Design of a Zooming Viewer for Statecharts

Roger Bernier

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements
For the Degree of Master of Computer Science at
Concordia University,
Montreal, Quebec, Canada

December 2000

0-612-59317-7

ABSTRACT

DESIGN OF A ZOOMING VIEWER FOR STATECHARTS

Roger Bernier, M.Comp.Sc.
Concordia University, 2000

Traditional state-transition diagrams have been inherently flat in nature. With the advent of hierarchical state diagrams, known as *statecharts* [HAR87], there has been a marked improvement in the way the dynamics of a system can be modelled. As systems become larger and more complex, statecharts been used successfully to reduce the complexity. However, the current lack of tools that take advantage of this notion of hierarchy have hindered the development process. It is sometimes desirable to hide or display certain portions of a statechart. For example, a composite state may be collapsed or expanded depending on the zooming level. Thus, the aim of this thesis is to design and implement a 'zooming' viewer for statecharts. This goal is to provide the architect or system analyst with an easy and intuitive way to navigate a large statechart diagram. In so doing, the system should be easier to view and understand.

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Dr. Greg Butler for his assistance in the preparation of this manuscript. Also thanks to Dr. Ben Bederson of the University of Maryland who provided the Jazz framework used in this thesis and to his many useful comments. In addition, thanks to the many helpful comments of my fellow graduate students at Concordia. Special thanks to my wife Irene who gave me her much needed support during this undertaking.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

OVERVIEW

1.1 Introduction

This thesis concerns the design and implementation a zoomable viewer for Unified Modeling Language (UML) [RAT00] *statecharts*. A UML statechart is a state transition diagram that provides the software architect with a powerful and intuitive way of expressing the dynamic behaviour of an object or an interaction. Based on Harel's visual formalism [HAR87], UML statecharts provide additional semantics for displaying hierarchical object-oriented statecharts. A zoomable viewer is a software tool that enables a statechart to be viewed and navigated at various levels of detail. This section provides an overview of the problem, motivation, and solution in order to provide a framework for understanding later chapters.

1.2 The Problem

Information visualization is an important area of research in Human Computer Interaction (HCI). The problem of information visualization is well documented in the literature and on the Internet [OLIV00] and involves limitations on memory load and user capacity, the complexity of the information, and platform constraints and capability. The inherent problems of navigating a large information space, at various levels of detail, using multiple-representation of data, and zooming techniques [BMG00] are especially relevant when navigating a large statechart diagram.

The problem is that as a statechart diagram grows in size or complexity, it often becomes difficult to read, understand, and maintain on a single screen space. This results in an increase in the cognitive load that a user must endure while navigating the diagram. From a user's perspective, it is desirable to customize the environment in order to exploit the multiple representations of statecharts. From a designer's perspective, the problem is threefold: 1) deciding *what* information of the statechart to represent at a given level of detail (LOD), 2) *when* to represent it, and 3) *how* to represent it.

'What' information to represent depends on the context that is present at a given level of detail. For example, as a user zooms in, more details (such as a state's name) should be presented. 'When' to present the information depends on the policies set by the program and manipulated by the user. It depends on the scale or size of the statechart diagram components at a given instance. 'How' to represent the statechart has more to do with the graphic framework and the layout tools available to the software architect.

1.3 The Motivation

UML is a recently adopted modeling language standard in the software industry. In the last few years, many tool builders have embraced this new standard and have created various Computer-Aided Software Engineering (CASE) tools to support UML. However, in the case of statecharts containing hundreds of embedded states and transitions, there is a lack of tool support for managing large UML statechart diagrams. Most tools lack support for zooming and navigation. Thus, there is a real need for specialized tools such as the one presented in this paper to complement commercially available tools. Additionally, many such tools only implement only a subset of what a statechart viewer can represent.

1.4 Overview of the solution

With the advent of eXtensible Markup Language (XML) [W3C00] in the Internet community, a new standard called Extensible Model Interchange (XMI) [OMG99c] has become available for exchanging UML models. The purpose is to allow UML models to be exchanged between tool vendors.

The solution proposed is to create a statechart in a commercial tool such as Rational Rose and to export it into an XMI file containing model and view information. This information can be read by a Java program which is capable of reading and interpreting the XMI file, re-constructing the model and view information, and representing it as a *scenegraph* in a 2D graphics framework (called Jazz [JAZZ00]). Once in a Jazz scenegraph, the user can easily navigate the scene using basic GUI interaction techniques (i.e., keyboard and mouse). A large statechart diagram can be navigated using panning and zooming, complex states can be hidden/viewed depending on the level of detail, and various views can be presented to the user.

1.5 Overview of the conclusions

According to Ware, 'One of the greatest benefits of data visualisation is the sheer quantity of information that can be rapidly interpreted if it is presented well' [WARE00, p. 1]. The ability to easily navigate a large information space has wide applications in many areas of computer science and other domains. As systems continue to grow, it becomes important to manage this ever-increasing complexity. One such technique involves zooming while navigating a statechart diagram. The author has demonstrated that by providing a zoomable interface for statecharts, the system becomes more comprehensible.

1.6 Layout of the Thesis

Chapter 1 serves as introduction to the thesis. Chapter 2 covers the details of what a statechart is and what it kind of information it contains. The goal is to understand what a statechart is before trying to design the system. Chapter 3 contains background material concerning the state of current technology. The aim is to select the appropriate tools to design the statechart viewer based on the technology available at the present time. The Jazz framework in Chapter 4 provides a key component to the architecture of the system. Chapter 5 provides the high-level architecture the system, whose aim is to understand the system from a global perspective. Chapter 6 contains a detailed view of the architecture and covers the design issues involved during the development and implementation of the statechart viewer. Chapter 7 discusses what has been found and draws some conclusions. Appendix A shows some sample screenshots of the statechart viewer. Finally, there is a glossary of key terms used in this thesis.

Chapter 2

STATECHARTS

2.1 Introduction

This chapter reviews the basic terminology of statecharts. It then focuses on the hierarchical issues involving nested states and transitions. The ability of a state to contain other states is the key to simplifying the modeling of complex behaviours. The goal is to provide the context for the design of a zoomable viewer for statecharts presented in the next chapters. Several examples of the desired features of statecharts will be presented in this section along with their major concepts.

2.2 Background

Statecharts were first introduced by Harel as a means of modeling the complex reactive behaviour of a system [HAR87]. Reactive behaviour signifies event-driven behaviour (common in Graphical User Interfaces (GUIs)). According to Harel [HAR87], statecharts represented a major improvement over traditional finite state machines (FSM). Traditional state diagrams were 'flat', providing no notion of depth, hierarchy, or modularity. If an event caused the same transition to occur from a large number of states, such as a high-level interrupt, state transitions were required to be attached to each resulting in an unnecessary multitude of arrows. As a system grows linearly, the number of states grows exponentially, since each state must be explicitly shown. Finally, such state diagrams were inherently sequential in nature. Concurrency is not easily modeled.

Statecharts provide several improvements over traditional state transition diagrams. Hierarchy, nesting, sequential and concurrent substates are but some of the advanced features of statecharts.

As an example, Figure 1 shows a traditional finite state machine with two transitions leading from State A and State B. Figure 2 shows a semantically equivalent representation while reducing the number of transitions (arrows) by creating a composite state D. This improves the clarity of the model and simplifies the behaviour.

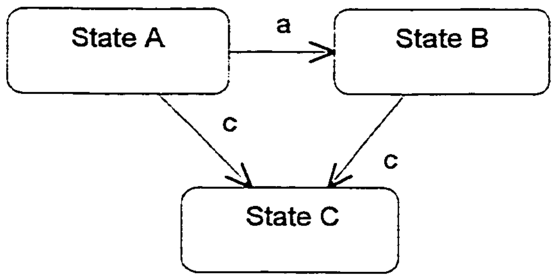


Figure 1: A Traditional State Transition Diagram

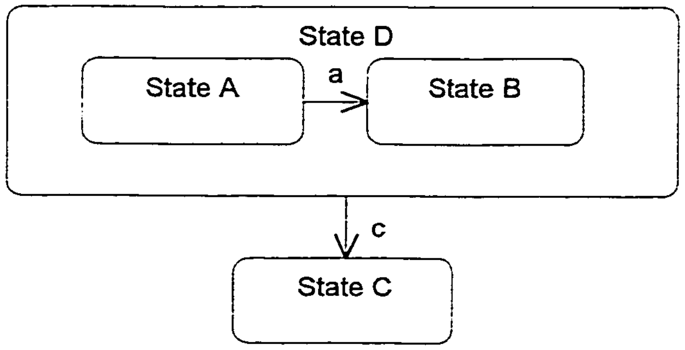


Figure 2: A Statechart Diagram

2.3 Events

An event is the specification of a significant occurrence that has a location in time and space [BRJ99]. There are several kinds of events that can be modelled with UML. *Signal events* represent a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. For example, a 'Printer Failure' signal event causes a message to be sent from the printer to the terminal. *Call events* represent the dispatch of an operation (usually a method). The difference with a signal event is that a call event is normally synchronous. Modeling signal events and call events are very similar. In both cases, events and parameter are represented as a trigger on a state transition. For example, Figure 3 has a call event 'startAutopilot' with the parameter 'normal', which causes the state 'Manual' to change to 'Automatic'.

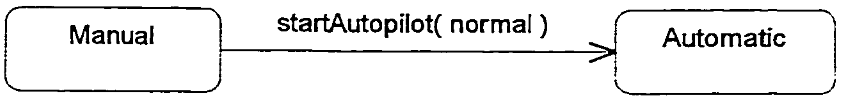


Figure 3: Call Events [BRJ99]

A *time event* is an event that represents the passage of time. For example, if one wants to model the number of seconds passed after picking up a telephone to hear a dial tone, then a time event (for example, after 1 second) can be used. A *change event* is an event that represents a change in state or the satisfaction of some condition. For example, if we want to model how long a furnace should stay on based on an upper temperature limit, then a change event could be used. Figure 4 shows an example of change and time events.

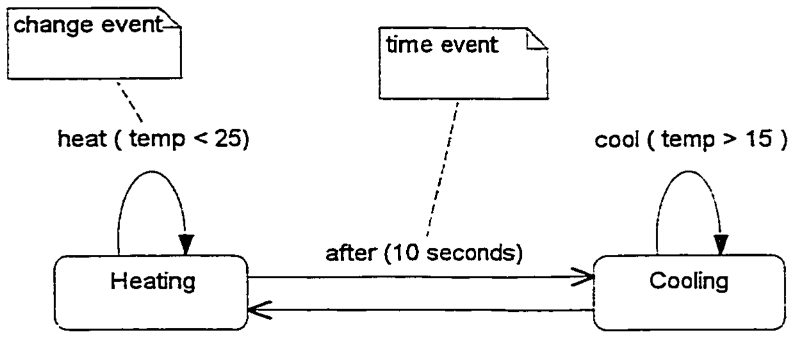


Figure 4: Change and Time Events

2.4 States

A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event [BRJ99]. A state has several (optional) components:

- **Name.** The name is used to identify a state. If it is not specified, then it is an anonymous state. However, each state will have a unique identifier that does not need to be presented.
- **Entry/exit actions.** Actions executed on entering and exiting the state.
- **Internal transitions.** Transitions that do not cause a change in state.
- **Substates.** The nested structure of a state, which can be composed of sequential or concurrent substates.

- **Deferred events.** A list of events that are postponed to be handled by an object in another state.

A state is represented by a rounded rectangle with three or four compartments depending on whether it is a simple or composite state. The *name* of the state distinguishes it from other states, and it may be empty, meaning that it is an anonymous state. The *entry* and *exit actions* are executed upon entering and exiting a state, respectively. Other components of a state will be discussed in later sections. For example, Figure 5 shows a simple state with a name and entry/exit states defined. The name indicates that it is in the 'Active' state. The 'entry' action is defined as 'setAlarm', which indicates that an alarm is set when entering the state Active. The 'exit' action 'clearAlarm' indicates that the alarm is cleared when the state is exited.

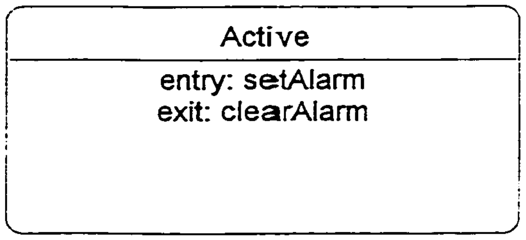


Figure 5: State with entry/exit Transitions [BRJ99]

2.4.1 Initial and Final States

There are two specialized kinds of states: *Initial* and *Final State*. In the UML metamodel [RAT00], these are referred to as *pseudostates*. The initial state indicates the default starting state for the state machine or substate. Likewise, the final state indicates that the state machine or enclosing state has terminated. Initial and final states do not have time duration, meaning that an object cannot be *in* an initial or final state. A small black circle represents the initial state and the final state is represented by a bulls-eye. Note that initial states cannot have incoming arrows and final states cannot have outgoing arrows. Figure 6 shows an example of how to represent an initial and final state.

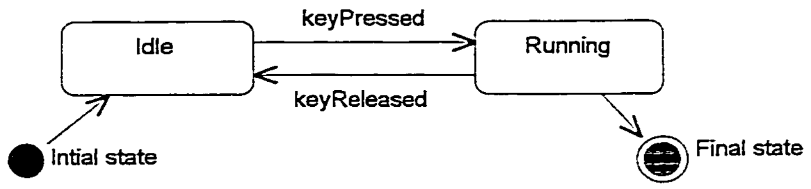


Figure 6: Initial and Final states [BRJ99]

2.5 Transitions

A transition indicates a relationship between two states such that an object in the first state will perform certain actions and enter the second state when a specified event occurs and certain conditions are satisfied. A transition is represented as a directed edge from a source state to a target state. A transition consists of five (optional) parts:

- **Source state:** This is the state which is affected by the transition
- **Event trigger:** The event that triggers the transition
- **Guard condition:** The condition which must be satisfied for the transition to fire
- **Action:** the action that occurs after the transition fires.
- **Target state:** the state which is active after the transition finishes

The parts of the transition are rendered as a string consisting of:

Event [condition]/action

For example, in Figure 7, assume that we are currently in the 'Idle' state. If the event keyPressed occurs and the Key is Help, then the action 'displayHelp' will occur and the system will change to the 'Running' state.

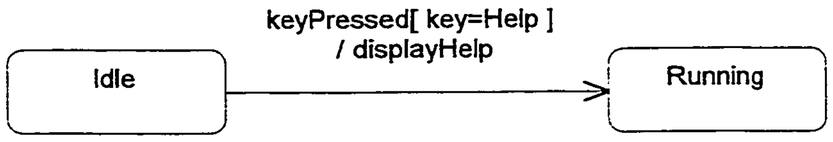


Figure 7: State Transition

2.5.1 Entry and Exit Actions

Frequently a modeling situation arises where the same actions are present on various incoming transitions. In order to simplify the modeling of this behaviour, a state may contain entry actions, which are executed on every incoming transition (after the transition's action). Also, when exiting a state, the same actions may be present on each outgoing transition. This may be modeled by using exit actions. In order to present this to the user, the keywords '`entry`' and '`exit`' are used for the event. For example, Figure 8 shows the entry and exit actions for the state 'Tracking'. Every transition entering the Tracking state will execute the action 'setMode(onTrack)'. Likewise, on exiting, every transition exiting this state will execute the action 'setMode(offTrack)'.

2.5.2 Internal Transitions

Often we want to execute an action within a state in response to some event but we do not want to leave the state. This can be handled by an internal transition. Internal transitions are special transitions in that they do not fire the entry and exit actions (unlike self-transitions). As an example, Figure 8 shows an example of internal transitions. In this example, if the event 'NewTarget' occurs, then the method 'tracker.Acquire()' method would execute without executing the entry or exit actions.

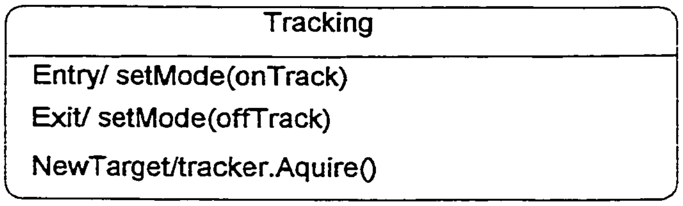


Figure 8: Internal Transitions

2.5.3 Self-transitions

Self-transitions are transitions from a state to itself. The result of this is that the exit and entry actions are executed. For example, in Figure 9, the event 'digit' causes the transition to leave the state 'dialing'. In so doing, the exit action 'sendSignal' is executed, followed by the action 'dialAction' on the transition, and finally, the entry action 'initSignal' is executed.

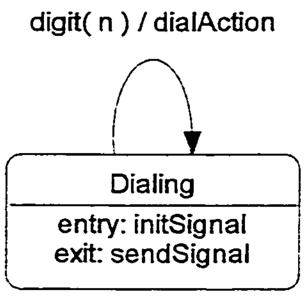


Figure 9: Self-transition

2.6 Substates

A *substate* is a state that is nested inside of another state. The ability of a state to contain other states is the key to simplifying the modeling of complex behaviours. It is also a feature that differentiates it over traditional state-transition diagrams.

2.6.1 Simple states versus Composite states

It is useful to distinguish between simple states and composite states. A *simple state* is a state that has no substates. A *composite state*, on the other hand, can contain substates. A composite state may be expanded or collapsed in order to expose the more or less details of its contained substates. This allows the view of the state to change, depending on its desired level of detail. This will have major implications in the chapter on the design of the zooming viewer. Figure 10 shows a statechart diagram containing three (3) Simple States (State A, B, and C). In addition, State D is a composite that contains State A and State B.

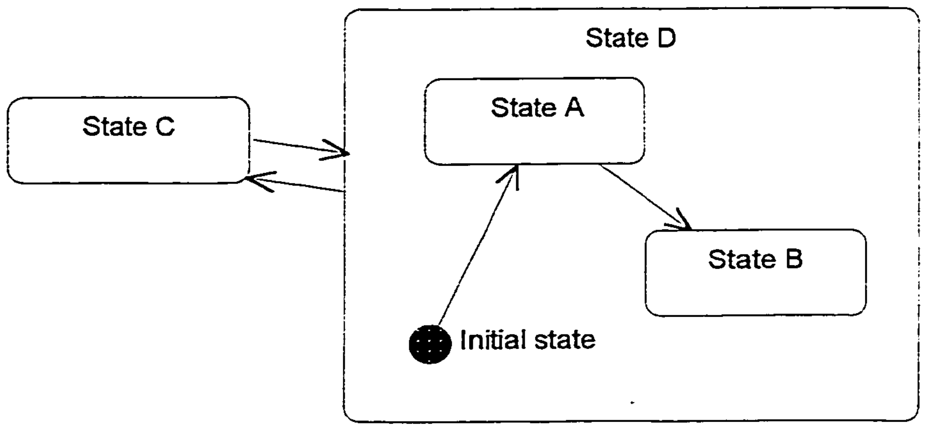


Figure 10: Simple versus Composite State

2.6.2 Sequential versus Concurrent Substates

Sequential substates represent the 'OR' decomposition of a state, meaning that a state may be in one of several disjoint substates. In other words, only one state can be active at any one time. Figure 11 shows that the sequential substate of the composite state 'Active' can be either in the 'Selecting' state **OR** in the 'Processing' state, but not both at the same time.

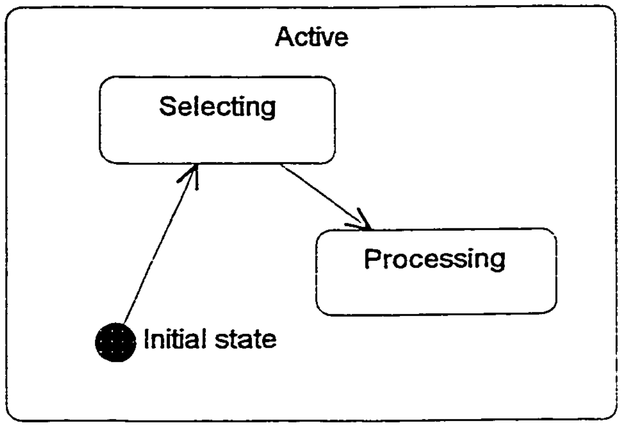


Figure 11: Sequential Substates

Concurrent substates allow the specification of two or more state machines to execute in parallel in the context of the enclosing object. Concurrent substates represent the 'AND' decomposition of a state. For example, in Figure 12, there is the 'Active' state that consists of two concurrent substates 'Testing' **AND** 'Commanding' running in parallel. Each concurrent substate must contain at least one sequential substate. In this example, 'Testing' has 'Running' and 'Self test' as its substates. This means that an object in the 'Active' state can fork two threads of execution, one representing the 'Testing' and the other the 'Commanding'. Only when both states reach their final states does the state cease to exist.

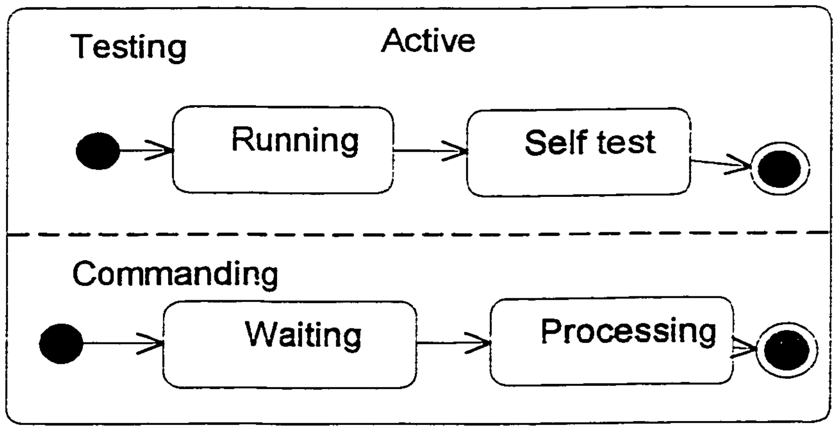


Figure 12: Concurrent Substates

2.7 Complex Transitions (Fork/Join)

Fork and join pseudostates are used to split and merge (synchronize) concurrent threads of execution. Both fork and join states are represented as short solid vertical bars. They are not supported in the current implementation of the statechart[i]. The author considers it a simple extension to add this capability to the statechart viewer.

