generally changes state in response to the messages. Transitions define how the system changes state; each transition is connected directionally to two states, and is triggered by a specific type of message. Incoming messages are delivered to the system's innermost active state, which looks for an outgoing transition triggered by that message. If such a transition is found, it is executed; this means the currently active state becomes inactive, and the state at the other end of the transition becomes active. Note that a state's parent must be active before the state can be activated, so state changes in higher parts of the hierarchy may happen while executing a transition; likewise, deactivating a state requires all substates to be deactivated first.

Any time a state lacks an appropriate transition to handle a given message, it passes the message to its parent state, which uses the same algorithm for find and executing a transition. If the message reaches the root state and no appropriate transition is found, the message is dropped and no state changes occur.

Harel defined two types of history that can be applied to a transition. The symbol "H" represents a top-level history transition; when this transition is taken, the target state activates its previously active substate rather than its initial state, and all substates of that activate their initial substates. Another symbol, "H*", represents a full-history transition; the target state's previously active substate is activated, then that state's previous substate is activated, and this continues recursively [Harel, 1987].

2.3.2 Executing Code

Implementations of hierarchical state machines typically allow the user to attach executable application code to various events, in common languages like Java and C++. Types of code available usually include entry code, executed when a state is activated, exit code, executed when a state is deactivated, and transition code, executed when a transition is followed. Each piece of code is attached to a specific state or transition.

2.4 Previous Implementations

While there have been many implementations of finite state machines, only a small number of these have implemented hierarchy. One of the best-known implementations of a hierarchical state machine editor is Rational Rose RealTime, a software modelling tool. Rose uses UML[1] diagrams to define objects, and ROOM to describe many other aspects of the system, including its behaviour.

Hierarchical state machines are a major part of Rose. Each component in the system has an HSM associated with it, which receives messages over communication links. C++ or Java code can be specified to be executed during a transition or while entering or exiting a state. The user defines the types of messages a state machine can send and receive by creating **protocols** and associating a set of protocols with each state machine.

Rose provides a graphical editor for creating and editing hierarchical state machines. It also allows the system to be debugged by injecting messages into a state machine and displaying the result.

[1] unified modelling language

Chapter 3

Eclipse Plug-in Architecture

Plug-ins are central to the design of the Eclipse integrated development environment; almost every feature in a default installation is provided by a plug-in, seamlessly integrated with the rest of Eclipse. As a result of this design, it is easily extensible by developing new plug-ins [Gallardo, 2002].

An Eclipse plug-in is defined by an XML[1] file. This file describes general properties such as the name and version of the plug-in, as well as a list of libraries and other plug-ins required. Extension points and extensions are also defined in this file.

3.1 Extensions

An **extension point** is an interface defined by a plug-in which allows other plug-ins to extend it. For example, Eclipse's default plug-ins define extension points which allow other plug-ins to create new types of editor windows, make file-creation wizards available, and add menu items to Eclipse's main menu.

A plug-in builds on an existing extension point by defining an **extension**. This is defined in the XML file by providing the name of the extension point, and including a plug-in-specific piece of XML containing any required parameters. A class name is

[1] extensible markup language

often included, which enables the extension point provide a more powerful interface by dynamically loading the given class.

3.2 Plug-in Development Environment

Eclipse includes a Plug-in Development Environment (PDE), which provides several extensions related to plug-in development.

A plug-in creation wizard is provided, which generates a new plug-in project and provides code for various features. For example, sample code is provided for creating multi-page editors, property pages, and file-creation wizards. Other PDE features include a graphical editor for plug-in XML files, and special views and perspectives.

3.3 Graphical Editing Framework

The **Graphical Editing Framework** (GEF) was used to develop the Eclipse plug-in for this project, and is a plug-in that makes it possible to develop graphical editors that have the same look and feel as the other components of Eclipse.

GEF hides most of the low-level implementation details of creating an Eclipse plug-in. The main pieces of code which work directly with Eclipse are for displaying the new file wizard, creating the console window, and adding menu and toolbar items; the remainder of the code deals primarily with GEF. These interactions are described in the next chapter.

