ACTIVE LEARNING ANIMATIONS FOR THE THEORY OF COMPUTING

THE FSA SIMULATOR

by

Adib Roy

A project submitted in partial fulfillment

of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY

Bozeman, Montana

STATEMENT OF PERMISSION TO USE

In presenting this project in partial fulfillment of the requirements for a Master's Degree at Montana State University, I agree that the Computer Science Department shall make it available on the department's website.

If I have indicated my intention to copyright this project by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this paper in whole or in parts may be granted only by the copyright holder.

Adib Roy

June 2012

ACKNOWLEDGEMENT

All Praise be to Thee, Source of all Glory…

I would like to first and foremost thank my advisor, Dr. Rocky Ross for his constant supervision and patience with me, as well as for providing the necessary guidance for completing the project. Thank you for making me believe this was possible.

I would also like to express my most heartfelt gratitude towards my parents and sister for their constant support and encouragement throughout this project and during my academic career.

TABLE OF CONTENTS

ABSTRACT

The Finite State Automaton (FSA) Simulator is a tool that was developed by one of Dr. Ross's students in partial completion of his PhD thesis. The tool has shown itself to be quite useful in the teaching and active learning of finite state automata, but was showing signs of decay as technology evolved—it would not function properly on Mac computers, one of the features (the tape display) would intermittently display on some computers and operating systems, and many of the features were not as intuitive as they could be. This project was formulated around remedying these defects, and around creating documentation to facilitate future revisions and making possible and entire translation from its current language (Java) to a Web client side language (JavaScript) easier to manage. The objectives of the project were met and exceeded. The hope of the author is that this report will provide the necessary documentation to aid in any future work carried out on the system.

1. INTRODUCTION

This report describes a project completed under the direction of Dr. Rocky Ross in partial fulfillment of the requirements for the project track of the Master's Degree in the Computer Science Department at Montana State University.

1.1 The Finite State Automaton Active Learning Model

One of Dr. Ross's areas of research is the creation of Web-delivered active learning models of the theory of computing. A number of such models were created by one PhD student and various Master's students over the past number of years. One of these models, the Finite State Automaton (FSA) model, has shown itself to be quite useful in the teaching and learning of finite state automata. However, this model, referred to interchangeably in this report as the FSA Simulator, was showing signs of decay as technology evolved—it would not function properly on Mac computers, one of the features (the tape display) would intermittently display on some computers and operating systems, and many of the features were not as intuitive as they could be. This project was formulated around remedying these defects, and it included a number of other objectives.

1.2 Objective 1: Enhance the FSA Model

The objective of this aspect of the project, which consumed the most time, was to modify the FSA model software to remedy the stated defects, thereby ensuring that:

- The model could run on all platforms both as an application and as an applet in the latest versions of up-to-date Web browsers
- The input tape problem was resolved
- The features for constructing and manipulating FSAs were more intuitive

Additionally, the software was to be further documented to make future revisions, or even an entire translation from its current source language (Java) to a Web client side language (JavaScript), easier to manage.

This was not believed to be a trivial or straightforward task. The original software system was well-written, but very large. Other attempts by capable programmers had been made to take care of some of these problems with limited success. It was recognized that a focused, concerted effort would be required to dive into the details of the existing system, locate the areas of concern, and make the modifications. Indeed, the project was left open-ended on the assumption that some of the revisions might require more time to tackle than the project warranted.

1.3 Objective 2: Learn Event-Driven Programming in Java

Since the author had little direct experience in Java programming and none at all in event-driven programming, this objective was necessary to the success of the project. It also provided the author with a valuable new programming paradigm.

1.4 Objective 3: Become Acquainted with Active versus Passive Learning in the Context of Computer Based Educational Software

Understanding the basic importance of creating active, versus passive, learning software systems was a stated objective of the project. This included perusal of seminal papers that discussed this issue as well as papers published on the FSA model of this project and an analysis of the FSA module of the Duke JFLAP project, which has a similar goal.