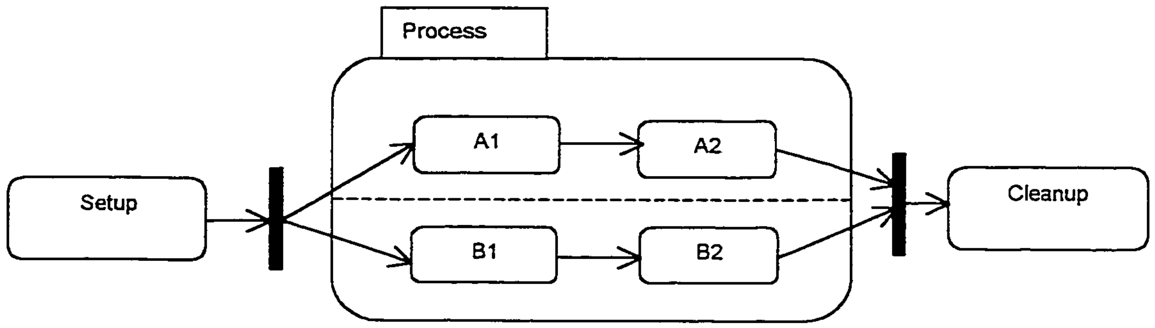


Figure 13: Forking and Joining of Transitions [OMG99b]

2.8 History States

History states are used as a way for an object to remember its past behaviour. A history state represents the last state was active prior to leaving a composite state. It is represented as a small circle with the symbol H (history). Figure 42 in Appendix A shows an example of *shallow* and *deep* history states. The symbol H designates a shallow history, which remembers only the history of the immediate nested state machine. You can also specify deep history, shown as a small circle containing the symbol H*. Deep history remembers down to the innermost nested state at any depth [BRJ99].

[i] As of this writing, Rational Rose 2000 does not implement fork/join of states in its statechart editor. Nor does it implement concurrent states.

2.9 Hierarchy Issues

This section discusses the issues involved with rendering hierarchical statechart diagrams. Since a composite state is capable of expanding and/or collapsing its containing substates, there are several items that need to be considered when zooming.

- How to hide states and transitions?
- How to represent stubbed transitions?
- When to display or hide substates?
- How and when to represent the textual components?

The substates of a composite state may be hidden from view *statically,* that is independently of zooming level. For example, Figure 14 and Figure 15 show a simple and effective technique used by the Rational Rose tool to hide substates. Figure 14 represents an expanded view of a state diagram while Figure 15 shows a collapsed view of the same diagram.

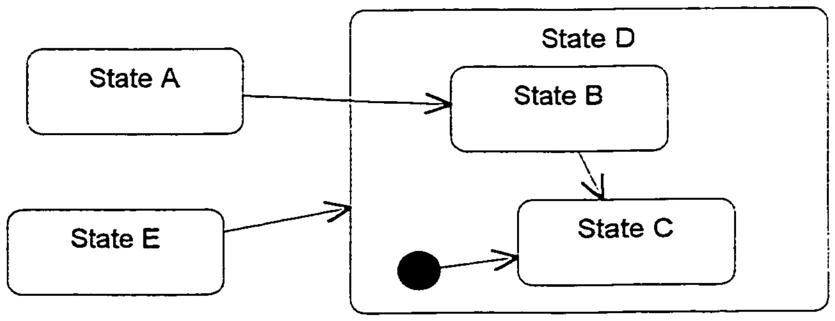


Figure 14: Expanded View of Substates

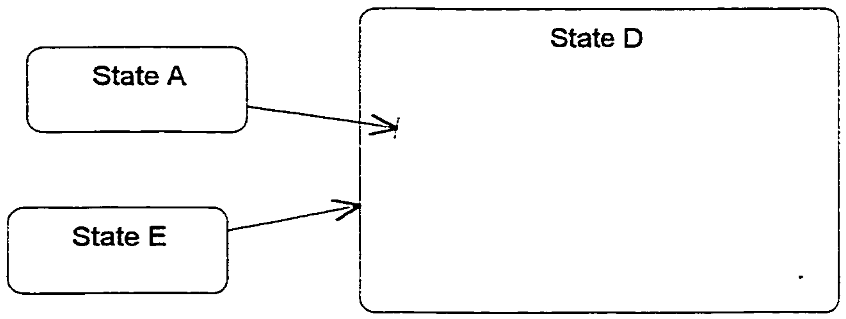


Figure 15: Collapsed View of Substates

When a state is collapsed, the transition edges crossing a state's border can be represented by *stubbed transitions*. Figure 15 shows a stubbed transition entering State D. However, the precise substate that it entered is *not* specified. In other words, from this we cannot easily tell if the state entered is State B or State C. This is one of the problems with stubbed transitions. A small icon (e.g., *) in the bottom left corner represents a composite state when collapsed.

Another approach is to allow states to be collapsed or expanded *dynamically*. This is the approach taken in this thesis. As the user zooms in, the user sees more details. States are expanded dynamically during the zooming of the diagram. For example, at one level of detail, the user may see a single state. As the state increases in magnification, the name appears. Next, the contents (assuming this is a composite state) are exposed, revealing the intricate and detailed structure of its substates. In a similar fashion, as the user sees more details, each level exposes more and more details.

The main issue with this dynamic approach is whether or not to allow states to expand/collapse in an independent manner or as a whole. This thesis takes the latter approach since it is simpler to implement. However, independent zooming can be an interesting future goal. Another issue involves the zooming of the textual components. Either the text increases or decreases in size while zooming or it does not. If the text is too small, then it should be hidden from view when zoomed out.

2.10 Discussion

It is interesting to note that many tool builders do not support concurrent substates, and by extension, forking and joining of complex transitions. In particular, there is no way in Rational Rose or Argo/UML to represent such notation. The reason for this is not clear. Rendering a concurrent state diagram does not seem to be a problem. The only reason that can be thought of is that it is difficult to generate code for a concurrent statechart diagram. Booch even refuses to support concurrent (orthogonal) states in his methodology, stating that it is unnecessary to so. In his opinion, objects are implicitly concurrent and therefore not necessary to model as such [BOOC94]. It is this author's opinion that concurrent state diagrams should be used since because many objects can contain concurrent 'threads' of execution, which need a notation and support to be modelled effectively. However, the lack of tool support from Rational and Argo/UML makes this difficult to implement, since there is no information on how to render concurrent states. Manual editing of an XMI file may be the only way to represent concurrency at this time, and this does not seem a viable option for the average end-user.

2.11 Conclusion

This aim of this chapter was to gain a better understanding of statecharts so that a zoomable statechart viewer can be designed and implemented. The focus on hierarchical issues was made in an effort to understand the challenges involved. Various techniques to reduce the complexity of a statechart diagram were discussed, such as the ability to collapse and/or expand composite states. However, as it can be seen, these efforts resulted in other layout and presentation problems that need to be addressed, such as the use of stubbed transitions. These will be examined in latter chapters.

Chapter 3

STATE OF CURRENT TECHNOLOGY

3.1 Introduction

This chapter examines the state of the current technology. The aim is to select a graphical framework for the design of the statechart viewer based on the current technology available. A statechart diagram can be easily represented on a two-dimensional (2D) surface. However, adding zooming capabilities involves the selection of a framework that can combine some features of three-dimensional (3D) modeling in a two-dimensional (2D) world. The next chapter introduces Jazz, a graphical framework for 2D graphics, which contains a number of features that combines the best of both worlds.

3.2 Background

Toolkits for building graphic applications can be broadly categorized as two-dimensional (2D) and three-dimensional (3D) [BM99]. Most of the research in the graphics area has emphasized 3D because of the level of interest, the computational requirements, and the complexity of building such applications. 3D applications are widely used in medical, scientific, game development. However, 2D applications are also now becoming prevalent in various domains such as business and Internet applications. Since the range of applications for 2D appears to be greater than for 3D applications, there is a need for a robust framework for 2D graphics.

3.3 Graphic Framework

3.3.1 2D Frameworks

All GUI toolkits include 2D graphics. Frameworks that support 2D graphic applications include:

- Microsoft Win32 API. Based on C++ for the Microsoft platform, it provides basic drawing operations.
- Mac toolkit. Used on the Macintosh platform.
- Xlib. A C++ toolkit for X windows.
- Tk Canvas.
- Java 2D. Discussed below.

Most 2D frameworks include only a *renderer* for 2D graphics. There is no way to *structure* the graphical information into hierarchies or groupings. Except for some (e.g., Java 2D), most of these frameworks are platform dependent.

3.3.2 3D Frameworks

- Silicon Graphics Inc. (SGI) OpenInventor.
- Java 3D
- Many others

Most 3D frameworks provide both a *renderer* and a *scenegraph.* A scenegraph is used to help an application manage the structure graphical objects. 3D frameworks tend to have a large number of features to assist in modeling three-dimensional worlds. This tends to increases the difficulty of using such a framework, since the developer must learn a large number of (possibly unused) classes and methods and understand their inter-relationships.

3.3.3 2D+ Frameworks

- Jazz 2D Graphics Toolkit

This framework combines a 2D renderer with a scenegraph. This will be discussed in the following chapter.

3.4 Java-Related Technologies

Java is regarded today as the language of choice for developing distributable applications over the Internet. Sun describes Java as a "*simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, and dynamic language*" [FLAN97]. For graphical applications, the main advantage to using Java is the wide range of available frameworks for constructing robust applications. The main disadvantage of Java is its current poor performance since it is an interpreted language. However, it is believed that this will not be an issue in the years to come as the implementation of the Java Virtual Machine (JVM) improves.

3.4.1 Java Foundation Class (JFC)

The Java Foundation Class (JFC) is included with Java 2 (core API). It contains Java 2D, Swing, the Accessibility API, and Drag and Drop. Swing is based on the Abstract Windowing Toolkit (AWT). The important thing to note is that Java 3D is not included in the "core" API. Therefore it cannot run with earlier versions of the Java Development Kit (JDK).

3.4.2 Java Media API

The Java Media APIs are designed to provide Java multimedia capabilities. It includes Java 2D, Java 3D, the Sound API, and the Advanced Imaging API.

3.4.3 Java 2D API

The Java 2D API is an Application Programming Interface for two-dimensional graphics. It is a set of classes that can be used to create high quality graphics [KNUS99]. It includes support for geometric transformations, antialiasing, alpha compositing, clipping, text, colour, images, image processing, and printing.

The Java 2D is used as a foundation in developing the Jazz framework.

3.4.4 Java 3D API

The Java 3D API is an Application Programming Interface used for writing three-dimensional graphics applications and applets [SRD99]. It gives developers high-level constructs for creating and manipulating 3D geometry and for constructing the structures used in rendering that geometry. Application developers can describe very large virtual worlds using these constructs, which provide Java 3D with enough information to render these worlds efficiently. The Java 3D API provides a scenegraph, however it is an extremely complex toolkit to use. Its goal is to satisfy the needs of the 3D modeling community. The Java 3D API uses the Java 2D primarily for developing textures for 3D.

3.4.5 Swing

Swing is used for developing GUIs, beans, applets, and plug-ins. Java 2D can be used to draw on Swing components via the `paint()` method.

3.5 Other Related Internet Technologies

3.5.1 Virtual Reality Modeling Language (VRML)

VRML is an authoring standard for placing 3D content on the Web. It is a text based file format that can be used to model 3D objects and create virtual worlds. The current standard uses a scenegraph similar to Java 3D that can handle events and perform animation. In fact, Java in conjunction with the Java 3D API can be used to load a VRML file and create Java 3D objects within it. Thus, a VRML browser can be created. In a similar fashion, a Java program can read XMI, which is a standard for exchanging UML documents, objects created in a Jazz framework, and browsed.

3.5.2 Dynamic Hypertext Markup Language (DHTML)

Dynamic HTML is a term used by some vendors to describe HTML pages with animated (changing) content. It combines Cascading Style Sheet (CSS), Hypertext Markup Language (HTML), and JavaScript. The three components are tied together using the Document Object Model (DOM). DOM is a standard for representing documents. However, DHTML is not a W3C standard [W3C00].

3.5.3 Flash

Macromedia Flash is an authoring tool for the Web [MACR00]. It offers a dynamic, media-driven standard for designing website navigation interfaces, technical illustrations, animations, and other effects. It uses layering techniques to place graphic objects on different levels. The idea of layering is similar to the Multi-Layer model [FBL96] and is used in the design of the statechart viewer.

3.6 Industry Standards

When designing a large software system, it is desirable to use industry standards. This ensures that the quality of the software is high and that many users (e.g., architects, system designers, programmers, etc...) can understand the system. It is also desirable to exchange information amongst the various users. XMI is an XML-based industry standard for exchanging object models (especially the UML metamodel) and allowing tools to interoperate. These terms are described below.

3.6.1 UML

Unified Modeling Language (UML) is an industry standard language for specifying software systems. There are nine different models (or views) in UML: Class diagram, Object Diagram, Use case diagram, Sequence diagram, Collaboration diagram, Statechart diagram, Activity diagram, Component diagram, Deployment diagram [BRJ99]. Some of these are static in nature (e.g., Class diagrams) while others are dynamic. A statechart diagram is one example of a dynamic model. These different models will be used in the chapters on the architecture and design of the StateChart Viewer.

3.6.2 eXtensible Markup Language (XML)

Extensible Markup Language (XML) is a language used on the Internet for exchanging information. It consists of markup tags, which can be used to identify each item. For example, the following is a small sample of how to specify a statechart consisting of two states, where the first contains an optional entry action.

```
<Statechart>
    <State name="State 1"/>
    <EntryAction>Do Entry</EntryAction>
    <State name="State 2"/>
</Statechart>
```

The major advantage of using XML is that it can store structured and semi-structured data [BRAD00]. Since a statechart can consist of many parts and these parts can be arranged in a recursive tree-like manner, XML is a prime candidate for specifying the information contained in a statechart diagram. A Document Type Definition (DTD) is used to specify the legal ways that the tags in an XML document can be used. However designing a DTD from scratch is not an easy task. It is desirable to use an existing XML standard for exchanging UML model information.

3.6.3 XMI

XML Metadata Interchange (XMI) is an XML-based format used for exchanging object models. XMI specifies an open information interchange model that gives developers working with object technology the ability to exchange models and data over the Internet in a standardized way, thus bringing consistency and compatibility to applications created in collaborative environments [IBM00]. XMI is used by the Object Management Group (OMG) to facilitate the exchange of UML models.

3.6.4 XMI versus UXF

It is desirable to have a standard for exchanging UML information independently between different software vendors. There were two standards that where considered in this design, UML eXchange Format (UXF) [UXF99] and XML Metadata Interchange (XMI). Both provided a standard exchange format for UML models. The basic structure of XMI and UXF are similar. UXF was designed to be a research tool prior to the release of XMI. Originally, UXF was the preferred choice since it was simpler to use and easier to understand. However, it had several shortcomings, such as the inability to uniquely identify states. Also, it was not an approved standard by the Object Management Group (OMG). For this reason, it was decided to select XMI as the standard for exchanging UML model information in this thesis.

3.7 Zooming Techniques

This section outlines the common techniques used for zooming that are relevant to the design of the statechart editor. Finding detail in a larger context is known as the *focus-context* problem [WARE00, p.355-362]. These techniques are said to solve the focus-context problem.

3.7.1 Rapid zooming

The 'Jazz' framework, presented in the following chapter, uses this technique. It involves navigating a large information landscape, although only a part of it is visible in the viewing window at any instant. The user is given the ability to zoom rapidly into and out of points of interest. The user can rapidly and smoothly move from focus to context and back.

3.7.2 Structural scale (Level of Detail)

This is representing a large amount of information at varying levels of detail. For statecharts, we may have different representations at different scales.

3.7.3 Elision Techniques

This involves visually hiding some information until it is needed. For example, a complex state can be collapsed and viewed as a simple state. In the case of text, less and less details are shown, as the distance from the focus of interest increases.

3.7.4 Multiple windows

This involves using multiple windows or portals to view the same data at different levels. For example, one may have an overview window and a details window. This can be accomplished with Jazz by using multiple canvases and/or cameras.

3.7.5 Distortion techniques (Fisheye views)

This involves distorting the information spatially so that more room is given to points of interest and less space is given to regions away from those points. Fisheye views, according to Furnas [FURN86], this provides a technique whereby nearby objects appear large and far away objects appear small. This technique allows the representation of large amounts of data on a small screen space. This allows the user to dynamically alter the *view* so that more interesting objects appear nearer (and larger).

The hyperbolic tree browser [LPR95] is another example of distortion that allows the focus to be changed by dragging a node from the periphery into the center [WARE00, p.357-358]. The user is able to maintain the context and does not get lost when presented with a large tree of nodes. The nodes closest to the center (focus) appear larger than at the periphery (context).

3.8 Conclusion

The aim of this chapter is to select a graphical framework for the design of a statechart viewer. Since statechart diagrams are easily visualized on a two-dimensional surface, there is really no need for the complexities involved in trying to represent them in a 3D world. Therefore, there is little advantage in selecting a three-dimensional (3D) graphical framework. However, 3D frameworks provide several advantages over 2D frameworks, such as the ability to provide structure to simple graphical objects. In the next chapter, we will examine the Jazz framework, a graphical framework for 2D applications. The reason for selecting this framework is that it provides 2D applications with both a *rendering engine* and the ability to *structure* graphical objects. Jazz can also be used to assist us in implementing the various zooming techniques discussed in this chapter. In the following chapters, the XMI standard for exchanging UML models will be used to represent the model (state hierarchy) and view (positional information) of the statechart. The main advantage of this is that XMI is an accepted standard and it has a well-defined DTD.

Chapter 4

JAZZ: A FRAMEWORK FOR 2D GRAPHICS

4.1 Introduction

Jazz is a general-purpose Java-based graphics engine that supports 2D visualizations [BEDE00]. Jazz is built on top of Java2D, which is the Java toolkit for 2D applications. Jazz provides several features common to 3D graphic frameworks, such as a *scenegraph*, which allows applications to structure primitive graphical objects in a 'scene' that can be grouped and manipulated by various operations or methods. A scene (or 'virtual universe') is made up of several parts: behaviour, model, object characteristics, coordinates, and everything needed to create a complex world of graphical objects [BP99]. The aim of this chapter is to provide a background on Jazz, which will be used in the design and implementation of the statechart viewer in later chapters.

4.2 The Jazz Framework

Jazz provides many services to application developers. By basing an application on such a framework, the time to develop is reduced since most of the details (such as zooming, panning, primitive shapes, scenegraph structure, etc...) are implemented in the framework. Jazz provides a number of features to aid the development process:

- Scenegraph support. The ability to structure primitive graphical objects into a scene.

- Built-in support for panning and zooming. This allows the developer to build an application that the user can navigate a 2D information space.
- Hierarchical grouping with affine transformations (translation, scale, rotation, and shear)
- Multiple layers. This allows graphical objects to be presented on different layers when rendering.
- Semantic zooming. Allows a scene to be rendered differently depending on the context (e.g., scale).
- Multiple representations. Internal cameras and lenses allows for multiple views of the underlying model in question.
- Multiple platform support. Since Jazz is based on Java2D, it can run on all platforms that support the Java Virtual Machine (JVM).

4.3 Jazz Architecture

This section outlines the architecture of Jazz. The architecture of Jazz consists of a root scene graph object (ZSceneGraphObject), which contains the common functionality of both nodes (ZNodes) and visual components (ZVisualComponents). The ZNode objects are of two basic types: leaf node (ZLeaf) or group nodes (ZGroup). A ZGroup object can contain either ZLeaf objects or other ZGroup following the composite design pattern [GHJV94]. In this pattern, a ZLeaf object cannot have children, whereas the ZGroup objects can have children (including objects of its own type). Visual components are the basic elements in a scenegraph and they must be attached to a ZVisualLeaf or a ZVisualGroup in order for it to be inserted into the scenegraph. Figure 16 illustrates the Jazz class diagram.

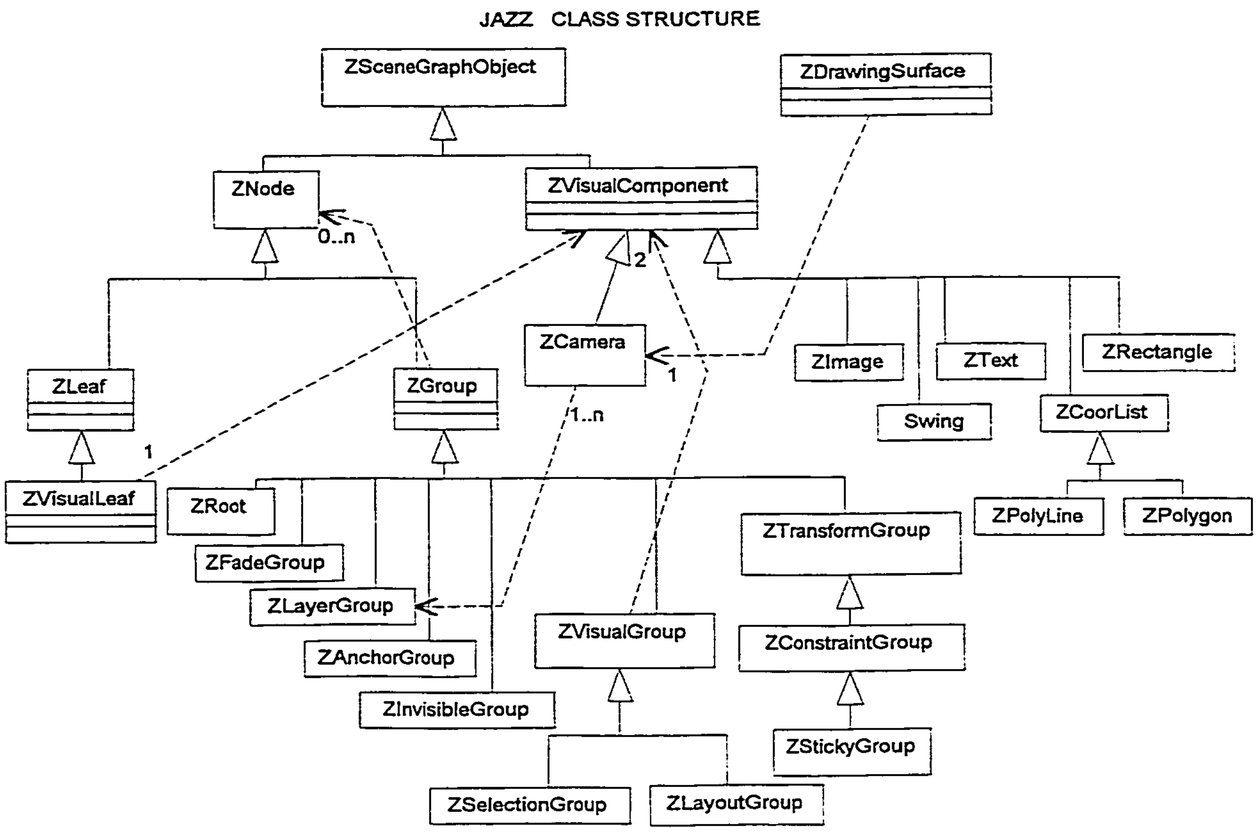


Figure 16: Class Hierarchy of Jazz Scenegraph Objects [BEDE00]

4.4 Jazz Scenegraph

This section discusses the way Jazz applications are structured and can be implemented. A scenegraph provides a way to organize and group visual elements in a in a hierarchical data structure. Each visual element consists of two types of objects in the scenegraph: *nodes* and *visual components*. A ZVisualComponent is the base class for objects that actually get rendered. It specifies the size of a visual element and how to render it. It cannot exist by itself in the scenegraph and, thus, is always associated with a node. A ZNode is responsible for maintaining the structure of the scenegraph and contains all characteristics (position, scale, transparency, etc...) that are passed on to child nodes. A scenegraph will also have exactly one *root node* and one or more *cameras*. The root serves as a holder to all elements in the virtual universe that is being represented. The camera serves as a viewport into a portion of the scenegraph. Figure 17 illustrates a typical Jazz scenegraph. This scene consists of a camera looking onto a layer that contains a rectangle and a group consisting of two polylines.