Chapter 4

Design and Implementation Details

4.1 Model Design

Since every level of a hierarchical state machine behaves in the same way, the entire state machine can be contained within a single root state; this facilitates code reuse by eliminating the top level as a special case.

A `State` object is defined to describe each state in the system. Important fields include a reference to the parent state, a list of children, and lists of incoming and outgoing transitions. A specific reference to the initial substate is included, which must be included in the list of children. Properties such as entry code, exit code, and name are also stored.

Simulation logic is contained within the `HsmSimulator` class rather than `State`. However, several simulation-related fields were added to `State`; although these are not strictly related to the structure of the state machine, they were defined here to simplify development. Data related to the visual representation of the state machine, such as the state's location and size, was included here for the same reason.

Each transition in the system is defined by a `Transition` object, which stores the transition's trigger along with a start state and end state. The history type and transition code are also stored.

4.2 Creating the Editor

4.2.1 `HsmEditor` Class

An editor is made available by extending the `org.eclipse.ui.editors` extension point, and providing the name of the editor's main class. In this case, the main class is `HsmEditor`. It extends from GEF's `GraphicalEditorWithPalette`, which provides an editing pane and a configurable tool palette.

`HsmEditor` initializes itself by instantiating several other objects, such as the simulator, the model, and the tool palette. It is also responsible for saving and loading files, and for creating the console window when the standard output or error streams are requested.

4.2.2 User Interface

The model-view-controller design pattern is used to create the user interface when working with GEF.

Model

The editor's model is represented by the `State` and `Transition` classes, as described above. `HsmEditor` stores a reference to the root `State` object, and each state stores a set of substates (also `State` objects) as well as a set of `Transition` objects.

View

GEF depends on the Draw2D package for its display [Moore et al., 2004]. The view is implemented by extending Draw2D's `Figure` class. The `StateFigure` object extends `Figure` directly, adding a scrollpane, a title bar, and a border. Various methods are defined to change properties such as the state name; the scrollpane can contain other `StateFigure`s, and is managed automatically by GEF. `TransitionFigure` is a

simple extension of the `PolylineConnection` class (a subclass of `Figure` defined by Draw2D), adding an arrowhead and a label to the line segment.

Controller

Classes known as `EditPart`s act as controllers, integrating the model with GEF and Draw2D. Each EditPart stores a reference to its model object, and creates its view object when the `createFigure` method is called. EditParts also provide properties by implementing the `IPropertySource` interface; Eclipse will automatically make these properties available in its property editor.

Each EditPart registers as a listener with its model object by implementing the `ModelListener` interface and calling the model object's `addListener` method. The model will then notify the EditPart of property changes by calling its `modelChanged` method, passing a `ModelEvent` object as a parameter; the EditPart will respond by updating the view.

4.2.3 Serialization

Java's serialization support is used to allow state machines to be saved and reloaded. This simplified development since it was not necessary to design a new file format and write code to work with it.

Each model class (`State` and `Transition`) implements Java's `Serializable` interface, informing Java that it can be serialized. Since all data is contained within one root `State`, saving is performed by writing this object to an `ObjectOutputStream`. Similarly, a file is loaded by reading the root object from an `ObjectInputStream`.

Since each model object references EditParts using the `ModelListener` interface, Java would normally attempt to write the EditParts to the stream. This is not possible since they are not serializable, and they can't be made serializable because they reference non-serializable GEF classes. The list of listeners was declared as `transient` to prevent this situation. A `readObject` method which re-initializes any

transient variables was added to each class; Java automatically calls this method when reading the object from a stream. Several variables related to to the HSM simulation are also marked as transient because it would be unnecessary to save and reload them.

By default, changing any of the serializable classes will make them incompatible with previously serialized files. The static variable `serialVersionUID` was defined for each model object to prevent this; as long as this variable remains the same, Java will allow serialized files to be loaded. Java's `serialver` command was used to determine the proper value of this variable for each class.

4.2.4 New File Wizard

Eclipse automatically created a "new file" wizard when the editor plug-in was first created. This wizard lets the user choose a filename and directory; then it creates an empty file and loads it in an editor window. The wizard was included in the project with minor modifications.