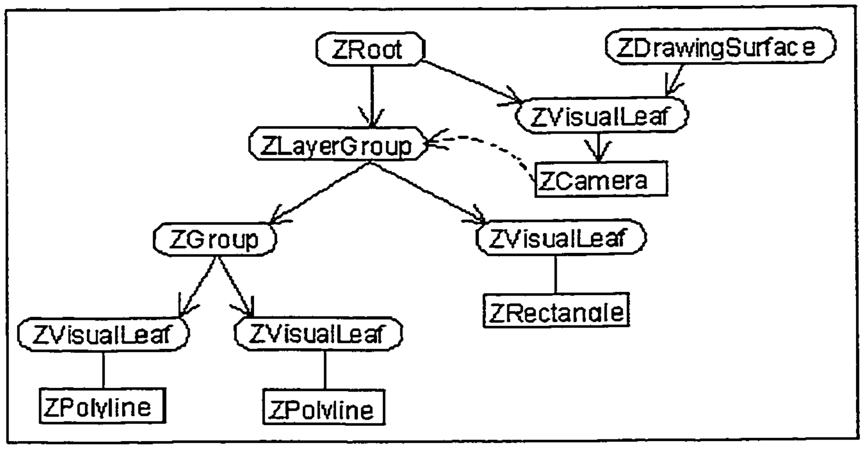


Figure 17: A Typical Jazz Scenegraph [BEDE00]

4.4.1 Visual Components

A visual element is represented by a ZVisualComponent object and is associated with a ZNode. A visual component is a primitive graphical object in a scenegraph. It contains basic information that specifies its size and how to render itself. For example, a ZCircle object can be represented by its centre point, radius, and pen colour. Thus, it contains the 'model' component in the Model-View-Controller (MVC) architecture. In addition, a visual component may contain methods that tell the visual object if it was picked or not. A basic set of visual components is provided with the Jazz toolkit (rectangles, polylines, text, cameras, Swing components, etc...). Visual components are easily extensible. If a new graphical object is needed, such as a rounded rectangle (ZRoundRect) to represent a state shape in the statechart viewer, we only need to extend the class ZVisualComponent and implement the methods `paint()`, `computeBounds()`, and (optionally) `pick()`. The `paint()` method is used to render the object, the `computeBounds()` is used to find the size of the object, and the `pick()` method tells whether an object has been picked or not.

4.4.2 Nodes

Nodes are the basic building elements that form the structure of the Jazz scenegraph. Jazz contains several types of nodes, each providing an application with additional functionality. For example, a ZTransformGroup node allows an application to position, scale, and rotate a visual element. A ZFadeGroup allows a visual component to fade in/out depending on the level of magnification. In addition, several node types can be combined to achieve various effects. Table 1 shows the kinds of nodes that can exist in a Jazz diagram

ZRoot	Each scenegraph contains exactly one root which serves a the root of the entire scenegraph tree
ZLeaf	Serves as a tag, identifying all sub-classes as being leaves.
ZVisualLeaf	ZVisualLeaf is a leaf node with a visual component. Visual components are normally attached to visual leaves.
ZGroup	ZGroup is a node that serves to group children. Each time a group node is manipulated, it affects all of its children
ZFadeGroup	ZFadeGroup is a group node that controls transparency and fading of its sub-tree. When a fade group is inserted, the transparency and minimum/maximum magnification can be controlled.
ZLayerGroup	Is used to specify the portion of a scenegraph, which the camera can see.

ZAnchorGroup	Used for hyperlinking. It is not used in the statechart viewer
ZInvisibleGroup	Used for making all the nodes below it to be invisible.
ZVisualGroup	ZVisualGroup is a group node for visual components. It has two visual components, which tells which node is rendered in front of or behind another visual component
ZSelectionGroup	ZSelectionGroup is a visual group that allows a group of visual components to be selected.
ZLayoutGroup	Allows nodes to be positioned automatically according to a layout manager.
ZTransformGroup	Allows nodes to be positioned using an arbitrary affine transform. This allows the subtree of this node to be translated, rotated, scaled, or sheared.
ZContraintGroup	A constraint group changes its transform based on a computation defined in a specified method. It is called every time the camera view is changed, allowing for dynamic behaviours to be created depending on the view
ZStickyGroup	A special kind of constraint group that moves its children inversely to the camera view. This is used in the case of a text field where the text does not change magnification when zooming in/out.
ZDrawingSurface	Represents the drawing surface that a camera renders upon. This is associated with a ZCanvas window or anything that a Graphics2D can render on.

Table 1: Jazz Object Types [BEDE00]

4.5 Multiple Representation of Objects

This section discusses how an object can be changed depending on the circumstances for which it is viewed. Often it is desirable to let the context of an object being viewed affect the rendering of an object. This is referred to as *context-sensitive rendering*. Two especially common cases are magnification level and camera. When the context of a representation depends on the magnification level, it is termed *semantic-zooming*. When the context depends on the type of camera used when viewing, it is termed *lens* or *filter*. For example, Figure 18 illustrates the case of semantic-zooming. As the magnification level is decreased (zoomed-out), the arrows leading into a composite state is replaced by stubbed transitions.

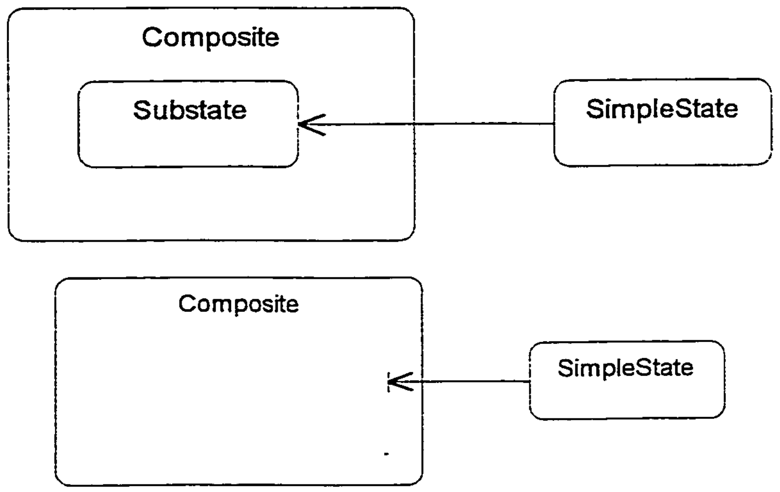


Figure 18: Context-sensitive Rendering of Transitions Depending on Zooming Level

4.6 Conclusion

The aim of this chapter was to describe a framework for the development of a zoomable viewer for statecharts. Jazz provides both scenegraph support and zooming capabilities that are crucial in the design of the statechart viewer. The scenegraph will allow primitive visual components such as lines, rectangles, and text to be structured and managed as a hierarchy of graphical objects in a scene. The primitive graphical objects will eventually represent states, transitions, events, etc... in the statechart viewer. The zooming capabilities that are already built-in to Jazz demonstrate the reusability of the framework. This shortens the development time and improves the reliability of the system.

Chapter 5

SYSTEM ARCHITECTURE

5.1 Introduction

This chapter presents the architecture of our system, StateChart Viewer. The aim of this section is explain the major organizational decisions taken as well as provides the structure to be used in the design of the system in the following chapter.

5.2 Overview of the system

StateChart Viewer is a program that allows a UML statechart diagram based on the XMI[i] exchange format to be imported, rendered, zoomed and navigated. The XMI format is an XML-based file representing a UML statechart. A statechart diagram is constructed using a commercial Computer Aided Software Engineering (CASE) tool, which in this case is Rational Rose 2000 [RAT00]. Rational Rose consists of an XMI export facility that allows the model information to be saved to a file in XMI format. The StateChart Viewer imports the previously created XMI file. Based on this information, a scenegraph of the model and view information is constructed in the StateChart Viewer. This is constructed using Jazz and Argo/UML [ROBB00]. Jazz is used to create the scenegraph, while Argo/UML contains the structural components of the UML metamodel for statecharts. Finally, once the

[i] XMI used in this context is an exchange format based on the UML metamodel. It may also be used for other kinds of models than UML.

scenegraph is created, the statechart diagram is rendered on the screen, and the user is able to navigate and zoom.

5.3 Architecture

The program consists of several layers, as shown in Figure 19.

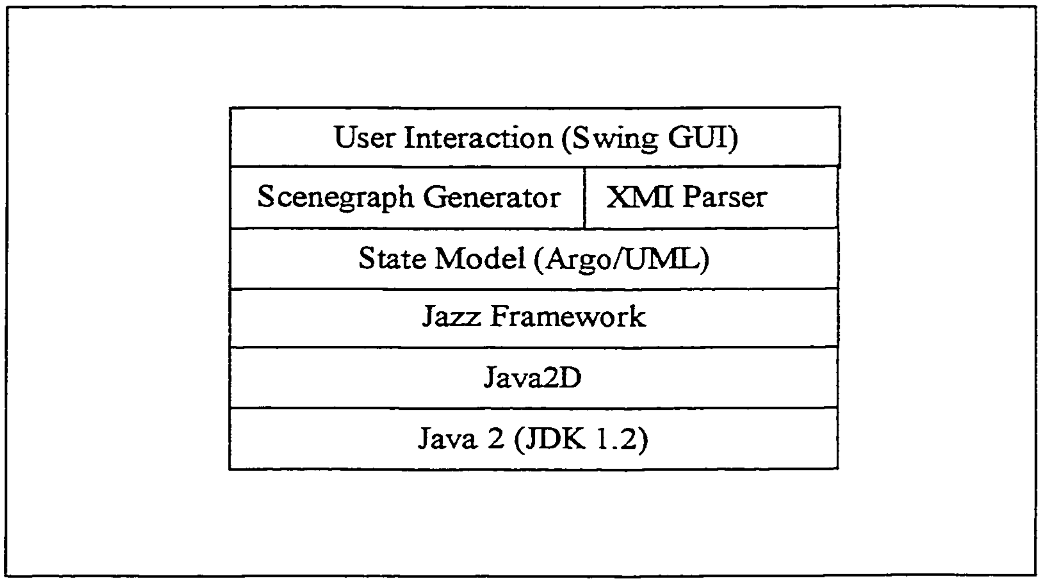


Figure 19: High-level Architecture of the StateChart Viewer

At the top is the User Interaction Layer, which is the user interface responsible for interacting with the user. The user interacts with the GUI by the WIMP (Window, Icon, Mouse, Pointer) model using the keyboard and mouse. For example, the user opens a file from the file menu and then uses the mouse to navigate the statechart diagram in the drawing canvas. These commands will be translated into various actions with the Action interface of Swing. The program first opens a previously created XMI file and parses it with the XMI Parser. The XMI Parser uses the State Model layer to construct the objects to be used by the Scenegraph Generator. The Scenegraph Generator is responsible for building the scenegraph and rendering the statechart diagram on the screen. The scenegraph layer uses the services from Jazz framework. For example, the Jazz framework provides default interaction mechanisms such as panning and zooming. This allows the user to navigate the statechart diagram. Jazz is based on the Java2D framework. Java2D provides basic rendering of the visual components via its `paint()` method. Finally, Java2D is part of the Java 2, the standard Java software development kit (SDK). Java 2, which is the base of the architecture, provides all the classes and methods that support the above-mentioned layers.

5.4 User Interaction Layer

The User Interaction Layer is responsible for executing services on behalf of the end users. This section examines how the user interacts with the system through the user interface. The GUI is built using Swing, which is part of JFC. This provides the structure for all visual components (such as menu, viewing canvas, status bar, etc...). Swing components are used to build the overall look and feel of the application. The interface consists of a menu bar, a canvas (i.e., the drawing area where the statechart is viewed), and a status area. When a user selects an item from the menu, an action is executed, which causes an Action object to be created which performs the required action. However, when the user is interacting with the statechart diagram in the canvas area, it is the Jazz framework that is providing the desired behaviour (i.e., panning, zooming). Finally, the status area is used to provide feedback to the user. Figure 20 shows a screenshot of the StateChart Viewer.

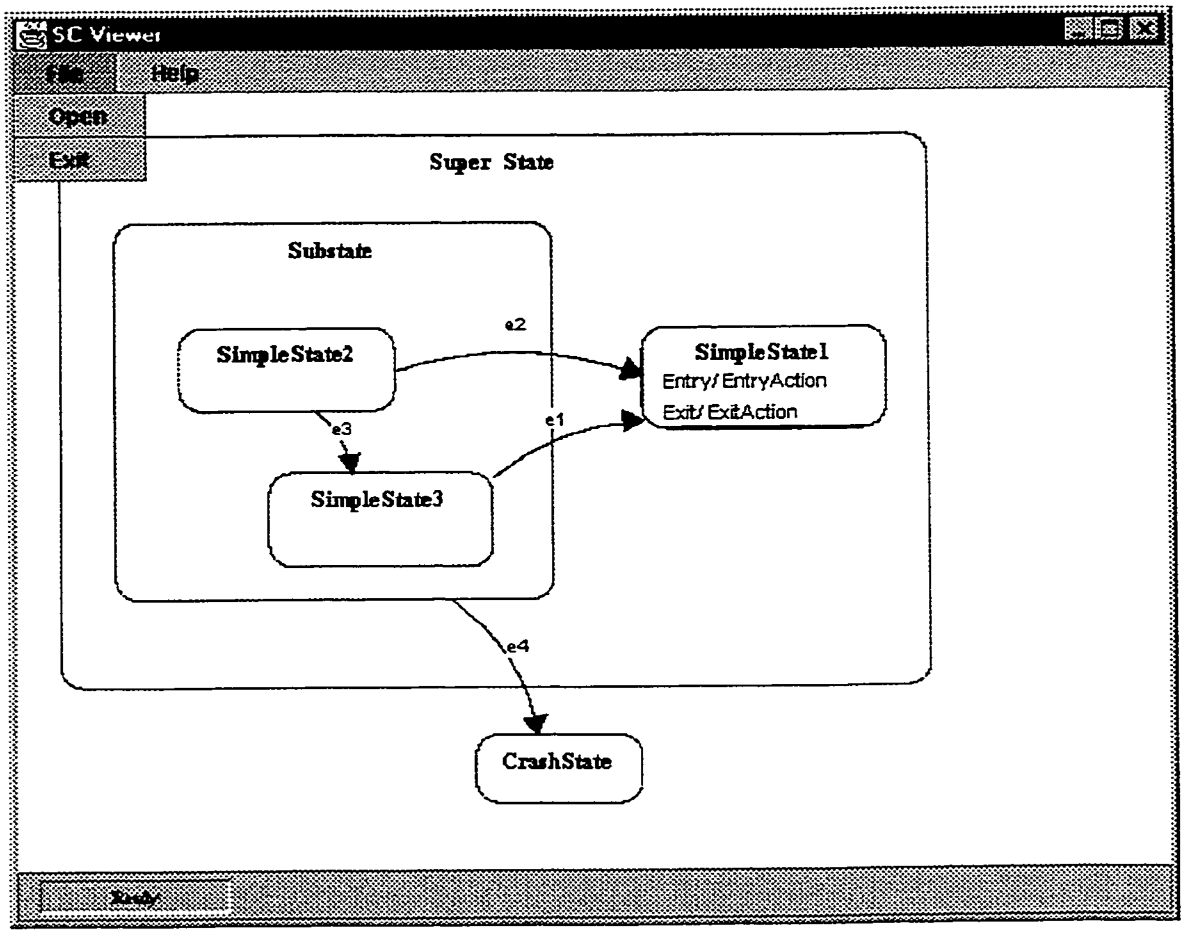


Figure 20: Sample Screenshot of StateChart Viewer

The user can load an XMI file by selecting File/Open from the menu. Once the statechart diagram is loaded, the user can then navigate the diagram by using the mouse. Jazz is then used to provide default navigation capabilities for a scene (i.e., the statechart diagram). It is desirable to show the different ways a user can interact with a system through a *use case diagram*. A *use case* is a description of a set of sequences of actions, including variants, which a system performs, that yields an observable result of value to an actor [BRJ99]. Figure 21 shows the use case diagram for our system.

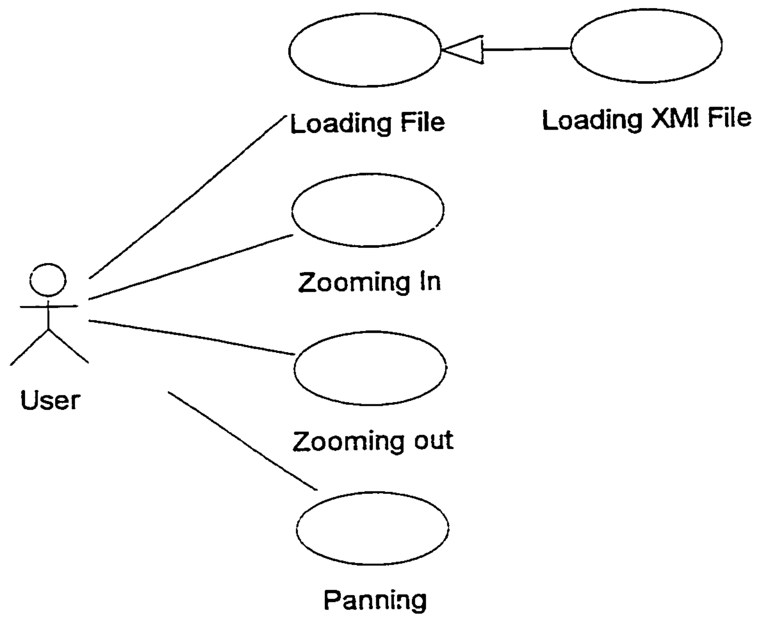


Figure 21: Use-Case Diagram for the StateChart Viewer

The user interacts with the system by loading an XMI file using the File/Open from the main menu. Note that opening a file is common behaviour that can be reused. In fact, Swing provides a default dialog box (JFileChooser) for implementing this behaviour. The generalization arrow shows how loading an XMI file is a specialized case of loading a file. The zooming and out are shown as two (2) separate behaviours, since they are caused by two separate actions.

5.5 XMI Parser

The XMI Parser is responsible for parsing an XMI file and constructing the State Model to be used by the Scenegraph Generator. A user will typically create an XMI file with a commercial software tool, such as Rational Rose, which contains an XMI export facility. The XMI file is an XML file that contains the model and view information of an entire UML diagram. However, for the purposes of this thesis, only a single statechart diagram will be visualized at a time. Therefore, a statechart will always be created for a single class or package. This is a limitation of the current system, since more than one statechart diagrams could be present in a system. In Figure 22, the XMI file consists of two parts: model and view. The model contains the all information relating to the state hierarchy, transitions, actions, and events. The view contains of the location and size of the states.

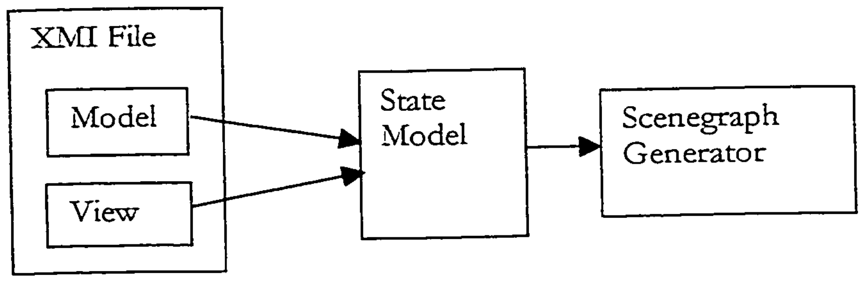


Figure 22: Functional View of the Architecture

The XMI parser uses this information to construct the State Model. The mechanism that the parser uses will be discussed in the following chapter.

5.6 State Model

The State Model is contains the objects that are created during the XMI parsing stage. For example, this contains all objects and relationships between the classes State, Transition, Event, etc... which are represented by the UML metamodel. The UML metamodel is used as a basis for the construction of the State Model. Figure 23 shows a class diagram for the UML metamodel of the state machine. The details of this model will be discussed in the next chapter.

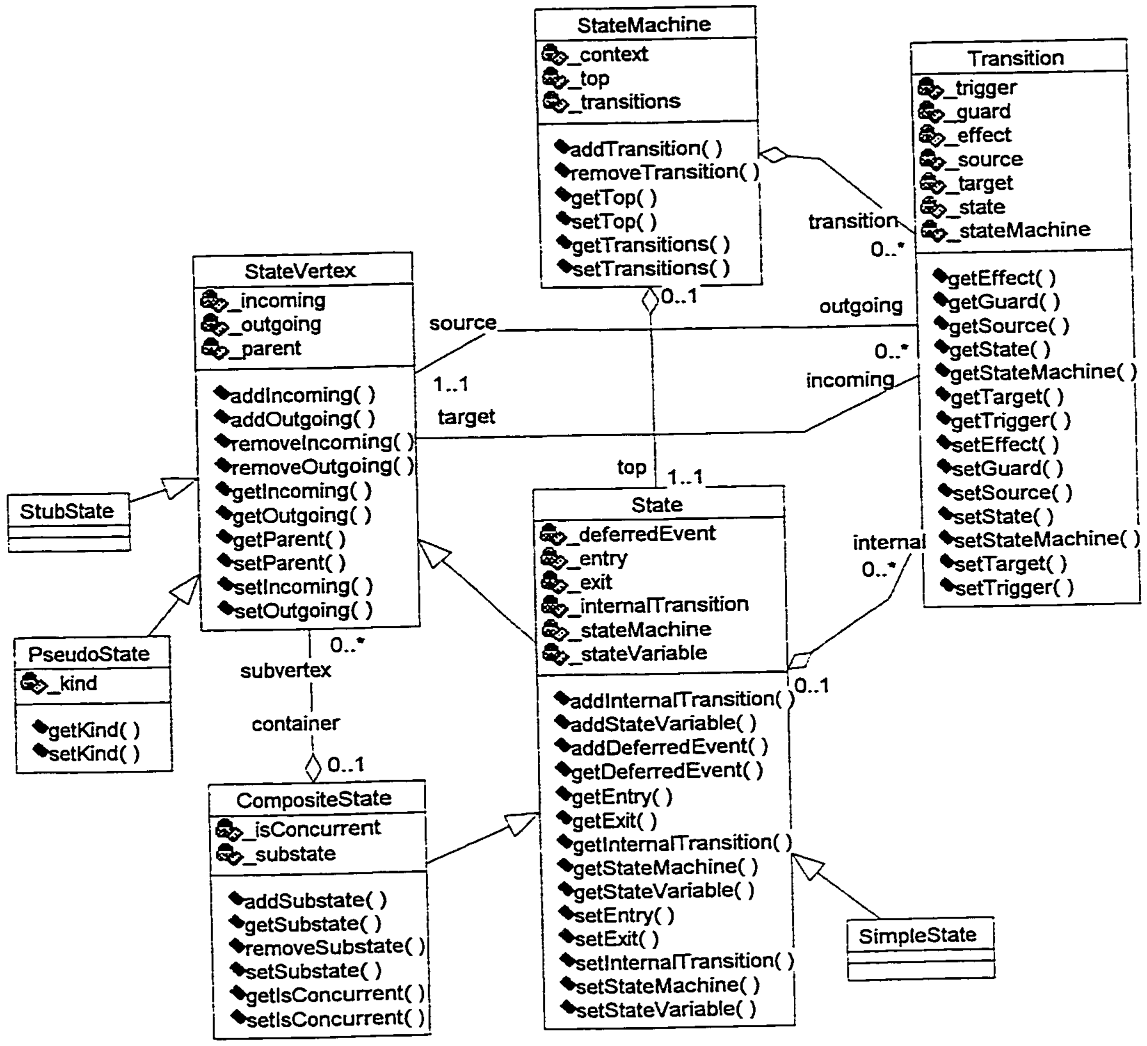


Figure 23: Class Diagram for the StateChart Viewer

5.7 Scenegraph Generator

A scenegraph is a hierarchical data structure that is used to store the visual elements in the Zoomable User Interface (ZUI) [BEDE00]. The Scenegraph Generator is responsible for translating the State Model into a scenegraph that can then be zoomed and/or panned. A Scenegraph object receives an XML file from the user interaction layer. It sends a message to the XMI parser to parse a file. The parser in turn creates the necessary objects in the State

Model. Next, the scenegraph generator receives the root state in the state hierarchy from the State Model. From this, the scene is generated by recursively examining each state in the state hierarchy and placing it on the correct level in the Jazz scenegraph. The level that is chosen is based on the depth of the state in the state hierarchy. The position of the states are added to the model are used to determine the correct position of the states. Finally, the transitions are placed in the scenegraph. Figure 24 shows a simplified mapping between the State Model information and the scenegraph. This will be explored in greater details in the following chapter.

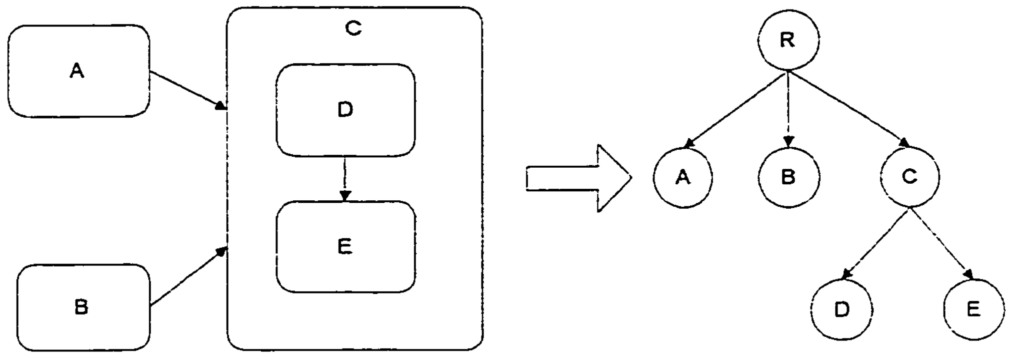


Figure 24: Mapping between State Model and Jazz Scenegraph

5.8 Conclusion

The aim of this section was to provide an overview of the system's architecture. Using a layered approach, we can see how the system is decomposed. This facilitates the design and implementation since the layers are well defined. The interaction between the XMI Parser, Scenegraph Generator and State Model form the major part of the development effort. The rest of the layers offer support for the above-mentioned layers. The XMI Parser provides the input facility that creates the State Model. The Scenegraph Generator uses the information from the State Model to construct the scenegraph. This mechanism ensures that the system is easier to maintain since the system is there is a loose coupling between layers and high cohesion within the different modules.

Chapter 6

STATECHART VIEWER: SYSTEM DESIGN

6.1 Introduction

This chapter discusses the design of a zooming editor for statecharts. The aim is to present the design of the system so that the system can be understood and implemented effectively. The system's static and dynamic aspects can be best userstood by examining it from several views. The UML standard will be used to represent these views.

6.2 Structural Model

In order to make the system more comprehensible, the system is divided into several packages, as shown in Figure 25. These packages are:

- **StateChart Viewer** contains several sub-packages, including:
    - **XMI Parser:** responsible for parsing the XMI file;
    - **State Model:** contains the classes for structuring the state model and view information. This is the model component in the Model/View/Controller (MVC) architecture [RUMB00];
    - **Visual Components:** contains the visual elements used to render the statechart diagram. This is the view information in the MVC architecture;

    - **Scenegraph:** contains the classes responsible for constructing the Jazz scenegraph. This class, in conjunction with the Jazz Framework, provides the controller in the MVC architecture.

- **Jazz Framework:** Provides the framework for constructing a scenegraph and controlling the model and view information.
- **Java 2 (JDK 1.2):** Provides the Java2D package and the Swing package. Java2D provides the graphics capabilities used by Jazz. Swing is used in the construction of user interface components such as menus and toolbars.
- **DOM Parser:** contains the classes and methods used in parsing an XML file used by the XMI parser.

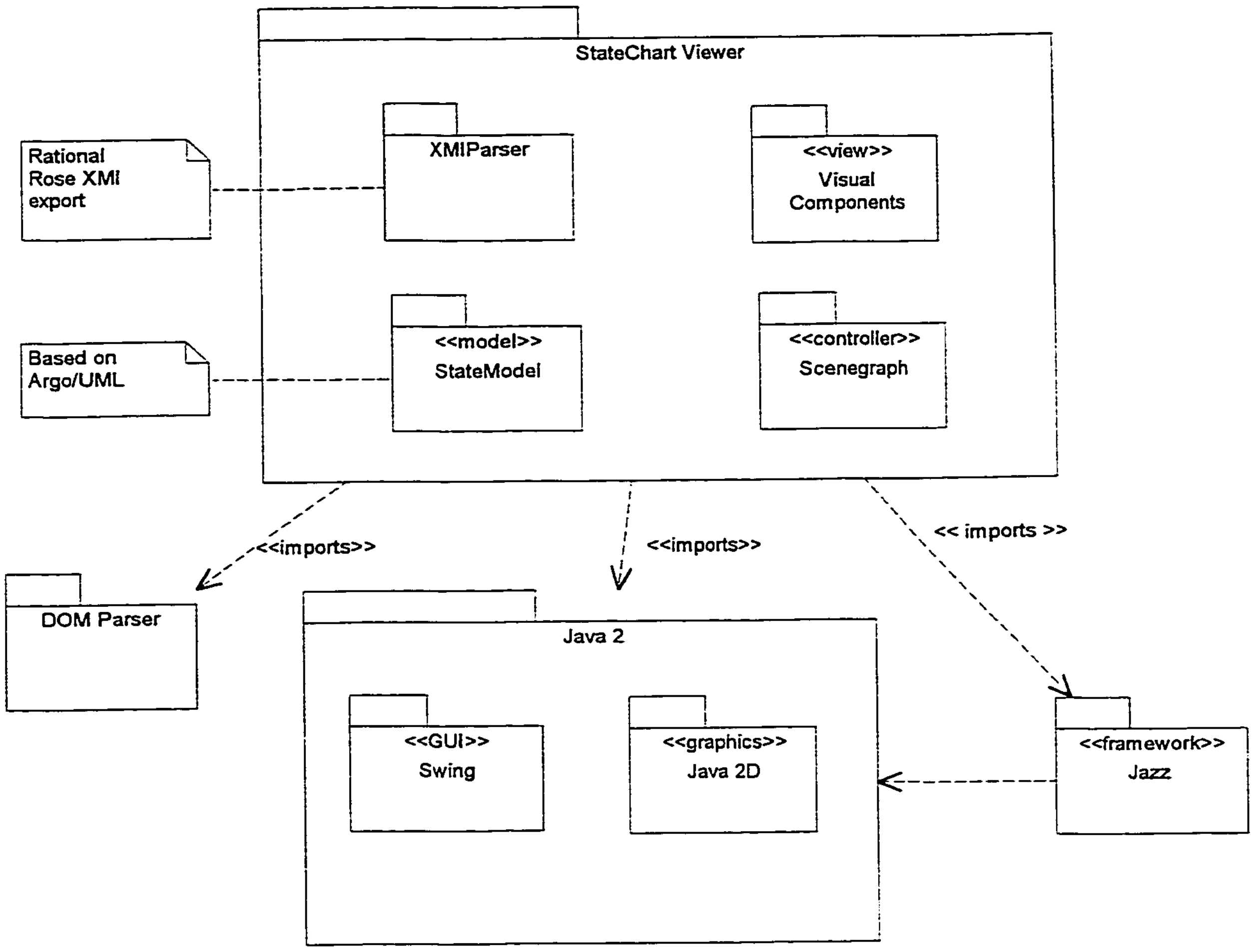


Figure 25: Overview of the System

6.2.1 XMI Parser Package

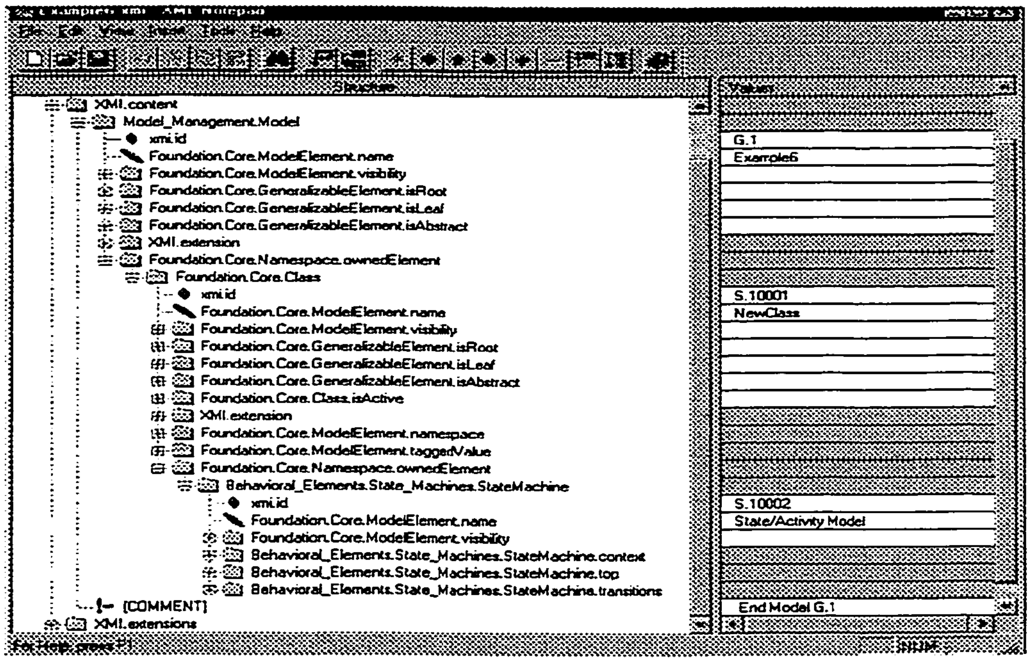
The XMI Parser is responsible for parsing an existing XMI file. The JAXP 1.0 XML parser from Sun is used to construct a Document Object Model (DOM) tree [DOM00]. During the parsing of the XMI file, the State Model is constructed which contains model and view elements. The XMI Parser package contains an XMIParser class to do the actual parsing of the XMI file. The XMIParser class contains a large number of methods to parse each XMI element as shown in Figure 29. The XMI file consists of several logical parts.

- **XMI.header**: Provides basic information on the type of exporter used (e.g., Unisys) and the UML version (e.g., 1.1).

- **XMI.content**: Provides the model information. XMI Content Containing an Overview of the Model Information shows an example of the content using the XML Notepad tool [MICR00]. The 'StateMachine' element is an owned element of the class 'NewClass'. The 'StateMachine' element has a unique identifier (e.g., 'S.10002'), name (e.g., State/Activity Model), visibility (e.g., 'public'), context (e.g., class = 'NewClass'), root state in a state machine, and a set of transitions. Figure 26 shows the elements for the 'StateMachine' element.

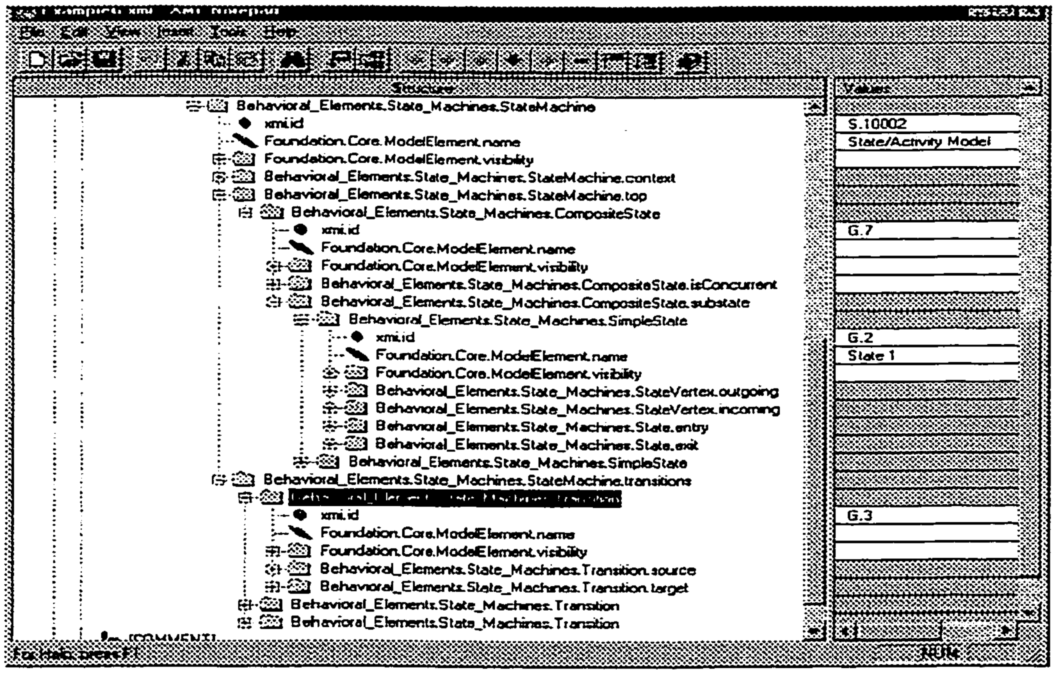
- **XMI.extensions**: Provides the view (geometry) information. The XMI.extensions provide a mechanism to store additional information in the XMI file. In particular, the state's geometry (size and position) is located here. For example, Figure 28 shows the geometry for 'State 1' located in the Foundation.DataTypes.Geometry.body element. The value of this attribute is a fixed integer coordinate (center x, center y, width, height)[i].

[i] This took a while to figure out, since it was not documented anywhere by Rational or Unisys. I assumed that the state position was (top, left, width, height).



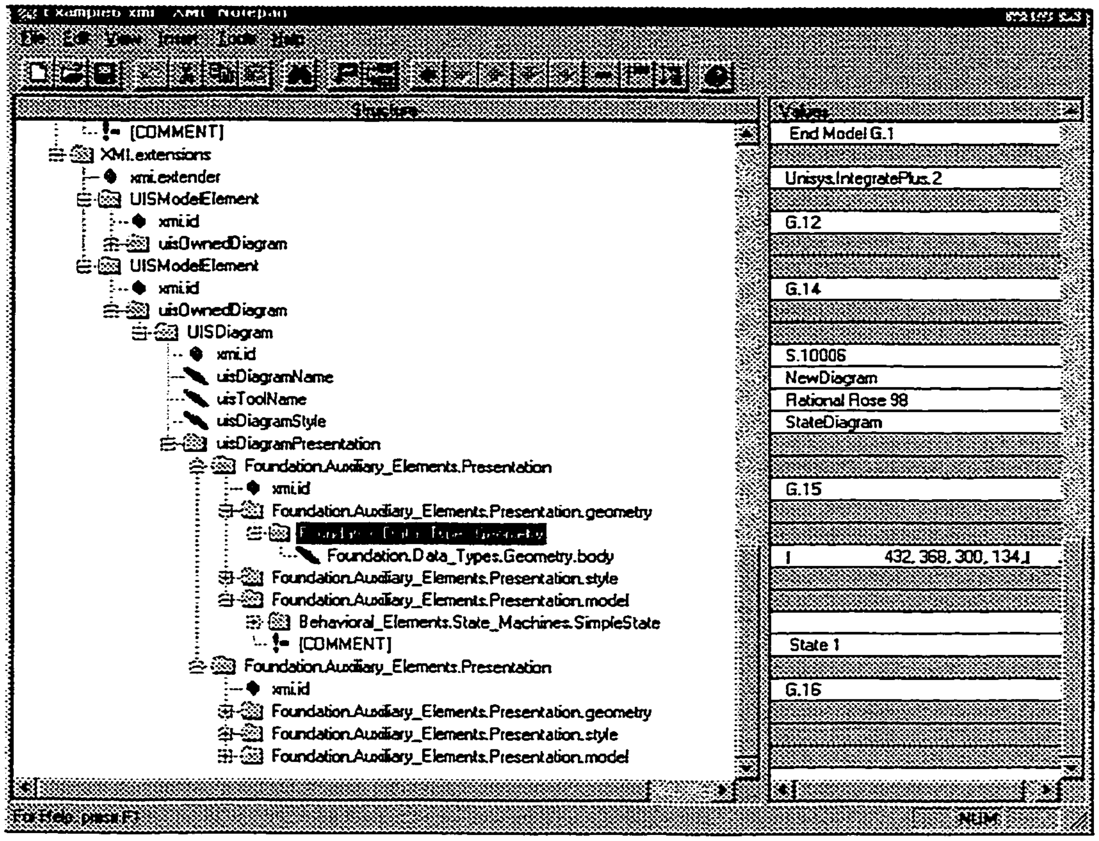
```
XMI.content                                                         
  Model_Management.Model
    xmi.id                                                          G.1
    Foundation.Core.ModelElement.name                               Example6
    Foundation.Core.ModelElement.visibility
    Foundation.Core.GeneralizableElement.isRoot
    Foundation.Core.GeneralizableElement.isLeaf
    Foundation.Core.GeneralizableElement.isAbstract
    XMI.extension
    Foundation.Core.Namespace.ownedElement
      Foundation.Core.Class
        xmi.id                                                      S.10001
        Foundation.Core.ModelElement.name                           NewClass
        Foundation.Core.ModelElement.visibility
        Foundation.Core.GeneralizableElement.isRoot
        Foundation.Core.GeneralizableElement.isLeaf
        Foundation.Core.GeneralizableElement.isAbstract
        Foundation.Core.Class.isActive
        XMI.extension
        Foundation.Core.ModelElement.namespace
        Foundation.Core.ModelElement.taggedValue
        Foundation.Core.Namespace.ownedElement
          Behavioral_Elements.State_Machines.StateMachine
            xmi.id                                                  S.10002
            Foundation.Core.ModelElement.name                       State/Activity Model
            Foundation.Core.ModelElement.visibility
            Behavioral_Elements.State_Machines.StateMachine.context
            Behavioral_Elements.State_Machines.StateMachine.top
            Behavioral_Elements.State_Machines.StateMachine.transitions
    [COMMENT]                                                       End Model G.1
  XMI.extensions
```

Figure 26 XMI Content Containing an Overview of the Model Information



```
Behavioral_Elements.State_Machines.StateMachine
  xmi.id                                                            S.10002
  Foundation.Core.ModelElement.name                                 State/Activity Model
  Foundation.Core.ModelElement.visibility
  Behavioral_Elements.State_Machines.StateMachine.context
  Behavioral_Elements.State_Machines.StateMachine.top
    Behavioral_Elements.State_Machines.CompositeState
      xmi.id                                                        G.7
      Foundation.Core.ModelElement.name
      Foundation.Core.ModelElement.visibility
      Behavioral_Elements.State_Machines.CompositeState.isConcurrent
      Behavioral_Elements.State_Machines.CompositeState.substate
        Behavioral_Elements.State_Machines.SimpleState
          xmi.id                                                    G.2
          Foundation.Core.ModelElement.name                         State 1
          Foundation.Core.ModelElement.visibility
          Behavioral_Elements.State_Machines.StateVertex.outgoing
          Behavioral_Elements.State_Machines.StateVertex.incoming
          Behavioral_Elements.State_Machines.State.entry
          Behavioral_Elements.State_Machines.State.exit
        Behavioral_Elements.State_Machines.SimpleState
  Behavioral_Elements.State_Machines.StateMachine.transitions
    Behavioral_Elements.State_Machines.Transition
      xmi.id                                                        G.3
      Foundation.Core.ModelElement.name
      Foundation.Core.ModelElement.visibility
      Behavioral_Elements.State_Machines.Transition.source
      Behavioral_Elements.State_Machines.Transition.target
    Behavioral_Elements.State_Machines.Transition
    Behavioral_Elements.State_Machines.Transition
[COMMENT]
```