Attempting to load an empty file from an `ObjectInputStream` will result in an `EOFException`. The file-loading code in `HsmEditor` was modified to create a new model when it catches this exception. This approach was chosen because it was simpler than the alternative of adding serialization code to the wizard and having it write a valid root state to the file.

4.2.5 Command Pattern

The **Command** design pattern is used in GEF to encapsulate any changes to the model, and enables undo and redo functionality to be implemented easily.

Every possible command is represented by a different class which extends the GEF `Command` class. A `Command` object is instantiated whenever a modification of the model is requested. Any necessary parameters, such as the model object to be modified, are provided to the `Command`'s constructor.

A `Command` object has three required methods. `run` executes the command, modifying the model in an appropriate manner. `undo` reverses the effect of a previous `run` invocation. In some cases, it is necessary to save some information before modifying the model so that `undo` will be able to reverse the changes. The `redo` method re-runs the action after `undo` has been used, and is often the same as `run`.

Since each change to the model is encapsulated in a `Command` object, it is trivial to implement undo functionality by storing each object in a list after its `run` method has been called. Performing the undo action consists of removing the most recent item from the list and running its `undo` method. Redo can be implemented in a similar manner, provided the `Command` objects are stored after performing an undo. GEF automatically implements this functionality, and makes the "Undo" and "Redo" options available in the **Edit** menu of Eclipse.

4.2.6 Edit Policies

When the user tries to modify the model, GEF looks for an appropriate `EditPolicy` object. Edit policies define what can be done with `EditPart`s [Moore et al., 2004]. Different subclasses of `EditPolicy` exist, each with a different set of abstract methods which must be implemented. These methods generally take a `Request` object as a parameter and return a `Command` object. `Request` is a class used to describe a desired model change.

The state and transition `EditPart`s both register a `ComponentEditPolicy`, containing only a `createDeleteCommand` method. GEF calls this method when the delete key is pressed, and executes the returned `Command` to delete the selected object.

`StateEditPart` also registers a `GraphicalNodeEditPolicy`, which deals with the creation and editing of transitions, and an `XYLayoutEditPolicy` which is used for moving and resizing states, as well as creating substates.

4.2.7 Properties

An `EditPart` can display its properties in Eclipse's property window by implementing the `IPropertySource` interface. Methods for retrieving, setting, and resetting these properties need to be implemented.

The `getPropertyDescriptors` method also needs to be implemented, and must return a list of property descriptors that describe which properties are available and how to display or edit them. Standard property descriptors are available that provide text fields or dropdown lists for editing.

A class called `CodePropertyDescriptor` was created to allow code to be edited. This is a trivial class which simply returns a `CodeCellEditor` when the property needs to be edited.

`CodeCellEditor` is an extension of `DialogCellEditor`, a class which displays a button beside each property and opens a separate property editor when the button is pressed. A `CodeInputDialog` is used for editing. This is a custom extension of Eclipse's standard `InputDialog`, but displays a multi-line text box suitable for editing code.

4.2.8 Menus

Several simulation-related actions needed to be made available to the user, and the menu bar was determined to be an appropriate location for these actions.

A `contributorClass` parameter was added to the `org.eclipse.ui.editors` extension to provide the menu. A class extending from `ActionBarContributor` was created. A `buildActions` method was defined which creates several `Action` objects and adds them to the editor's list of available actions.

A `contributeToMenu` method creates a new menu, adds several `Action` objects after retrieving them from the editor's list, and inserts the menu into Eclipse's menu bar at a specific location. A separate `contributeToToolBar` method makes the Undo, Redo, and Delete actions available.

Some actions require a reference to the active `Editor` object. The `setEditorPart` method is automatically called when the active editor changes, and simply calls the `setEditorPart` method of the `Action` objects, providing a reference to the new editor as a parameter.

Context Menu

A state can be set as its parent's initial state by right-clicking on it and selecting an option from the context menu that appears. This was implemented by creating a class which extends `ContextMenuProvider`, and defining a `buildContextMenu` method which adds `Action` objects to a given menu.