Figure 27: XMI Content Containing Detailed View of States and Transitions



```
Structure                                                        Values
   [COMMENT]                                                     End Model G.1
 XMI.extensions
   xmi.extender                                                  Unisys.IntegratePlus.2
   UISModelElement
     xmi.id                                                      G.12
     uisOwnedDiagram
   UISModelElement
     xmi.id                                                      G.14
     uisOwnedDiagram
       UISDiagram
         xmi.id                                                  S.10006
         uisDiagramName                                          NewDiagram
         uisToolName                                             Rational Rose 98
         uisDiagramStyle                                         StateDiagram
         uisDiagramPresentation
           Foundation.Auxiliary_Elements.Presentation
             xmi.id                                              G.15
             Foundation.Auxiliary_Elements.Presentation.geometry
               Foundation.Data_Types.Geometry
                 Foundation.Data_Types.Geometry.body             |        432, 368, 300, 134 |
             Foundation.Auxiliary_Elements.Presentation.style
             Foundation.Auxiliary_Elements.Presentation.model
               Behavioral_Elements.State_Machines.SimpleState
               [COMMENT]                                         State 1
           Foundation.Auxiliary_Elements.Presentation
             xmi.id                                              G.16
             Foundation.Auxiliary_Elements.Presentation.geometry
             Foundation.Auxiliary_Elements.Presentation.style
             Foundation.Auxiliary_Elements.Presentation.model
For Help, press F1                                               NUM
```

Figure 28: XMI Extensions Containing View Information

6.2.2 Parsing Mechanism

An XML parser constructs a tree model of an XML document in memory using the Document Object Model (DOM) standard [DOM00]. This allows the document tree to be parsed, navigated, and processed using a well defined set of interfaces. A document consists of a set of nodes that describe the elements, text, comments, processing instructions, CDATA sections, entity references and declarations, notational declarations, and even entire documents [BRAD00]. The methods used in the design of the XMI Parser follow a specific pattern. The goal is to create the DOM tree, navigate it, extract the required elements, and create the object model representing the statechart (i.e., StateModel). First, a Document object is obtained through a factory method of the DocumentBuilderFactory class. This method, called 'newDocumentBuilder()', is used to create a DocumentBuilder object. The DocumentBuilder has a method 'parse()' which is used to parse the XML file and create the Document object, as illustrated below:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
factory.setNamespaceAware(true);

try
{
        // get a instance of a builder, and use it to parse the specified file

        DocumentBuilder builder = factory.newDocumentBuilder();

        // document is an instance variable of the XMIParser class for a
        // Document object
        document = builder.parse( new File(filename) );
}
```

Next, the DOM tree is navigated using the method 'getChildNode()' and examining its children. For example, the method 'getNodeList()' from the Node interface generates a list of child nodes. Each of these child nodes may be a different type (e.g., Element, Entity, Text, etc...). Based on the type of node, the context (location in the tree), and the DTD, we can determine how to process a given node. The following shows how a CompositeState object is created:

```
public void createCompositeState(Node node)
{
        NodeList list = node.getChildNodes();
        int size = list.getLength();
        int idx = 0;
        String name="";
        for (int i = 0; i < size; i++)
        {
                Node state = list.item(i);
                if (state instanceof Element)
                {
                    top = new CompositeState(getNodeAttribute(state,ID));
                    top.setStateName("Top state");
                    top.setParent(null);