`HsmEditor` instantiates a `ContextMenuProvider` object while initially configuring the editor. It then registers this context menu provider with Eclipse and GEF.

4.2.9 Console Window

A console window was necessary to display output from the HSM simulator, including status output, errors, and messages printed by user-defined code. Eclipse automatically provides such a window when an external process is launched, but the version used doesn't allow plug-ins to create their own consoles without launching a program.

A `SimulatedProcess` class was created to solve this problem. This object implements Eclipse's `IProcess` interface, and is designed to make Eclipse think an external process has been launched. It provides two `Writer` objects, for writing to the output and error streams; anything written to these streams is returned as the output of the virtual process.

Some configuration is needed to inform Eclipse of this simulated process. The HSM editor adds a new `Launch` object to the `LaunchManager` of Eclipse's debug plug-in during initialization. When a console window is needed, a new `SimulatedProcess` object is created and added to the `Launch`. Eclipse will then display a console as if an external application was launched. Any text written to one of the `Writer` objects

Table 4.1: Variables exported to Jython code

Variable	Description
`storage`	a dictionary object accessible from all Jython code, which can be used to store global data
`message`	the name of the message triggering this state change
`parameter`	a string parameter associated with the message
`transition`	a reference to the `Transition` being executed
`state`	a reference to the `State` whose code is being executed (or `None` when executing transition code)

will be displayed on the console; text from the standard error stream is highlighted by displaying in red.

4.3 Jython Integration

Since transitions and states can have code associated with them, it was necessary to embed a programming language into the simulator. The language chosen was **Jython**, an implementation of the Python programming language in Java. It was chosen primarily because it is easy to integrate with Java code.

A JAR file must be added to the HSM plug-in's classpath to make the Jython classes available. A Python interpreter can then be created when required, by instantiating the `PythonInterpreter` class.

Java variables are made available to Python code using the `PythonInterpreter`'s `set` method, which takes a variable name and value; Jython automatically makes the variable's public methods available to the interpreted code. Table 4.1 lists the available variables.

Python's `print` statement can be used to output text to the standard output or standard error streams. These streams are redirected to the editor's console by calling the interpreter's `setOut` and `setErr` methods, providing the console's `Writer` objects as parameters. This allows the state machine code to easily output messages to the

user.

Python code is executed by creating a new `PythonInterpreter` instance and calling its `exec` method, providing the code as a `String` argument. A `PyException` error is thrown if any errors occur while interpreting or running the code; the simulator handles this by displaying the error in the console.

To allow data to be shared between all the code segments that run during a simulation, a Python dictionary object is created by the simulator. This stores key-value pairs, similar to a Java `Hashtable` object, and is always available from Python code as the `storage` variable. Note that since a function can be assigned to a variable in Python, it would be possible to define global functions in `storage` and call them from other code.

4.4 State Machine Simulation

Once a hierarchical state machine has been created, its behaviour can be tested by simulating messages. An `HsmSimulator` object was created to implement this functionality, rather than adding code to the `State` and `Transition` classes. This was done to keep the system's behaviour separate from its data, which makes the code easier to maintain. It also means the simulation state is not saved when serializing the model, which is the desired behaviour.

The `HsmSimulator` object is created by `HsmEditor` when `setModel` is called, which is done whenever a new model is created or an existing one is loaded. A reference to `HsmEditor` is provided to the constructor, and the simulator stores this reference.

4.4.1 Model Fields

State-specific simulation data is stored directly in transient fields of `State` objects. One field contains a reference to the active substate, and is used for the simulation logic. This field is not modified when a state becomes inactive, so it can be used

while performing a history transition to find the last active substate.

Another value indicates whether the state is active, inactive, or inactive but saved for history; this value only affects the visual representation of the state, and was included to simplify the model-view-controller interactions (its value could have been calculated by looking at the active substate of each higher-level state).

4.4.2 Starting the Simulation

The simulation is started by calling the `HsmSimulator`'s `start` method. As well as initializing variables related to the simulation, it calls the root `State` object's `reset` method. This recursively goes through the state machine, and sets each state's initial substate as its history state. All other states are marked as inactive.