                    // Continue processing DOM tree...
                    getCompositeState(state, top);
                    System.out.println("Top is " + top.getName());
                }
            }
        }
```

In this way, the entire tree can be processed using a common mechanism. Since each node in the tree may require slightly different specialized processing, several methods are used (as shown in Figure 29).

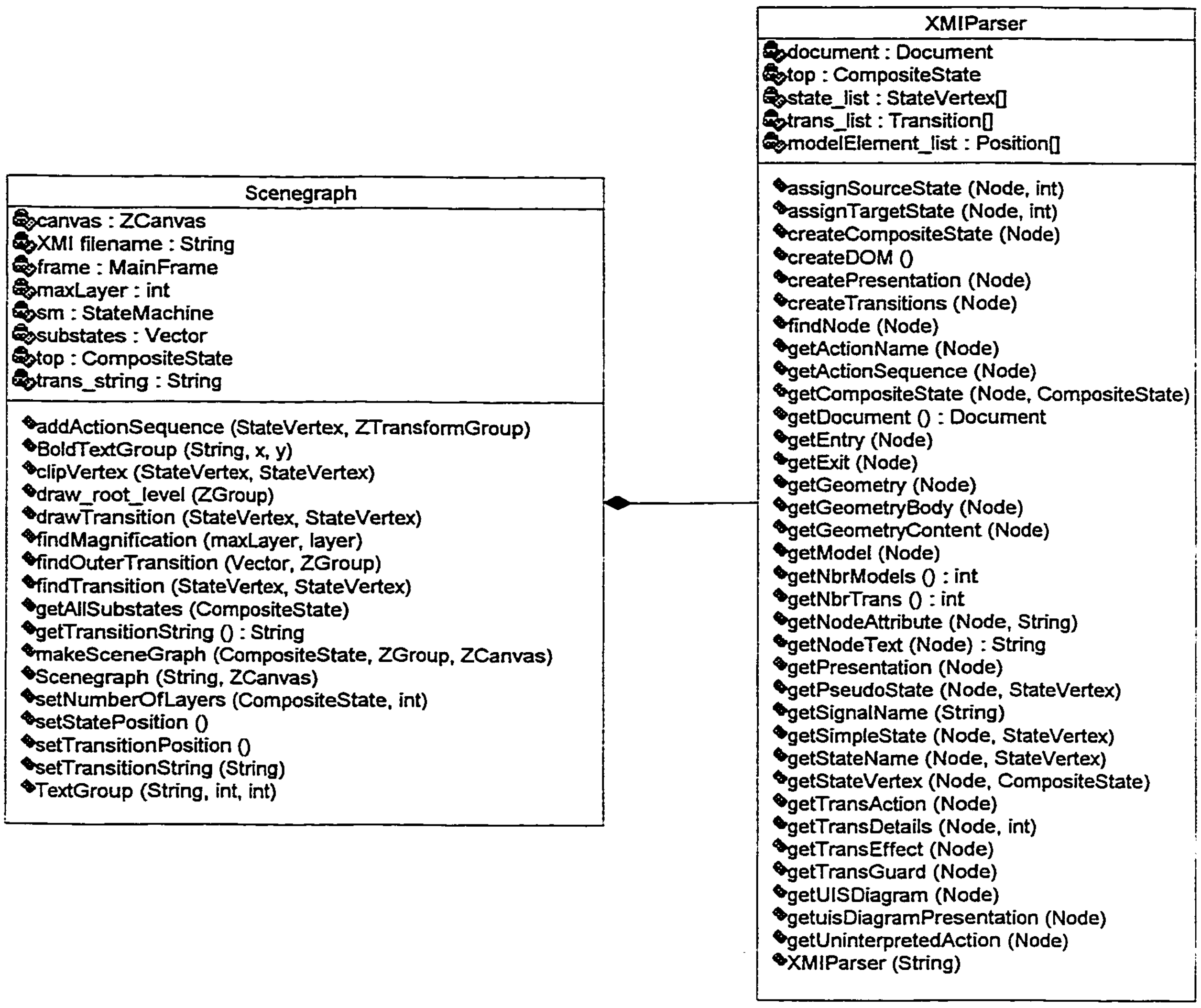

Scenegraph
canvas : ZCanvas
XMI filename : String
frame : MainFrame
maxLayer : int
sm : StateMachine
substates : Vector
top : CompositeState
trans_string : String
addActionSequence (StateVertex, ZTransformGroup)
BoldTextGroup (String, x, y)
clipVertex (StateVertex, StateVertex)
draw_root_level (ZGroup)
drawTransition (StateVertex, StateVertex)
findMagnification (maxLayer, layer)
findOuterTransition (Vector, ZGroup)
findTransition (StateVertex, StateVertex)
getAllSubstates (CompositeState)
getTransitionString () : String
makeSceneGraph (CompositeState, ZGroup, ZCanvas)
Scenegraph (String, ZCanvas)
setNumberOfLayers (CompositeState, int)
setStatePosition ()
setTransitionPosition ()
setTransitionString (String)
TextGroup (String, int, int)
XMIParser
document : Document
top : CompositeState
state_list : StateVertex[]
trans_list : Transition[]
modelElement_list : Position[]
assignSourceState (Node, int)
assignTargetState (Node, int)
createCompositeState (Node)
createDOM ()
createPresentation (Node)
createTransitions (Node)
findNode (Node)
getActionName (Node)
getActionSequence (Node)
getCompositeState (Node, CompositeState)
getDocument () : Document
getEntry (Node)
getExit (Node)
getGeometry (Node)
getGeometryBody (Node)
getGeometryContent (Node)
getModel (Node)
getNbrModels () : int
getNbrTrans () : int
getNodeAttribute (Node, String)
getNodeText (Node) : String
getPresentation (Node)
getPseudoState (Node, StateVertex)
getSignalName (String)
getSimpleState (Node, StateVertex)
getStateName (Node, StateVertex)
getStateVertex (Node, CompositeState)
getTransAction (Node)
getTransDetails (Node, int)
getTransEffect (Node)
getTransGuard (Node)
getUISDiagram (Node)
getuisDiagramPresentation (Node)
getUninterpretedAction (Node)
XMIParser (String)

Figure 29: Detailed Class Diagram for XMI Parser and Scenegraph

6.2.3 State Model Package

The State Model is responsible for containing the model and view information for the system. It contains the structural components (classes, relations, etc...) for the state machine. It is based on Argo/UML, which is an Open Source object oriented design tool for UML models. The following modifications have been made to Argo/UML to support the work of this thesis:

- Addition of Position information in the state vertex class to support the efficient layout of states, pseudostates, transitions, etc.

- Addition of magnification information along with accessor and mutator methods in the state vertex class to support the zooming mechanism.

- Removal of information that may be important in a real-life application but reduces the clarity of the model (e.g., fireVetoableChange()).

- Merging of information from other packages, such as the Common_Behavior and the Foundation, into the State_Machine package in order to simplify the design. Argo/UML deals with the entire UML metamodel, however, we are focusing on only the State_Machine package.

One of the advantages of using Argo/UML is that is based on the UML metamodel, which correlates well with the XMI package from the OMG. Figure 23 shows the class diagram for the State Model. This is based on the UML metamodel for state charts.

A brief overview of this model is presented here. For a more detailed explanation, please refer the UML reference document [OMG99a].

- **State Machine** (class) is at the root of the state hierarchy. It consists of a context (e.g., class, package, or system) that represents the owner. Since a state machine 'owns' its transitions, it has methods that adds, removes, and sets transitions. It also contains methods to set the root of the state hierarchy.

- **State Vertex** (class) is a node in a graph of states. It contains the common behavior of all types of states (e.g., composite state, simple state, pseudostates, etc…). Its main responsibility is to manage the association between the states and their attached transitions. It contains Position information (discussed later) which tells a state where to render itself and how big it should be.

- **Transitions** (class) are associations between a source and a target state. They also manage information regarding the associated triggers (*events*), guards (*conditions*), and effects (*actions*). This information is represented as a String of characters.

- **State** (class) is derived from the StateVertex. It contains the methods to manage the internal transitions, state variables, deferred events, and entry/exit actions. In addition, it can manage the associations between the state and the state machine.

- **Composite state** (class) is a container of other states vertices such as simple states, composite states, pseudostates, and stubbed states. The composite state follows the *composite design pattern* where the composite acts as a container of simple and composite objects. It contains methods to manage its contained substates and indicate whether a state is running sequentially or concurrently. Only sequential states are implemented in the program, since Rational Rose XMI export does not support concurrent state diagrams.

- **Simple state** (class) is a state that has no substates. It inherits attributes and methods from the State Vertex class.

- **Pseudostate** (class) uses the 'kind' attribute to determine what type of state it is. Several kinds of states may be differentiated: initial, final, shallow history, deep history, join, fork, junction, and choice. Only the *initial*, *final*, *shallow history*, and *deep history* pseudostates are implemented in this implementation. The other types of pseudostates are considered to be extensions to the program.

- **Position** (class). Each state contains a reference to the position (x, y, width, height) of the state on the screen. This is done to improve the performance of the system. A Position object is stored as an attribute in the StateVertex class. The Transition makes use of this position information when rendering the transition. It looks up the attached source and target state to determine how to render the transition.

6.2.4 Visual Components Package

The Visual Components package contains a number of ZVisualComponents that correspond to the viewable elements in a Jazz scenegraph. The classes ZInitialState, ZFinalState, ZCurvedLine (for transitions), ZHistory, ZDeepHistory, and ZSelfTransition extend the ZVisualComponents class. These implement the methods `paint()`, `pick()`, and `computeBounds()`. As mentioned in Chapter 4 (page 33), the `paint()` method is used to render the object, the `computeBounds()` is used to find the size of the object, and the `pick()` method tells whether an object has been picked or not.

6.2.5 Scenegraph Package

The Scenegraph package contains the classes required for creating a scenegraph, which is a hierarchical (tree-like) grouping of visual components.

- **Scenegraph** (class) contains the methods necessary to construct a scenegraph. It first calls the method makeSceneGraph(). This is a recursive function that uses a layered approach [FBL96] to constructing a scenegraph. This has three parameters:
    - The current root StateVertex of the state model. First, the root is the root of the state hierarchy. As soon as the method detects a composite state, it calls itself recursively.
    - The group or layer to which the state will be added. This is used to place the objects in back-to-front order when rendering and to allow fading at each layer.
    - The canvas tells where the drawing will occur.

Next, the states are constructed (depending on the type of state). For composite states, the makeSceneGraph() is called recursively for each level in the state hierarchy. Thus, objects appearing closer to the user are added in front of other visual components. The algorithm constructs a scenegraph for the various visual components. To do this, it creates a visual component, adds it to a ZVisualLeaf object. The ZVisualLeaf object is added to a grouping node (e.g., ZTransformGroup). Finally, the group of objects is added to a ZFadeGroup, which permits the layer in the scenegraph to be faded in/out. The ZFadeGroup uses a method setMinMax() to set the minimum magnification level for that node in the scenegraph.

Finally, transitions are added to the scenegraph at the appropriate level. The methods used in the Scenegraph class are:

- TextGroup() and BoldTextGroup() are responsible for rendering adding a string of normal text and bolded text to the scenegraph. They encapsulate a ZTransformGroup node.

- ClipVertex() is responsible for clipping a transition (line) with the border of a source or target state.

- DrawTransition() is responsible for adding a directed (arrow) curved lines between two state vertices.

- FindMagnification() is responsible for determining the level at which a visual group of objects fade in or out.

- FindTransition() is responsible for finding all the transitions on a given layer in the state hierarchy.

- GetAllSubstates() is responsible for finding all the substates at a given layer in the state hierarchy.

- GetTransitionString() is responsible for constructing the text to be rendered on a transition.

- Scenegraph() is a constructor which calls the XMI parser and then generates a scenegraph.

- SetNumberOfLayers() is responsible for determining the number of layers in the state hierarchy.

- SetStatePosition() is responsible for placing the geometry (size and location) of the states in the StateVertex object.

- SetTransitionPosition() is responsible for determining if a transition is visible or not. It is used so that a transition only appears once in a state diagram, starting from the top most (closest) layer.

BoxLineClipper (class) is a helper class that assists the scenegraph generator in clipping a box to a line. This is used when constructing a Transition between two states.

6.2.5.1 Scenegraph Construction Example

Figure 30 shows a simple example of how a scene can be constructed. The figure is divided into three parts: 1) a legend, 2) a sample scenegraph, and 3) the resulting statechart diagram. The root of a statechart consists of a set of states and a set of transitions. First the state group is constructed. A fade group is used so that an entire branch of the scenegraph can be faded or shown as needed. This allows for individual fading of the state vertices (i.e., simple states, pseudostates, and composite states). Under each fade group, there is a grouping node. This allows the states to be treated as a unit. Each state can be positioned independently by means of a transform group node. In this example, state A is an initial pseudostate. It is positioned by applying the translate() method of a ZTransformGroup object. A Simple state (State B) is slightly more complex. The state's border is rendered as a rounded rectangle. However, the name of the state requires extra work. First, a fade group allows the state's name to be faded out at a minimum level using the method setMinMag() in the ZFadeGroup class. Since the state's name does not depend on the zooming level, we use a sticky group (ZstickyGroup) to handle this case. Finally, the text itself can be positioned relative to the state's border. For this, we use a TextGroup, which is in fact a ZTransformGroup. Finally, the name of the state is attached to the TextGroup.

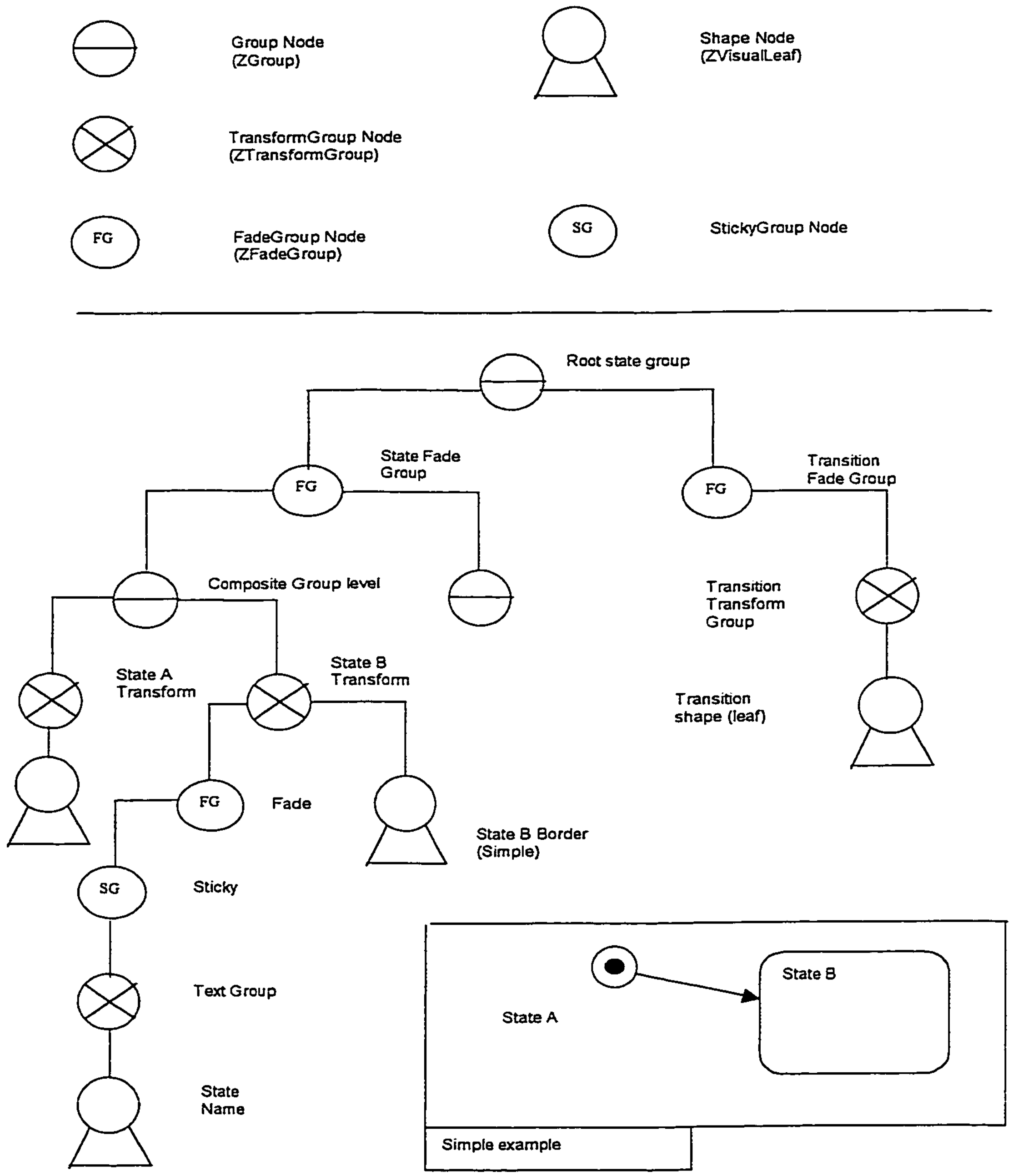
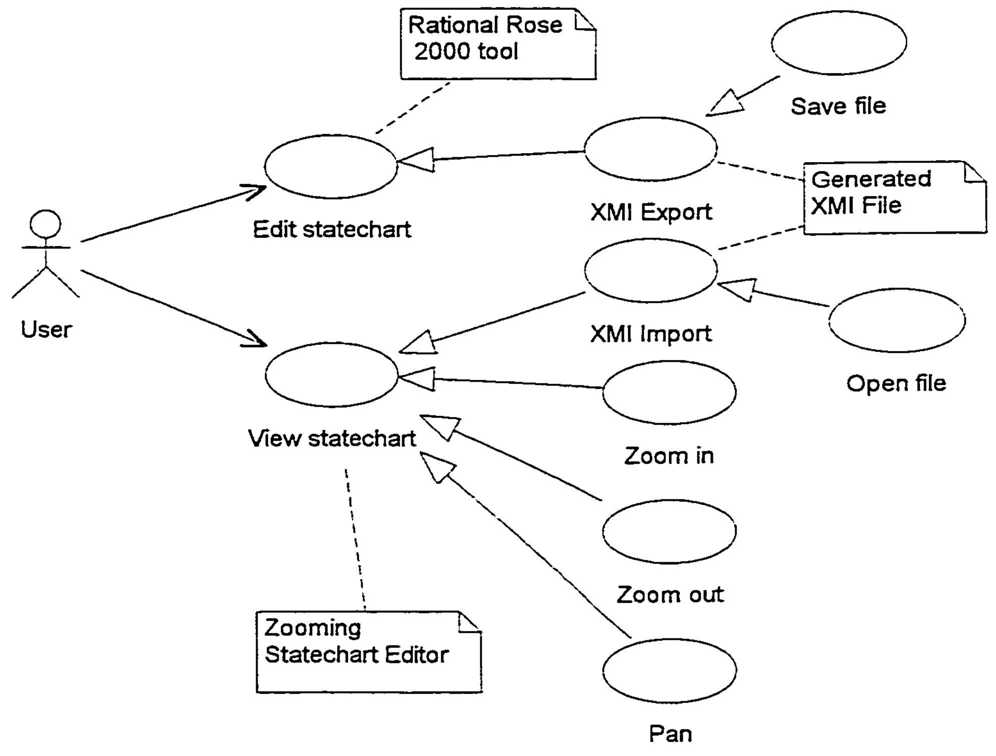


Figure 30: A Scenegraph Example [MC97]

6.3 Behavioral Model

6.3.1 Use Case view

When modeling a system, it is desirable to view the system from the user's point of view. UML provides a notation called Use Cases to show the intended uses of a system. The advantage of this method is that it gives the end user and the system architect a common understanding of what the system should accomplish. Figure 10 illustrates a Use Case diagram for the zooming statechart editor.



This diagram shows that the system comprises of two main activities: Editing a statechart and viewing a statechart. Editing a statechart is accomplished by using a commercial CASE tool such as Rational Rose 2000. The state diagram is created and modified with this tool.

Next, the data is exported to an XMI file. The XMI file consists of all of the model and view information necessary to render the statechart on the screen.

6.3.2 Scenarios

From a use case, various scenarios can be constructed. This section discusses various "interesting" scenarios that the statechart viewer undergoes. This helps to understand the system by examining its dynamic aspects. Scenarios can be represented in UML by an interaction diagram. There are two main types of interaction diagrams in UML: Sequence Diagrams and Collaboration Diagrams. For these examples, I have selected sequence diagrams to model some typical uses of the system. Collaboration diagrams are better suited for visualising systems with multiple concurrent flows of execution.