The state machine is then started by activating the initial states, descending from the root to the deepest substate. Each state's entry code is executed when the state is activated, as expected. This allows the root state's entry code to be used for global initialization code, as it is always run before any other code. Similarly, the root's exit code can be used as cleanup code, as it is always the last code run before stopping the state machine.

4.4.3 Simulating Messages

The simulator's `processMessage` method takes a message name and a parameter, both as `String`s. This causes the simulator to look for an appropriate transition, and change state if one is found.

Finding the correct transition

The root state's `getInnermostActiveState` method is called to determine the currently active state. This method follows the path of active states, descending from the root to the lowest active state with no substates.

That state's `getTransitionForMessage` is called, to check if an appropriate transition is defined. If not, the simulator calls the method on the parent of that state. If no transition is found at the root, an error is printed and the message is discarded. Otherwise, the first transition found is noted as the one that will be taken when changing state.

Building a path to the new node

Once the correct transition has been identified, transitioning will involve exiting the current state and entering the new state. However, it may also be necessary to exit and enter some of their parent states. To perform the state change, a path will be created from the current state to the new one, listing the states that need to be exited followed by the states that need to be entered. Note that the current state might not be an endpoint of the transition; the transition could start at any of the parent states.

The most obvious method to generate this path would be use the paths from both states to the root, but this would cause certain states to be exited and entered again if the current and new states shared any common parents. To prevent these redundant state changes, two paths are first constructed: one path leads from the current state up to the root; the other goes from the root down to the new state. Then redundant states are repeatedly removed from the lists: if the last state in the up-list is equal to the first state in the down-list, both are removed.

If the new state has substates, the down-list will need to be extended deeper. Depending on the transition's history mode, this will consist of choosing either initial substates, substates that have been saved for history, or a single saved state followed by initial states. The appropriate states are determined and added to the list.

The up-list and down-list are then concatenated, separated by a `null` value which represents the transition, and the new list is stored. This representation makes it easy to execute one step at a time, which will be useful when animating the transition.

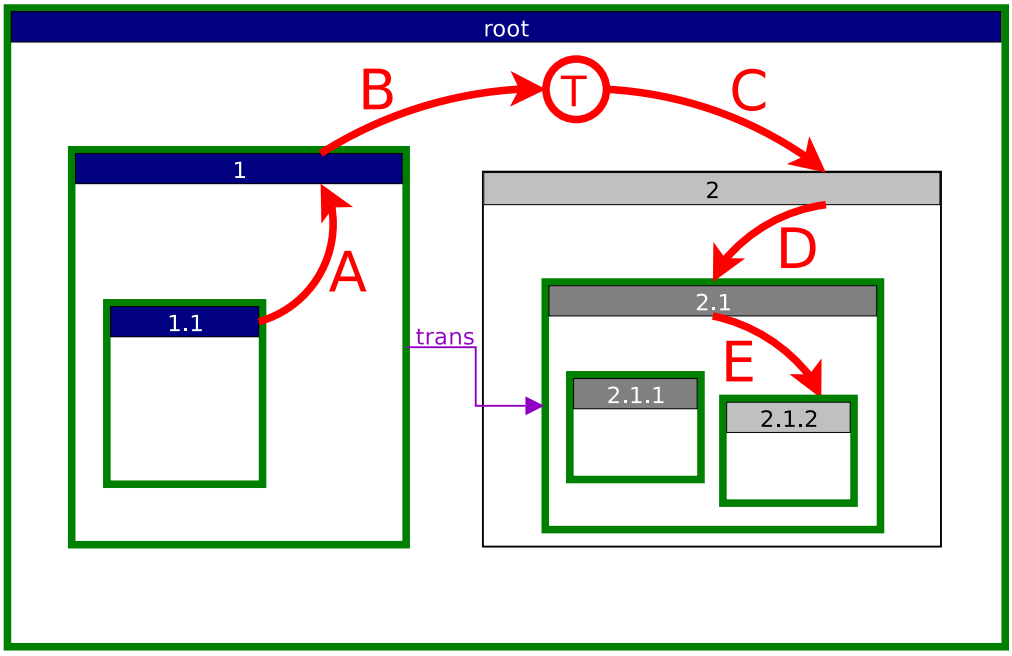

Figure 4.1: Executing a transition

Changing state

Changing state is a simple process once the transition and paths have been found. First, each state in the up-list is exited, in order: the state is marked as inactive but saved for history, and its exit code is executed. Next the transition code is executed. Finally each state in the down-list is marked as active, in order, and has its entry code executed.