6.3.3 Object Interaction - Sequence Diagrams

6.3.3.1 Starting the application

Starting the application will generate a default scenegraph for a sample file. Figure 1 shows this type of interaction.

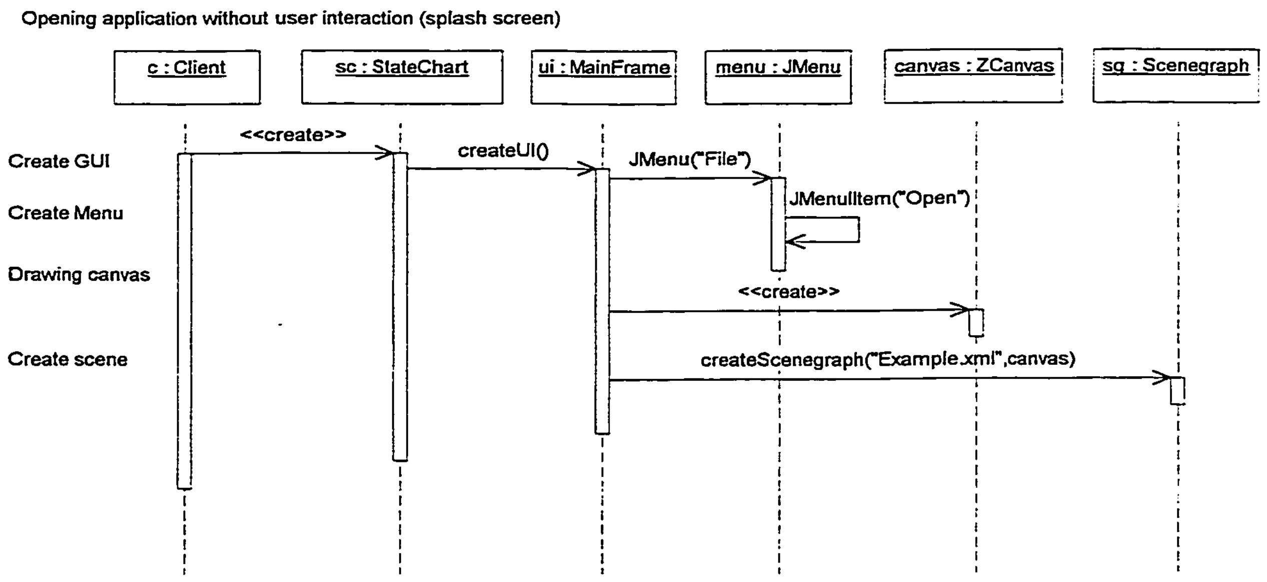


Figure 1: Sequence diagram for starting the application

6.3.3.2 Opening a new file

Figure 2 illustrates the dynamics involved with opening a file. Opening a file causes a file action object to be invoked. An action object is a listener as well as an action. This allows the file to be selected from the JFileChooser dialog box. Once selected, the scenegraph is created.

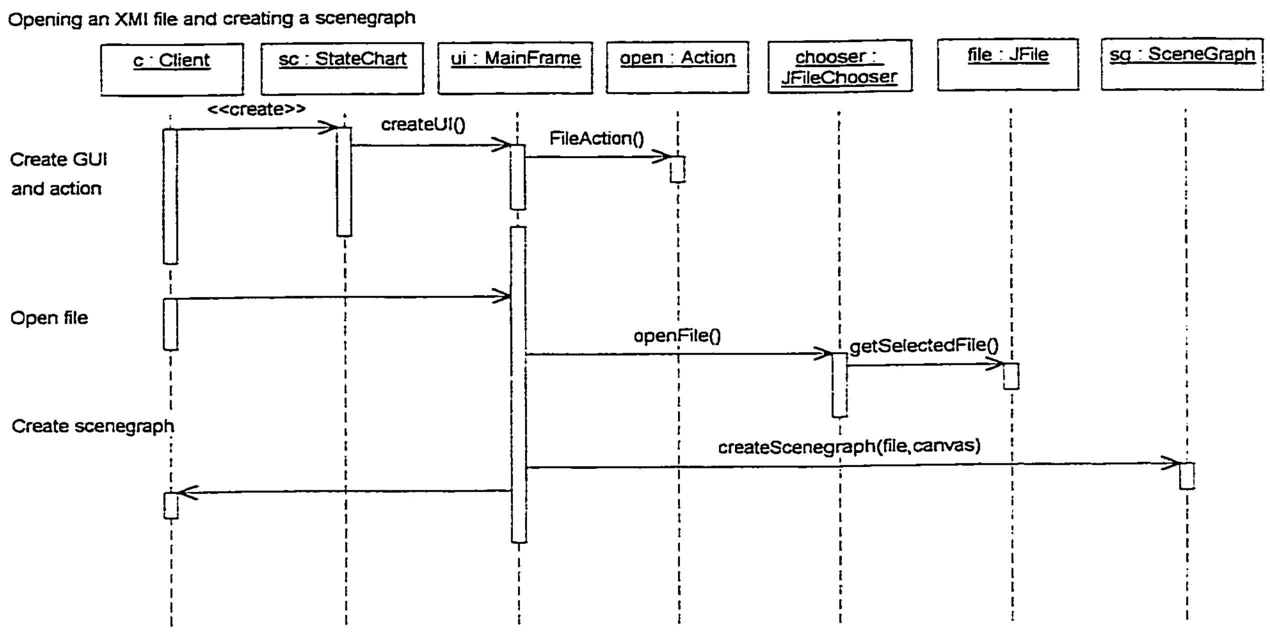


Figure 31: Opening an XMI file and Creating a Scenegraph

6.3.3.3 Interaction between SceneGraph and XMI Parser

This is a sequence diagram for the interaction between the objects in the SceneGraph and the XMI Parser. From the client object, the MainFrame is called. The details of the UI are omitted in this in this scenario. Assuming the user, when calling the createScenegraph method has created a file object, it causes the parser to be invoked. The XMI parser will, in turn, ask for a Document object. The document represents a Document Object Model (DOM) tree. Parts of the tree that we are interested in are the *transitions*, the *states* (composite), and the *presentation* (position, size, geometry, etc...) information. Once this is ready, we are able to ask the scenegraph to create itself. Finally, the scenegraph is added to the canvas, which is rendered by the Jazz framework.

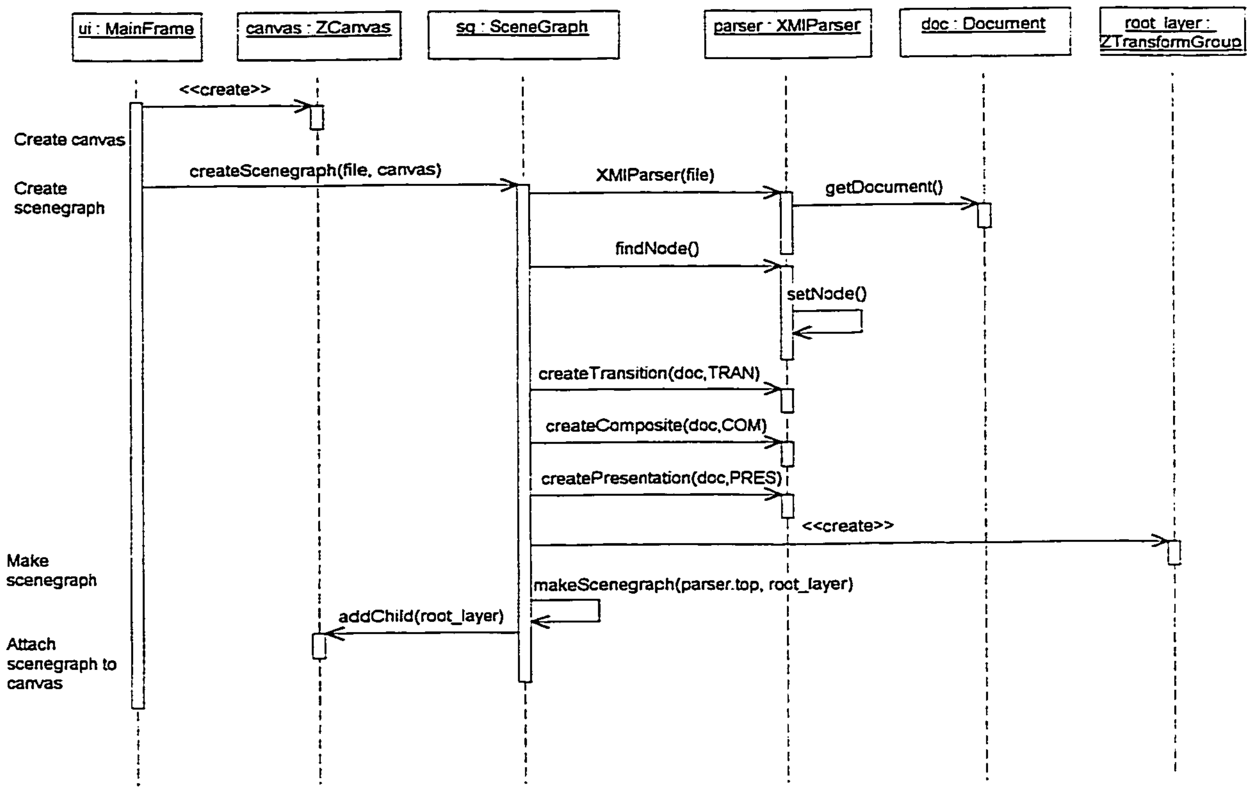


Figure 32: Object Interaction when Constructing a Scenegraph

6.3.4 Zooming Mechanism

The zooming mechanism is an important part of the StateChart Viewer. Jazz implements a default zooming event handler in a class called *ZoomingEventHandler*. This event handler responds to actions performed when pressing and dragging the right mouse button. In order to understand how zooming occurs, a statechart diagram shown in Figure 11 can be used to represent the behaviour of this class.

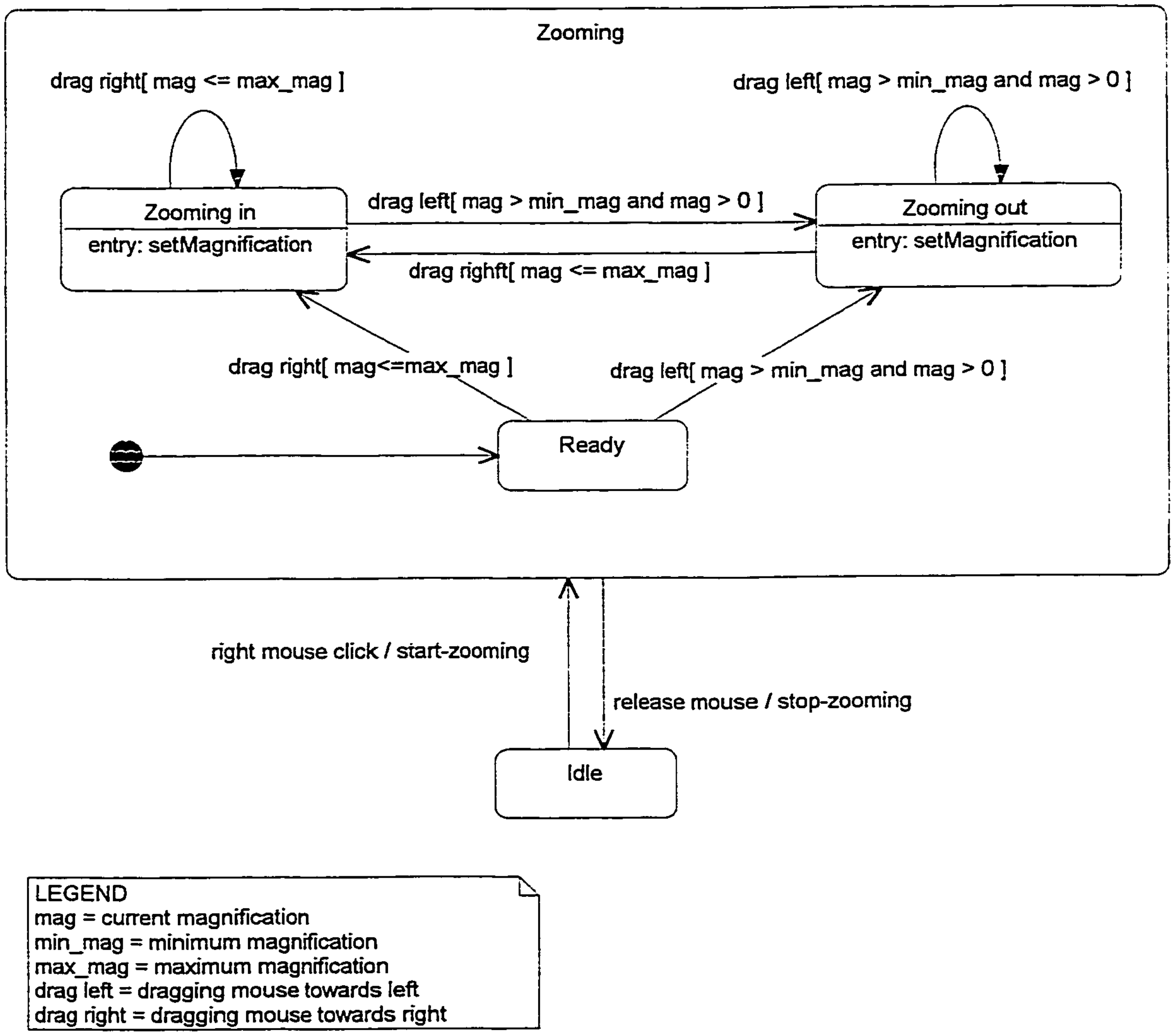

Zooming
drag right[mag <= max_mag]
drag left[mag > min_mag and mag > 0]
Zooming in
entry: setMagnification
drag left[mag > min_mag and mag > 0]
drag righft[mag <= max_mag]
Zooming out
entry: setMagnification
drag right[mag<=max_mag]
drag left[mag > min_mag and mag > 0]
Ready
right mouse click / start-zooming
release mouse / stop-zooming
Idle
LEGEND
mag = current magnification
min_mag = minimum magnification
max_mag = maximum magnification
drag left = dragging mouse towards left
drag right = dragging mouse towards right

Figure 33: Statechart for Zooming Mechanism

When the user presses the right mouse button, the system enters the *Zooming* state. This is composite state that contains the substates '*Zooming in*', '*Zooming out*', and '*Ready*' as well as the *initial* pseudostate. The first substate that is entered is the *initial* state. This state has an automatic transition to go into the *Ready* state. From this state, the system waits for the user to drag the mouse button right (to zoom-in) or left (to zoom-out). If the user, for example, drags the mouse towards the right with the mouse button down, the 'drag right' event will occur. If the current magnification is less than the maximum magnification, then the system will enter the *Zooming in* state. This will cause the entry action 'setMagnification' to update the magnification and the camera position will be updated. If another event 'drag right' occurs, then the system will exit this state momentarily and re-enter it causing the action 'setMagnification' to fire again. If, however, the user begins to drag the mouse to the left, then the 'drag left' event will occur, causing the state to change to *Zooming Out* if the current magnification doesn't exceed the minimum magnification or the current magnification is less than 0. Note that the entry action in both *Zooming in* and *Zooming out* cause the same 'setMagnification' method to be invoked. This method is responsible for updating the zooming magnification level and positioning the camera in the correct location.

6.4 Multiple representation of nested transitions (stubs)

There are actually two problems with nested transitions. First, how should we represent stubbed-states? Second, what layer to place the nested transition on so that it will be hidden or displayed. In order to understand how to represent stub states, we need to understand the goal of stub states. The goal is to be able to represent a 'hidden' view of the internal states so that we can view the model from a higher level. Often, it is desirable to hide the sub-states of a super-state. However, this causes a problem on what to do with the transitions that lead into/out of the states. Both of these problems require a solution that tells the transitions when it should be displayed, when it should be hidden, and how it should be displayed (regular arrow, stub on the incoming edge, stub on the outgoing edge, stub on both the incoming and outgoing edge). Figure 34 shows a sample screenshot of a composite state that is zoomed-in revealing its details. Figure 35 demonstrates the use of stubbed transitions when the composite state is zoomed-out.

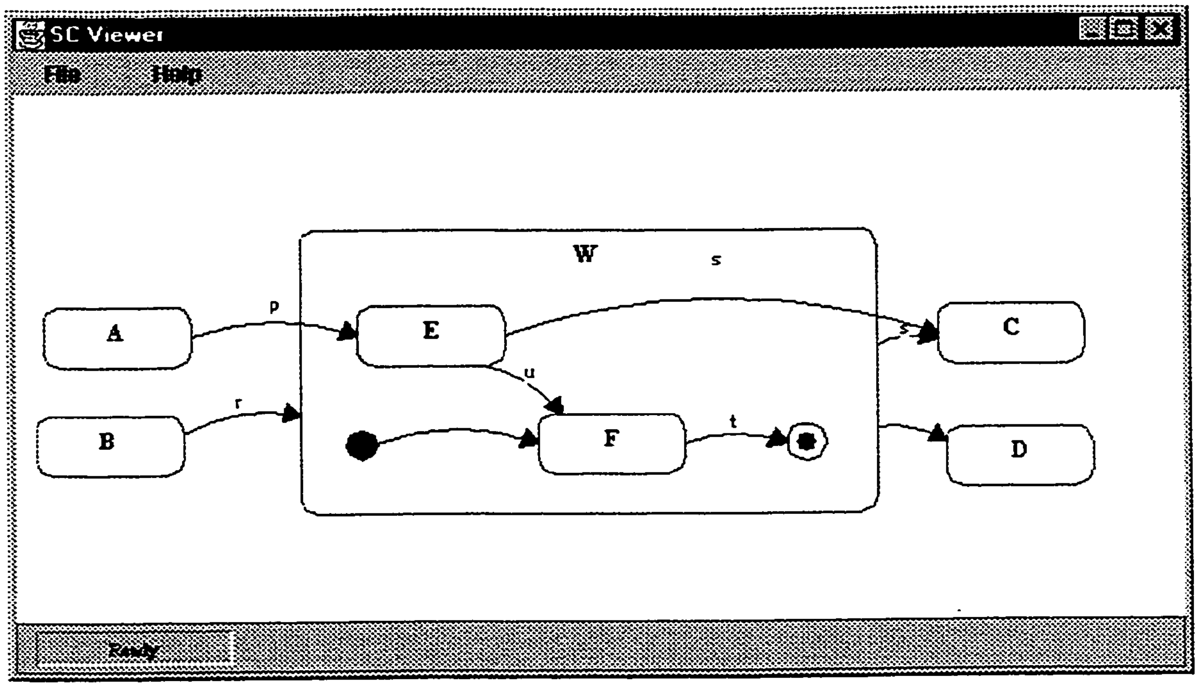


Figure 34: Detailed (Zoomed-in) View of Composite State [OMG99b]

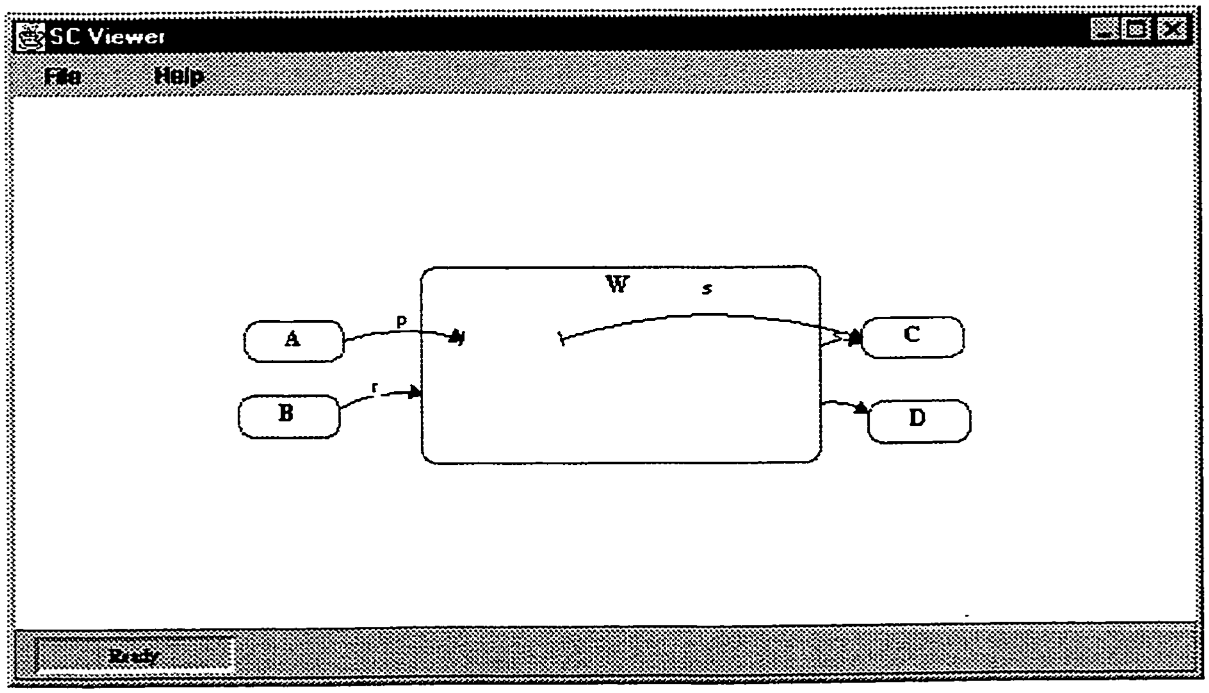


Figure 35: High-level (zoomed-out) View of Composite State with Stubbed Transitions [OMG99b]

The solution to the first problem of how to represent stubbed transitions is to realize that the underlying model does not change, only the view at which the transitions are represented. In other words, if we can use the geometry (position information) of the states to guide us, we can establish how to draw the stub states. In this thesis, we use a simple algorithm for drawing the stubbed states. A state's center position is established and its length and width is known. We clip either the incoming or outgoing transition to the boundary of the substate. A special transition is used to represent the small stubs (either incoming or outgoing). So that as each layer hidden from view, the transitions leading into/out of each layer must change depending on the position of its direct parent state and its magnification level. The magnification level depends on the level of the state in the state hierarchy.

The second problem deals with what layer to place the transition on so that as the containing states are concealed, all nested transitions should also be hidden. In order to solve this problem, when we request a state to be hidden, the composite state should query all of its 'owned' (contained) transitions. All transitions that are contained within the composite state will be hidden. The transitions leading to/out of a composite state will be visible.

6.4.1 Lowest Common Ancestor

Another way of thinking about hiding transitions is to use the lowest-common ancestor technique. From the point of view of the transition, this looks at the lowest common ancestor of its source and target states to determine whether the transition should be rendered or not.

Ex. Transition:

- Source: Root: Composite1: substate1: simplestate1
- Target: Root: Composite1: substate2: simplestate2

In this example, the lowest common ancestor is the Composite1. Therefore, only when the sub-states of Composite1 are hidden, the transition should also be hidden.

A problem with this is that a transition may have no knowledge of its path or it may not be very efficient to query each transition to determine what source and target states are in order to find the lowest-common ancestor.

The solution to this is to notice that this is the same as saying 'determine and hide all the transitions contained within a given state'. Thus, if we are looking from the state's point of view, all we need to do is to find out what transitions occur within a nested composite state. We can find this out easily by querying the composite state to determine what transitions are present in it. Once we have a list of transitions, we can simply hide them from the view.

6.4.2 Implementation in Jazz Scenegraph

It is desirable to implement the transitions in a Jazz scenegraph. Once in a scenegraph, the transitions can be easily be hidden or displayed as needed by using specialized group nodes for this purpose. For example, a ZFadeGroup allows visual components to be selectively faded or displayed based on the current magnification level. The difficulty is to know what layer to place the transition. When recursively drawing a state diagram, it is difficult to know where to place the transition in the scenegraph.

A possible solution is to use divide-and-conquer techniques to break the problem into sub-problems. A trivial problem occurs when two states are on a same layer and connected with a transition. In this case, the transition would go in the same layer as the contained substates. Figure 36 illustrates this example. A more complex problem occurs when the transitions connecting two states where the states occur on different layers in the scenegraph, as in Figure 37. In this case, we need to determine the lowest-common ancestor of the two states and insert the transition at this layer. The problem is that we do not know what layer to place the transition in while we are creating the scenegraph. When we create the scenegraph, we recursively add visual components (states, transitions) into it beginning with the root and following its children. Only after we are exiting the recursive call can we identify which layer the transition should appear.

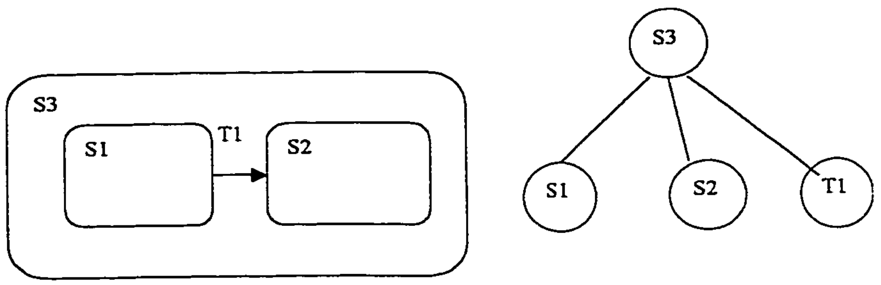


Figure 36: The Transition T1 is inserted in the same layer as S1 and S2 in the Scenegraph

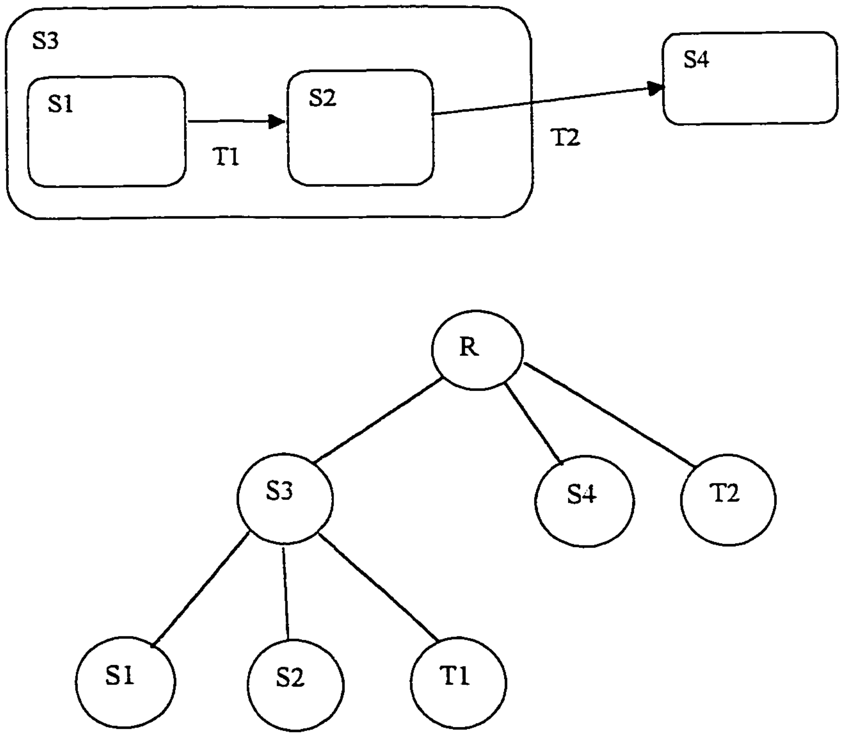


Figure 37: The Transition T2 is inserted in different layer in Scenegraph than T1

Chapter 7

DISCUSSION AND CONCLUSION

7.1 Introduction

This chapter discusses the issues involved during the design and implementation of the statechart viewer. It provides the information of what was done and what was learned. The achievements and limitations of the system are given. Finally, the work that needs to be done in the future is provided.

7.2 Design Issues

Statecharts were invented as a formal way to visualize the dynamic behaviour of a system. As systems become larger and more complex, statecharts helps to reduce this complexity in several ways. As shown in chapter 2, hierarchies of states can be constructed by embedding states inside one another and forming composite states. Once this is done, the composite state can be treated as a unit whose details can be hidden or exposed from view.

A state machine forms a tree-like structure of state vertices. The state vertices can be either simple or composite. It is fairly easy to represent this hierarchy of states on a 2D plane. The difficulty lies in how to represent transitions when a portion of the state diagram is hidden from view. For example, when a state appears on different levels in the state hierarchy in a state diagram, it is difficult to see how transitions can be hidden or shown from view. Knowing when a transition should be rendered is a key to understanding and implementing the zooming mechanism. Section 6.5.1 discusses this approach. The zooming mechanism allows the user to dynamically change the view of the statechart. Stubs are an example of context-sensitive rendering based on magnification level.

Constructing a scenegraph for statecharts using the Jazz framework is not an easy task. The scene is composed of a hierarchical grouping of states and transitions. It was important to be able to place the groups in the correct layers of the scenegraph so that the state could be displayed or shown as desired. Fade group nodes were used to provide a notion of fading in/out while zooming. Various recursive algorithms were used to create the scenegraph. The difficulty was in mapping the state model to the Jazz scenegraph, especially when it came to layering the states and transitions. Several layout issues became apparent, as follows:

1. Should an automatic layout algorithm to position states and transitions? It was decided, that the state would be positioned as in the original model (from the Rational Rose tool), while the transitions would be based on the position of the states. Also, the position of the text would depend on the position of the states. This was based on the fact that the information to position the transitions was missing from the XMI export[i]. Therefore, no layout automatic layout algorithm was used for states (only transitions).

2. Should we represent transitions as curves or straight lines? Curved lines were used since they were more pleasing to the eye and allow two edges to appear between states.

[i] This contraint actually helped improve the layout, since it was later decided to use curved lines

3. Should the textual elements be scaled? This is more a matter of personal taste, but it was decided not to scale the text in the state name and on the transition. Only the internal transitions would be scaled.

4. Where should the text on a transition appear? It was decided to position the text in the centre of the transitions after a review of the literature on edge label placement [KT97a, KT97b] problem.

5. How to represent seemingly simple graphical objects such as arrowheads and stubs? The basic Jazz framework's arrowheads were insufficient and required a custom solution. For this, some of Argo/UML's library was used (in particular, Graph Editing Toolkit (G.E.F) [ROBB00]). This consumed quite a lot of time because of the transformations involved.

7.3 Contributions

There are several contributions made by this research effort:

- Development of a statechart viewer for visualising UML statecharts
- Improved representation of statechart diagrams over commercial tools
- Ability to zoom and navigate a large statechart diagram
- Demonstrated the use of the Jazz framework
- Automatic layout of curved lines on transitions

- Construction of an XMI parser for statecharts

7.4 Limitations

Currently, the system provides no editing facilities, such as the ability to manipulate (move, resize, add, modify, delete) the states and transitions. Our original intention was to create an editor for statecharts. However, we preferred to concentrate on the zooming aspects of the statechart diagram, since there were already a large number of commercial editing tools available on the market.

Several advanced features of statecharts were not implemented:

- Concurrent states.
- Fork/Join of transitions.
- Send events.

These could be considered as future work and would be relatively simple to implement. Also, there is no automatic layout mechanism of the statechart diagram. The system is currently limited to five levels of depth.

There was no usability test done to verify the assumption that 'the system should be easier to view and understand' or that 'the navigation of the system is easy and intuitive' (see Abstract, p. iii). This could be considered future work.

7.5 Conclusions

XMI is a powerful standard for exchanging model information between software vendors. Since the XMI standard is based on UML and XML, model and view information can be extracted from a commercial vendor such as Rational Rose. It is rather straightforward to parse an XMI file because UML metamodel is well described and several XML parsers are available. A disadvantage of using Rational Rose is that it does not support the entire UML standard for statecharts (e.g., concurrent states are missing). In addition, the XMI exporter from Rational Rose is flawed and incomplete. For example, only the positions of the states are available. Various workarounds were necessary. For example, the Rational Rose tool does not export transitions consisting of polylines; the XMI export loses this information[i]. The work around for this was to base the position of the transitions on the position of the source and target states. At first, straight lines were used to connect source and target states. This caused a problem when there were multiple transitions (edges) between two states (vertices). The solution to this was to use curved lines to represent transitions. Other advantages of using curved lines is that they are esthetically pleasing to the eye, can better represent the dynamics of a system, and are better at representing Gestalt's principle of continuity [WARE00, p.206-207].

It is interesting to note that the system can be easily extended to include other kinds of UML diagrams, such as package diagrams. Since the basic infrastructure is in place, one would only need to add new visual components and make appropriate changes to the existing architecture

Jazz provides an excellent framework for structuring 2D graphics. Once the XMI file is parsed, and the state model is created, the scenegraph must be generated. The mapping between the state model and the Jazz scenegraph is the most difficult part and several layout issues emerge at this stage. It is important to understand the underlying semantics of the statechart when deciding what information to present while zooming. Visual elements can be

[i] This is a defect in the current Rational Rose tool.

easily hidden or displayed when requested by inserting a ZFadeGroup in the scenegraph and applying the appropriate transformations. However, we must decide at what level a substate becomes invisible, when a transition end becomes a stub, when a state name disappears, the number of levels we can support, how to render the various visual components, whether text is zoomable or not, etc... Jazz facilitates this task by providing scenegraph support and basic zooming capabilities; however, the designer and/or developer must assume the responsibility of understanding the requirements to make the project a success.

Appendix A: SAMPLE SCREENSHOTS

This section shows some sample screenshots of the StateChart Viewer. It is based on Harel's stopwatch example [HAR87].

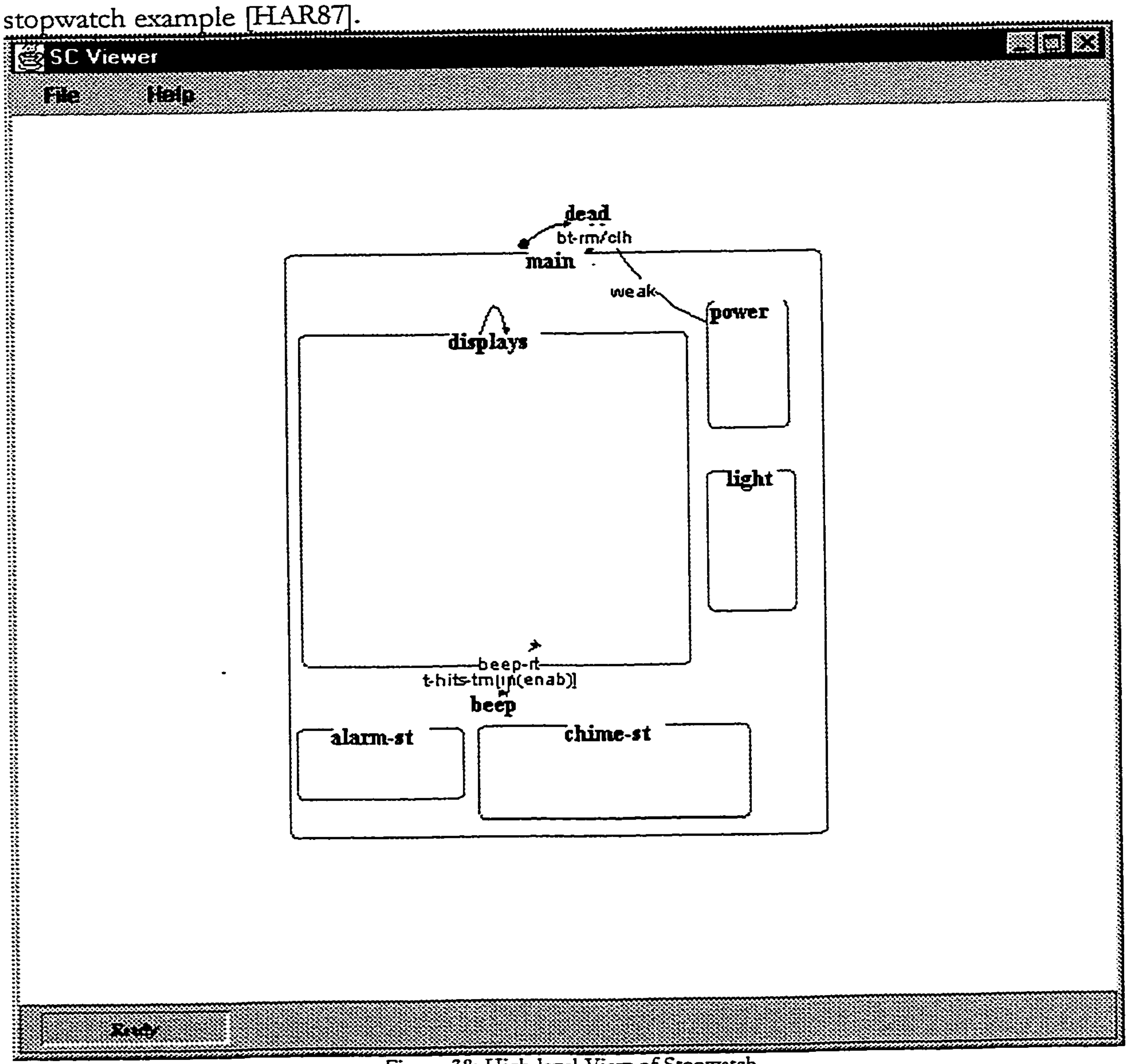


Figure 38: High-level View of Stopwatch

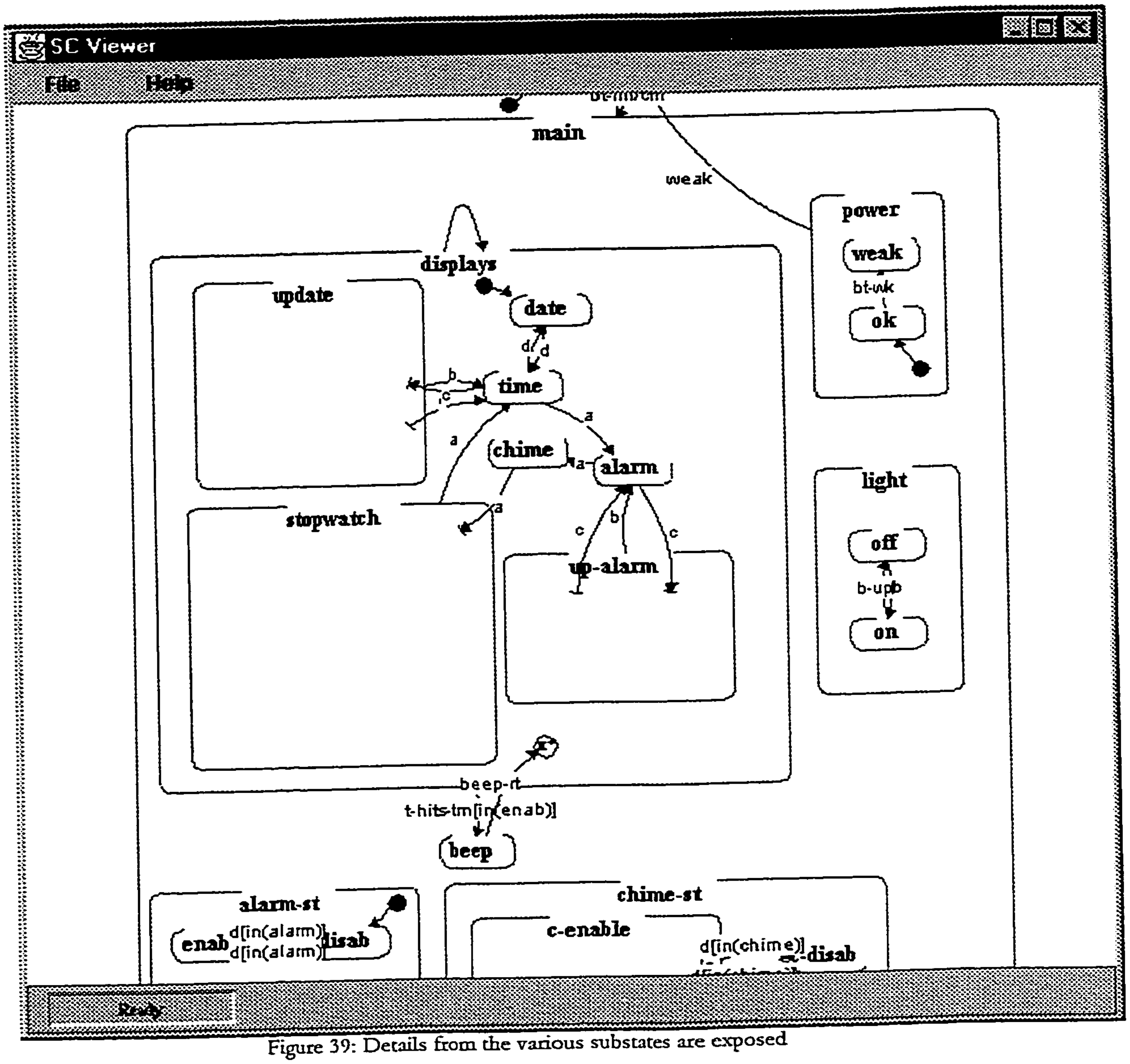


Figure 39: Details from the various substates are exposed

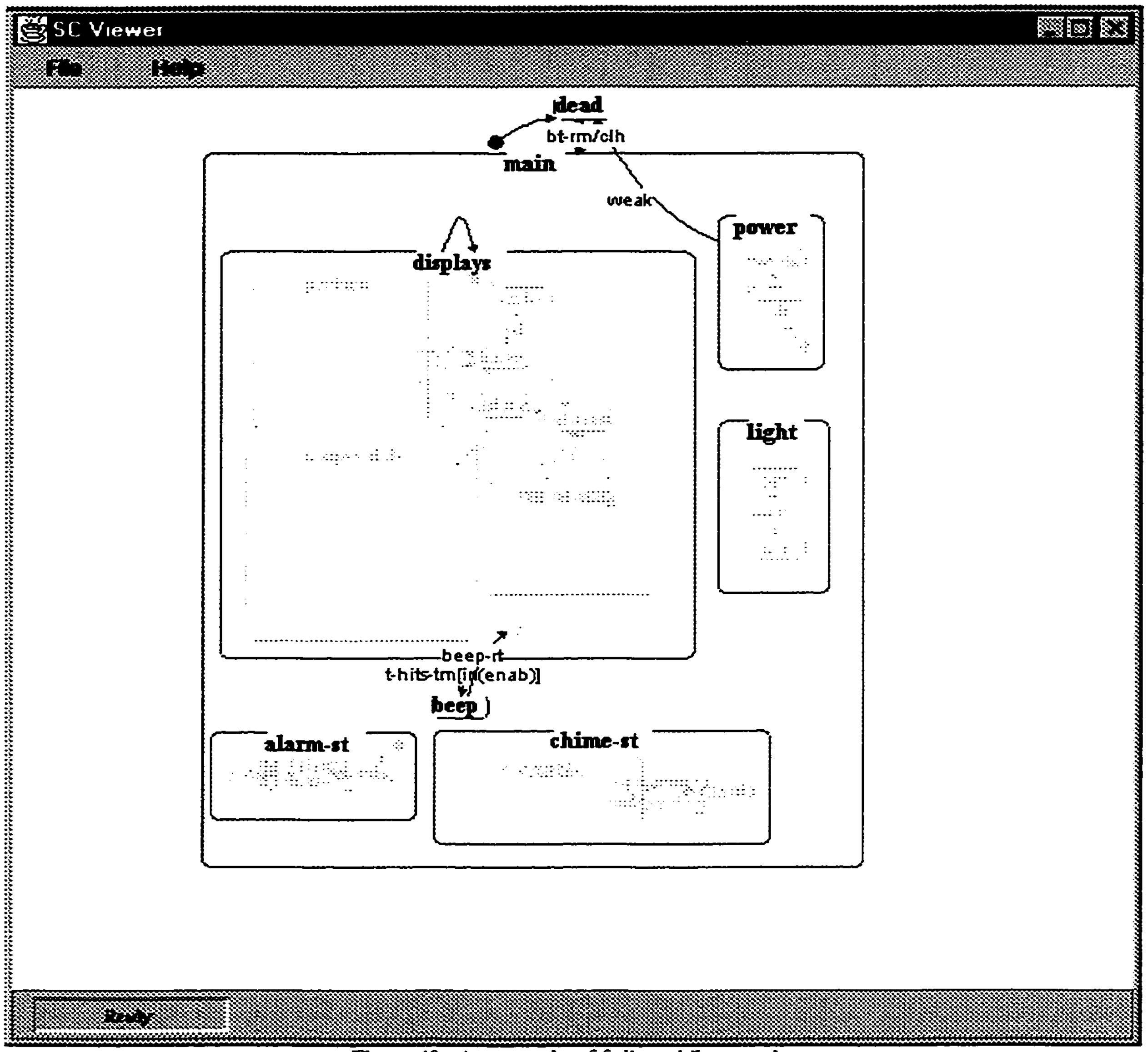


Figure 40: An example of fading while zooming

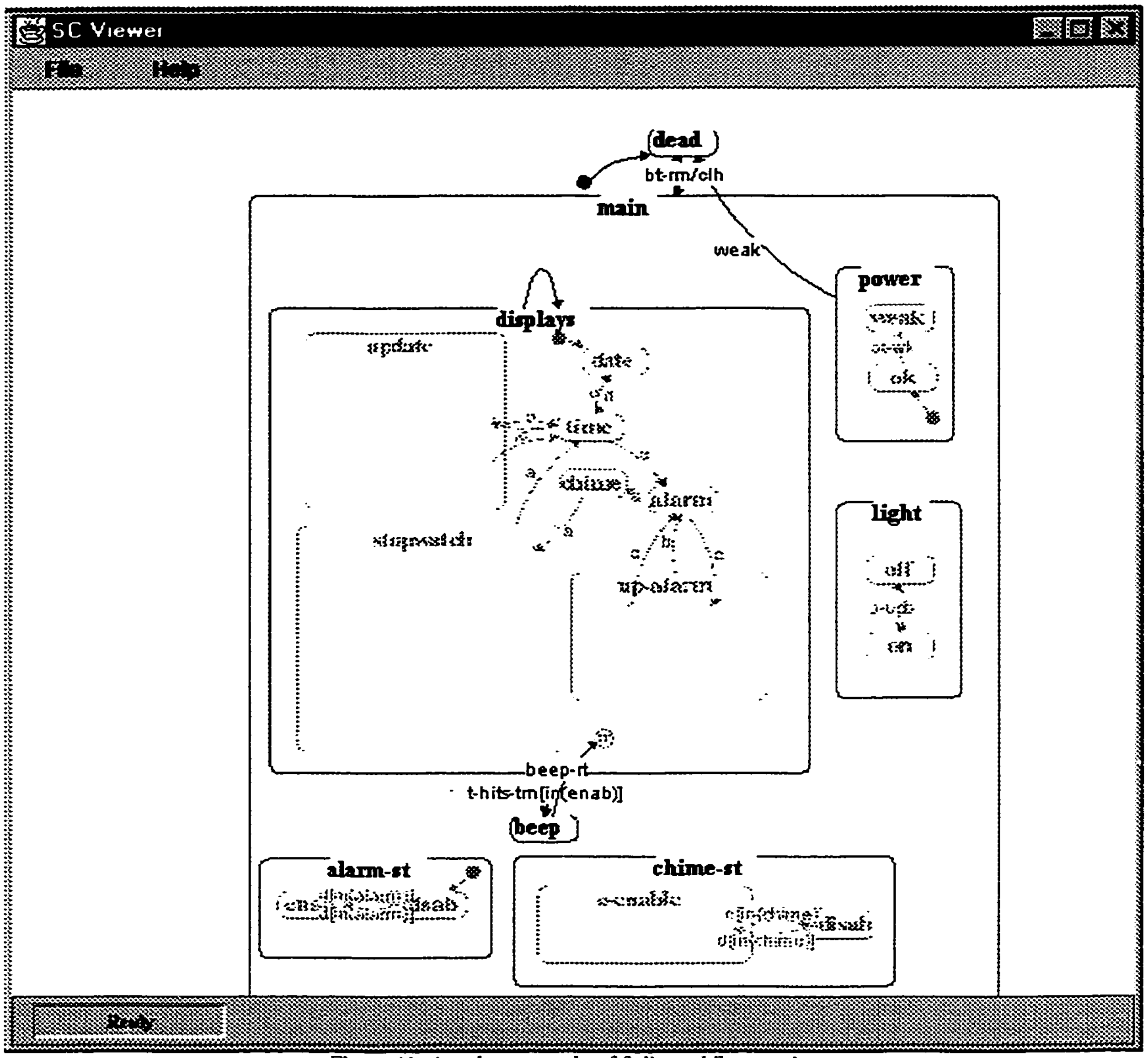


Figure 41: Another example of fading while zooming

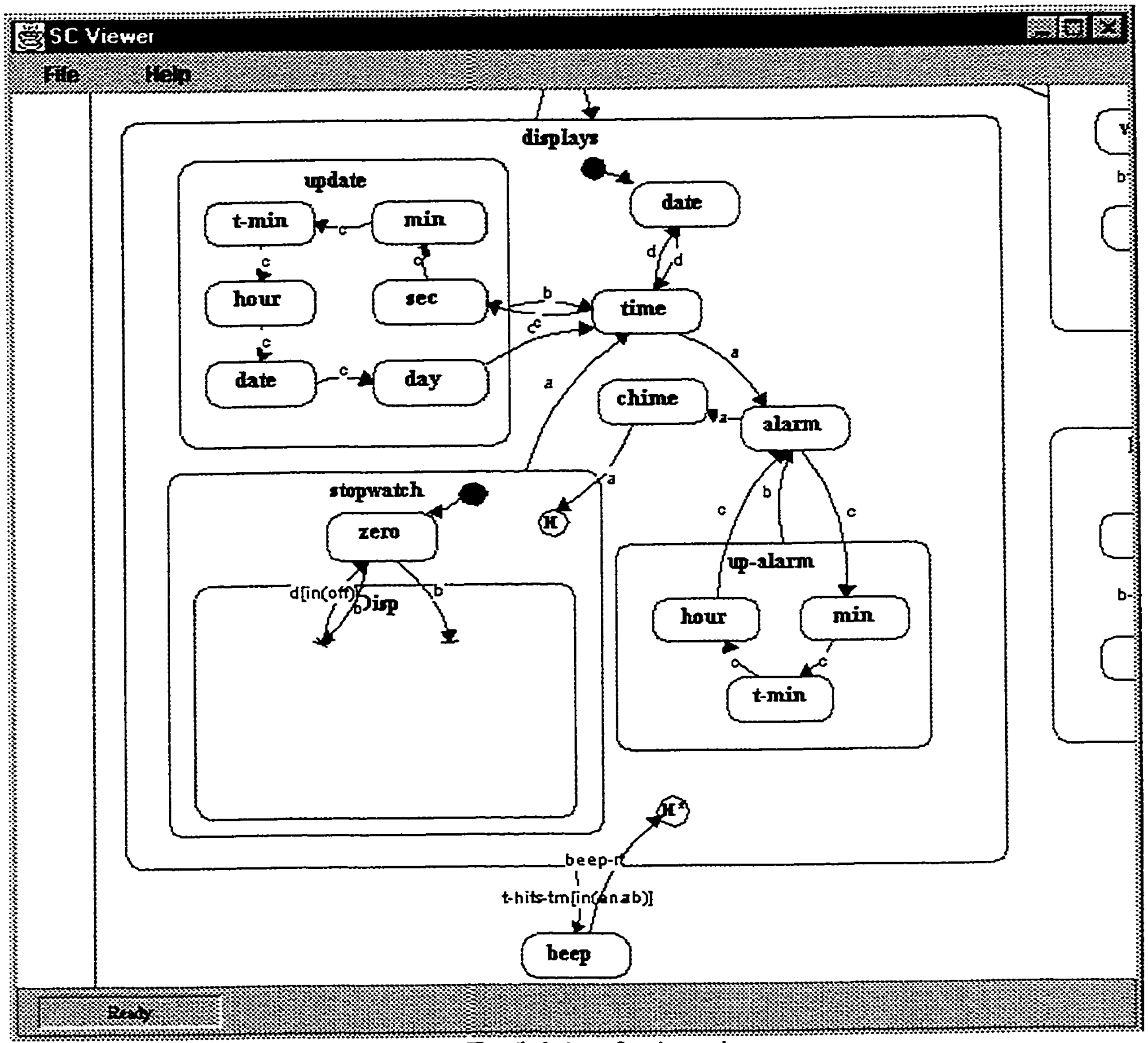


Figure 42: Detailed view of various substates

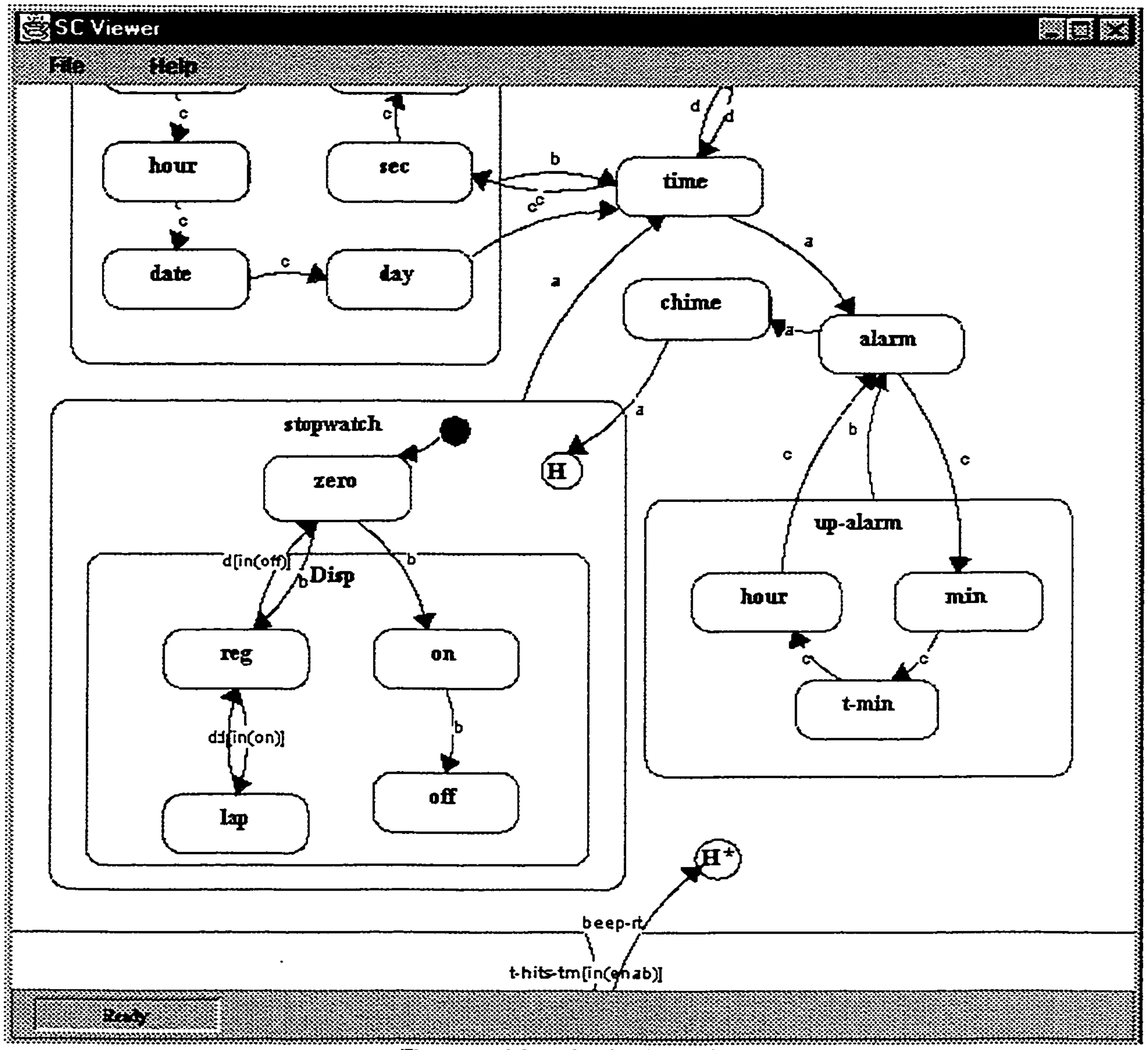


Figure 43: More details of Statechart

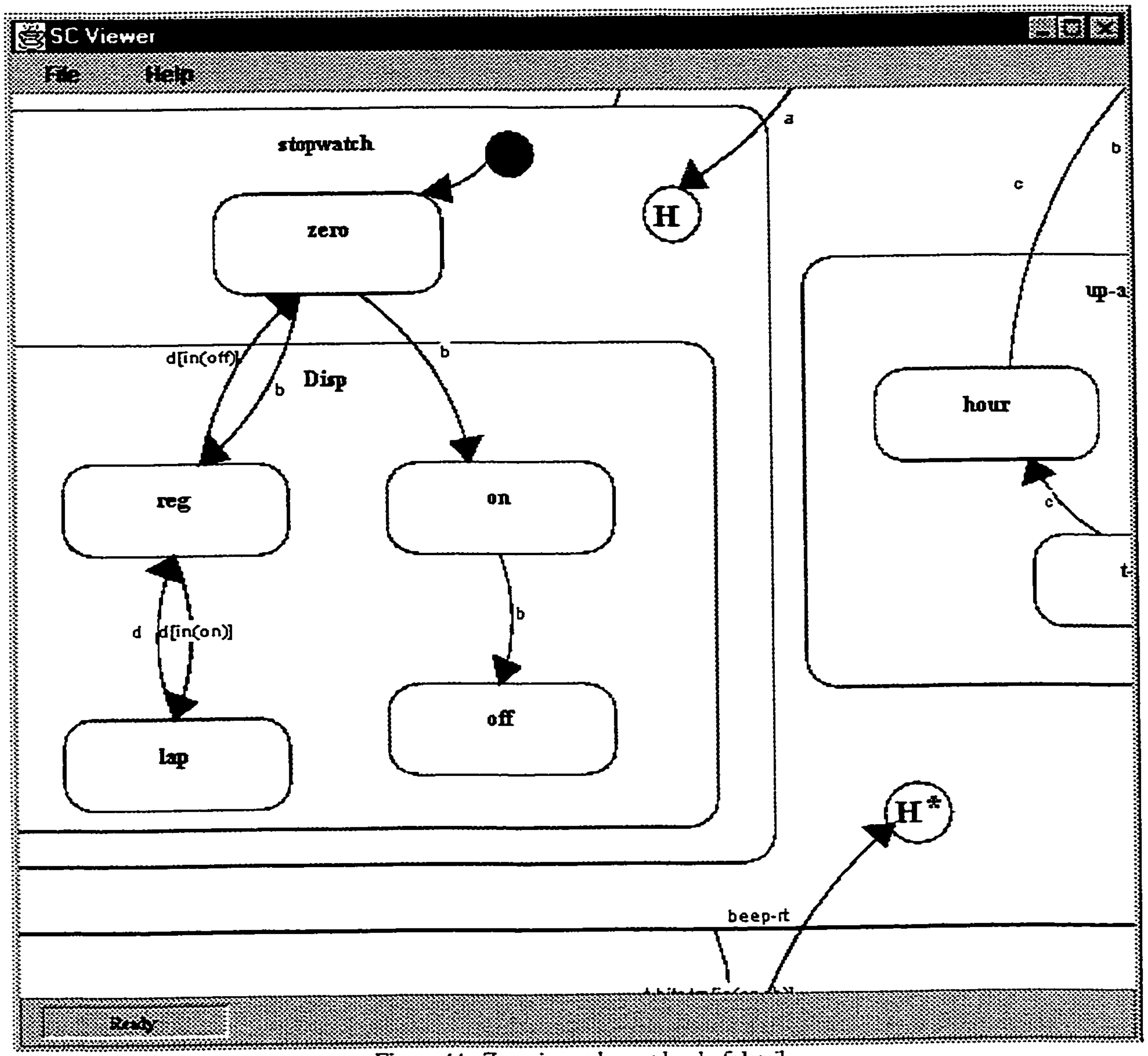


Figure 44: Zooming at lowest level of detail

GLOSSARY [BRJ99] [OMG99a]

Action. An action is an executable atomic computation that results in a change in state of the system or the return of a value.

Guard condition. A guard condition is a condition that must be satisfied in order to enable an associated transition to fire.

Statechart diagram. A statechart diagram is a diagram that shows a state machine, emphasizing the flow of control from state to state.

State machine. A state machine is a behaviour that specifies the sequence of states an object goes through in response to events, together with its responses to those events.

State. A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for an event.

State Vertex. A state vertex is an abstraction of a node in a statechart graph. In general, it can be the source or destination of any number of transitions.

Simple State. A simple state is a state that does not have substates. It is a child of State.

Composite State. A composite state is a state that contains other state vertices (states, pseudostates, etc.). The association between the composite and the contained vertices is a composition association. Hence, a state vertex can be a part of at most one composite state.

Pseudostate. A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. There are several types of pseudostates: initial, deepHistory, shallowHistory, join, fork, junction, and choice.

Initial State. An initial pseudostate represents a default vertex that is the source for a single transition to the *default* state of a composite state. There can be at most one initial vertex in a composite state.

Final State. A final state is a special kind of state signifying that the enclosing composite state is completed. If the enclosing state is the top state, then it means that the entire state machine has completed. A final state cannot have any outgoing transitions. A final state is a child of state.

Stub State. A stub state can appear within a submachine state and represent an actual sub-vertex contained within the referenced state machine. It can serve as a source or destination of transitions that connect a state vertex in the containing state machine with a sub-vertex in the referenced state machine. A stub state is a child of state.

Submachine State. A submachine state is a syntactical convenience that facilitates reuse and modularity. It is a shorthand which implies a macro-like expansion by another state machine and is semantically equivalent to a composite state.

Synch State. A synch state is a vertex used for synchronizing the concurrent regions of a state machine. A synch state is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can enter a particular state or states.

Event. An event is the specification of a significant occurrence that has a location in time and space. There are several kinds of events: Signals, CallEvent, ChangeEvent, TimeEvent,

Signals. The specification of an asynchronous stimulus communicated between instances.

Call Event. A call event represents the reception of a request to synchronously invoke a specific operation. The expected result is the execution of a sequence of actions that characterize the operation at a particular state.

Change Event. A change event models an event that occurs when an explicit Boolean expression becomes true as a result of a change in value of one or more attributes or associations.

Time Event. An event that denotes the time elapsed since the current state was entered.

Transition. A relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and conditions are satisfied.

REFERENCES

[BEDE00] Ben Bederson, *Jazz tutorial*, HCI lab, University of Maryland, URL: http://www.cs.umd.edu/hcil/jazz/, 2000

[BM99] Bederson, B., McAlister, B., *Jazz: An Extensible 2D+* Zooming *Graphics Toolkit in Java*, CS-TR-4014, UMIACS-TR-99-24, URL: http://www.cs.umd.edu/hcil/jazz/learn/papers, May 1999

[BMG00] Bederson, B., Meyer J., Good L., *Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java*, CS-TR-4137, UMIACS-TR-2000-30, http://www.cs.umd.edu/hcil/jazz/learn/papers, May 2000

[BP99] Brown, K., Petersen, D., *Ready-to-run Java 3D*, Wiley, ISBN 0-471-31702-0, pp.40-42, 1999

[BOOC94] Grady Booch, *Object-Oriented Analysis and Design with Applications, 2nd ed.*, Addison-Wesley, p.208, 1994

[BRAD00] Neil Bradley, *The XML Companion, 2nd Edition*, pp. 3, pp.261-286, 2000

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, 1999

[DOM00] World Wide Web Consortium, DOM standard, URL: http://ww.w3.org/DOM, 2000

[FBL96] Jean-Daniel Fekete and Michel Beaudouin-Lafond, *Using the Multi-Layer Model for Building Interactive Graphical Applications*, Proceedings of the UIST'96 Conf., Seattle, USA, p. 109-118, Nov, 1996

[FLAN97] David Flanagan, *Java in a Nutshell, 2nd edition*, O'Reilly, p. 1, 1997

[FURN86] George W. Furnas, *Generalized Fishey Views*, Human Factors in Computing Systems CHI'86 Conference Proceedings, 16-23, 1986

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, USA, 1994.

[HAR87] David Harel, *Statecharts: a visual formalism for complex systems*. Science of Computer Programming 8(1987), pp. 231-274

[HAR88] David Harel, *On visual formalisms*. Communications of ACM 31, 5 (May 1988), pp. 514-530

[IBM00] XMI Toolkit, IBM Alphaworks, URL: http://www.alphaworks.ibm.com, 2000

[JAZZ00] Jazz, URL: http://www.cs.umd.edu/hcil/jazz

[KNUS99] Jonathan Knusden, *Java 2D Graphics*, O'Reilly, 1999

[KT97a] K. G. Kakoulis and I. G. Tollis. *On the Edge Label Placement Problem* (Proc. GD'96), volume 1190 of Lecture Notes in Computer Science, pp. 241-256. Springer-Verlag, 1997

[KT97b] K.G. Kakoulis and I. G. Tollis. *An Algorithm for labeling edges of hierarchical drawings*, In G. DiBattista, editor, Graph Drawing 97, vol. 1353 of Lecture Notes in Computer Science, pp. 169-180. Springer-Verlag, 1997

[LRP95] Lamping, J., Rao, R., and Pirolli, P. A focus+context technique based on hyperbolic geometry for viewing large hierarchies. Proceedings CHI'95, ACM, pp.401-408, 1995

[MACR00] Macromedia, Flash, URL: http://www.macromedia.com

[MC97] Chris Marrin & Bruce Campbell, notation adapted from *Teach yourself VRML in 21 days*, p.17-18, 1997

[MICR00] Microsoft Corporation, Microsoft XML Notepad, URL: http://www.microsoft.com, 2000

[OLIV00] OLIVE (On-line Library of Information Visualization Environments), URL: http://www.otal.umd.edu/Olive

[OMG99a] OMG Unified Modeling Language Specification, URL: http://www.omg.com, UML reference (99-06-08), p2-129 to p2-158, 1999

[OMG99b] UML Notation Guide, UML v.1.3, URL: http://www.omg.com, 1999

[OMG99c] Object Management Group, XML Metadata Interchage, (XMI 1.0) URL: http://www.omg.org, 1999

[RAT00] Rational Corportation, Unified Modeling Language (UML), URL: http://www.rational.com, 2000

[ROBB00] Jason Robbins, *Argo/UML*, Tigris, URL: http://argouml.tigris.org, 2000

[RUMB00] Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice-Hall, p.325, 1991

[SRD99] Henry Sowizral, Kevin Rushforth, Michael Deering, *The Java 3D API Specification*, Addison-Wesley, 1999

[SUN00] Sun Microsystems, DOM Parser, URL: http://www.java.sun.com, 2000

[UXF99] Junichi Suzuki, UML eXchange Format (UXF), URL: http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/ , 1999

[W3C00] W3 Consortium, eXtensible Markup Language (XML), URL: http://www.w3.org, 2000

[WARE00] Colin Ware, *Information Visualization, Perception for Design*, University of New Hampshire, Academic Press, ISBN 1-55860-511-8, 2000