While these steps modify the model, they do not use the **Command** pattern to do so. This was done because it is unlikely the user would want to undo or redo a single step of the state change, and allowing them to do this might leave the system in an invalid state. Another reason is that only the `State` object's `transient` variables are changed, so none of the modifications regarding active or inactive states will be saved.

Figure 4.1 shows an example of this process, in which state 1.1 is active and the transition labelled "trans" has been chosen. While the transition starts at state 1,

this state has an active substate which must be exited first (shown as step A); next, state 1 is exited (step B). The root state does not need to be exited, since it will remain active after the transition (as it is a common parent of both 1 and 2.1). Since we are done exiting states, the transition runs at point T. Steps C and D show states 2 and 2.1 becoming active. The transition's end state is now active; however, it has a child state, so the process continues. The initial substate 2.1.2 is activated in step E, since the transition did not specify history (note that 2.1.1 would have been chosen in a history transition, as indicated by the state's dark title bar). The transition stops since this state has no substates.

Transition animation

To allow the user to see what is happening, the transition is performed one step at a time. A thread is created which alternately sleeps and calls the simulator's `step` method; the `step` method is responsible for performing the next step of the transition. When the transition is complete, the thread exits.

Sometimes it is necessary to stop the timer and finish the state change immediately; for example, to get the state machine into a consistent state before injecting another message. The thread is stopped by calling the thread's `interrupt` method, which will cause it to catch an `InterruptedException` and exit. To complete the state change, the `step` method is called repeatedly until no more steps remain.

Chapter 5

Conclusion

5.1 Results Achieved

The project's primary objective of developing a graphical HSM editor as an Eclipse plug-in has been achieved. The editor allows the user to design a hierarchical state machine consisting of states, transitions, and associated Python code. Figure 5.1 shows a screenshot of the editor, along with the simulation console and property windows.

The operation of the state machine can be simulated by injecting messages, and the resulting state changes will be displayed. One desired feature was the ability to inject a series of messages based on a script. While time constraints prevented this feature from being added, the existing simulation does work as expected when responding to manually-entered messages. Python proved to be a good choice for an embedded programming language, as the code is easy to understand and the Jython interpreter was simple to integrate.

The personal goals for the project have also been achieved. Academic knowledge has been applied to create a useful piece of software. Experience and knowledge have been gained in the design of Eclipse plug-ins and other modular systems, and the design patterns that are used to implement these systems.

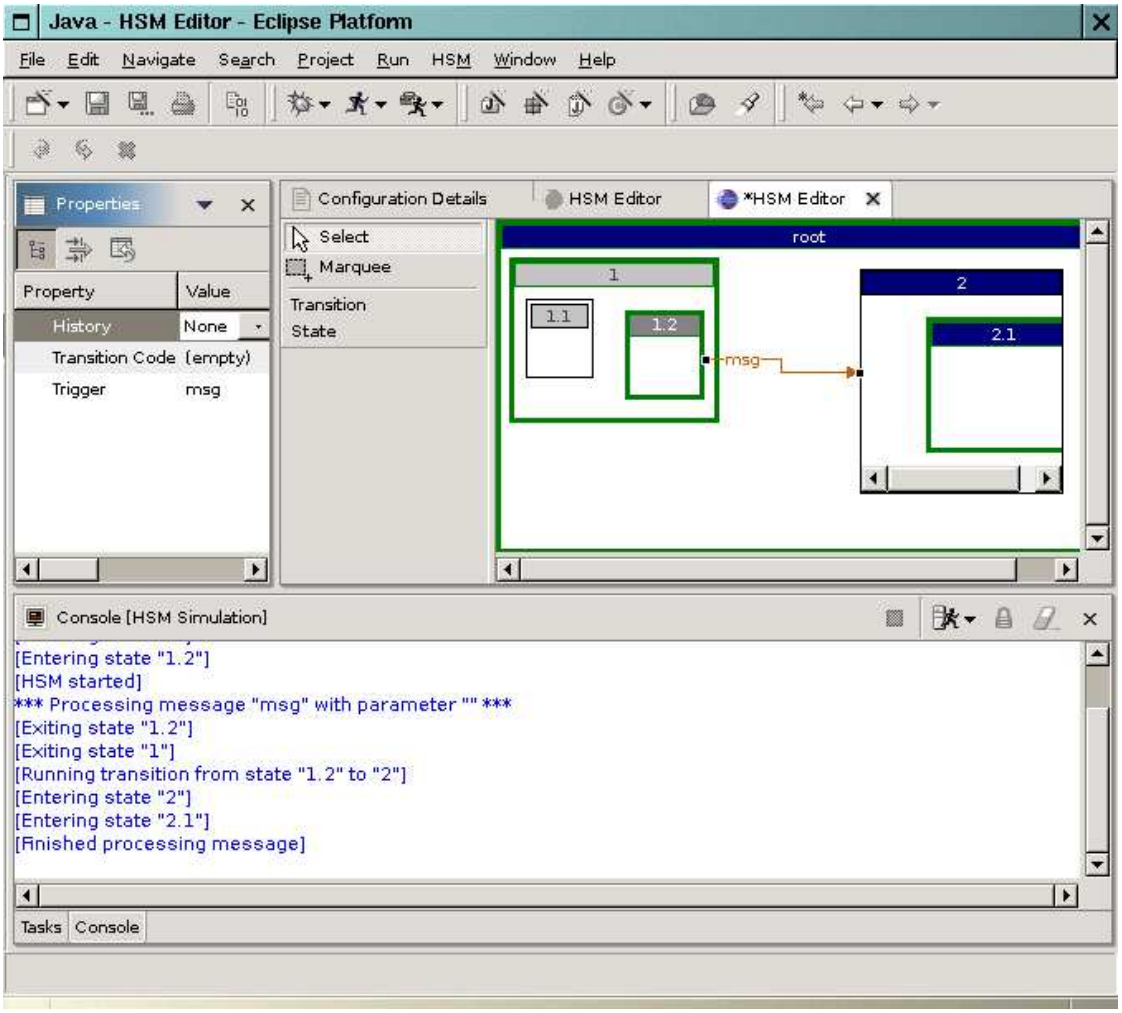

Figure 5.1: HSM Editor in simulation mode

5.2 Future Work

While the project works as expected, it is not well-suited to application development in its current form. One possible extension would be the ability to export a state machine to Java code, allowing it to be easily integrated with existing applications and run outside of the Eclipse environment.

Further work could be done based on the ideas presented in Harel's paper. For example, the project could be extended to support orthogonal states or conditional transitions; support for initial transitions could be added to replace the current method of specifying initial substates. Harel also describes several possible extensions to the statechart formalism, such as overlapping states, which may be worthy of further research [Harel, 1987].

Other ideas for future development can be taken from ROOM [Selic et al., 1994]. The project could be extended to support several components communicating through user-defined protocols, or to split transitions into segments when they cross state boundaries. Another possible extension would be the ability to define actors, data classes, and other object-oriented constructs defined by the ROOM language.

The ability to read a series of messages from a file and inject them into the system would be a useful feature during testing. Allowing Python code to send messages may also be useful. Several graphical improvements are possible as well, such as the ability to zoom into substates, or to bend transition arrows to make their labels more visible.

Bibliography

[Gallardo, 2002] Gallardo, D. (2002). Developing Eclipse plug-ins. `http://www-106.ibm.com/developerworks/opensource/library/os-ecplug/`.

[Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

[Moore et al., 2004] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., and Vanderheyden, P. (2004). *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. International Business Machines Corporation.

[Selic et al., 1994] Selic, B., Gullekson, G., and Ward, P. T. (1994). *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc.

Appendix A

System Usage

A.1 Running the Plug-in

The plug-in can be started by loading the project and selecting the **Run⇒Run As⇒Run-time Workbench** action from Eclipse's main menu. This will load another Eclipse instance in which the HSM plug-in is available.

A.2 Creating a New Project

A project must be created before creating a state machine. To do this, select **File⇒New⇒Project** from the main menu. Select **Simple** from the left list, and **Project** from the right list. Type a name for the project and click **Finish**.

A.3 Creating a Blank State Machine

To create a blank state machine, select **File⇒New⇒Other** from the menu, **95.495** from the left list, and **Hierarchical State Machine** from the right list. Use the **Browse** button to select a project which will contain the new file, and enter a filename for it, ending in the ".hsm" extension. Click the **Finish** button.

A.4 Editing a State Machine

To add a new state, click the **State** button on the palette to the left of the state machine editor; then use the mouse to draw a state on the editor by pressing and holding down the left button, moving the mouse to select the size, and releasing the button. A substate can be created by drawing within an existing state. To designate a state as its parent's initial state, right-click on it and choose **Set as initial state**.

To create a transition, click the **Transition** button on the palette, click the desired start state, and finally click the end state.

The **Select** tool is used to select a state or transition. A selected object can be deleted by pressing the **Delete** key on the keyboard, and a selected state can be resized by dragging its edges with the mouse.

Actions can be undone using the **Edit⇒Undo** menu item; **Edit⇒Redo** should also work as expected.

Properties for a selected object will be displayed in the properties window. If this window is not shown, select **Window⇒Show View⇒Other...** from Eclipse's menu, then choose **Basic⇒Properties** and click **OK**.

A.4.1 State Properties

A state's **Name** property sets a label for the state; it is only used for display purposes. The text can be edited in place by clicking on the **Name** row in the property list and typing into the text box that appears.

Entry Code and **Exit Code** specify Python code to be run when the state is entered and exited respectively. To edit it, click on the row, then on the **...** button that appears on the right; enter Python code into the dialog and press the OK button. The root state's entry and exit code can be used for application startup and shutdown code. Table 4.1 lists the variables accessible by Python code.

A.4.2 Transition Properties

Trigger is the most important property for a transition; it is a string defining the type of message which will trigger this transition, and is displayed as the transition's label.

Transition Code is Python code which will be run when the transition is taken.

History defines the type of history that will be used when entering the target state. The default value, **None**, indicates the initial substate will become active. **Top-level** (indicated by adding [**H**] to the transition's label) means the previously active substate will be chosen, and initial substates will be activated below this level. **Full** (indicated by [**H***]) means that the previously active substate is chosen at each hierarchy level.

A.5 Simulation

A.5.1 Visual Representation

A state displayed with a thick green border indicates it is the initial substate of its parent.

A blue title bar indicates an active state, and grey indicates an inactive one. An inactive state saved for history (i.e. the parent's previously active substate) is indicated with a darker grey title bar.

A.5.2 Starting and Stopping

To start the simulation, choose **HSM⇒Start HSM** from the menu bar. This will activate the initial states, running their entry code as needed.

Similarly, the **HSM⇒Stop HSM** menu item will stop the simulation and deactivate all active states, running their exit code.

A.5.3 Simulating a Message

Choose **HSM⇒Simulate Message** from the menu to inject a message into the state machine. Enter the message type, which should match the **Trigger** property of one of the transitions, and an optional parameter, which is ignored by the system but passed to any Python scripts run in response to this message.

State changes are done in three states: exiting states, running the transition, and entering states. The state change is performed slowly to let the user see what is happening; the delay between steps can be adjusted by choosing **HSM⇒Set Simulation Timing** from the menu and entering a new value in milliseconds.

A console window should appear the first time the state machine is started. This window logs every action taken, and every state change which happens in response. It also displays the output of any Python code, and exceptions thrown by the Python code or interpreter.