1.5 Objective 4: Extend Knowledge of the Theory and Practice of Finite State Automata

The theory and practice of finite state automata have broad application in computer science. Notable among these is string pattern matching, the scanner of a compiler being a prime example. Since the focus of this project is a software system for understanding FSAs, this objective was a natural part of the project.

These objectives are covered in more detail in the Section 4 of the report.

The remainder of this report is as follows. Section 2 provides background information for the project, focusing on Grinder's work—how it enforces active learning, and some of the key theoretical foundations upon which his work is founded. Section 3 comprises an analysis of an existing tool—JFLAP—that fosters active learning of topics within the Theory of Computating. Section 4 is a detailed description of the methods employed to carry out the objectives of the project and the results of testing the system thereafter. Section 5 summarizes the project and provides a synopsis of the direction the project can take in the

future, with a focus on migrating the system to JavaScript and what the advantages doing so would entail. Section 6 is the References.

2. BACKGROUND

2.1 Grinder's Finite State Automaton Simulator

Grinder's work [7] presents an implementation and evaluation of active learning animation software for teaching a concept of the theory of computing, specifically the finite state Automaton (FSA). The software builds on techniques used in traditional textbooks, in which concepts are illustrated with static diagrams. Developers have tried to animate these static diagrams by adding motion, color and sound to them; however this exploits only a small amount of the potential that modern personal computer environments provide. Grinder's aim was to develop software for simulating finite state automata, the FSA Simulator, that would make further use of this potential by providing learning activities that would be impractical or even impossible to duplicate using traditional methods.

The FSA Simulator can be used as a stand-alone application or an applet embedded in a webpage and thus can be used for a variety of purposes from in-class demonstrations to integration into a comprehensive hypertextbook. Although there are a number of similar software applications, the FSA Simulator advances a step beyond conventional automaton simulations. Using algorithms that compute the closure properties of regular languages, the FSA Simulator can be used to create interactive exercises that provide instant feedback to students and guide them toward correct solutions.

Grinder performed experiments [8] in the computer science laboratory at Montana State University to evaluate the effect of the FSA Simulator on students' learning. While these

initial investigations cannot be considered either comprehensive or conclusive, they do indicate that use of the FSA Simulator significantly improves students' performance on exercises and may have some positive impact on students' ability to construct FSAs without the assistance of the simulator. Details of the experiments and results are described in section 4 of this report.

The development of the FSA Simulator represents significant progress in creating and evaluating active learning animation software to support the teaching and learning of the theory of computing. Grinder has demonstrated that such software can be created, that it can be effective, and that students find such software more motivating than traditional teaching and learning resources.

2.2 Active vs. Passive Learning

In recent years, the computer has revolutionized learning on an individual level, and collectively as a whole. Since the arrival of the Internet, computers have routinely been used by students for doing research and collaborating for academic purposes with others through e-mail, forums, and even social networking. So far, most applications of computer technology within education have largely been passive. For example, encyclopedias like Wikipedia include photographs, diagrams, movies, and sound clips, but few accompanying opportunities for students to interact with the subject they are studying in a way that promotes active learning. Such passive attempts tap into only a small portion for enhanced learning afforded by computers. It is well known that students learn better when they are engaged in active, rather than passive, learning [1, 15]. Thus, it is imperative that interactive

learning software be developed in order to exploit the full potential of computer-enhanced learning.

In contrast to the computer-based passive learning environments alluded to above, active learning software provides opportunities for a student to directly control the animation of a concept being learned. One way of doing this would be to allow the student to submit different inputs to a system that animates a concept, and then watch as the system processes the input to yield differing results. An example of this would be a finite state automaton animator that would show the processing of the input provided by the student, and produce the result, based on the student's input. The animator tool might even offer other opportunities for interaction, for instance allowing the student to control the speed and appearance of the animation, or allowing the student to convert the view of the concept being animated into a different view that would further aid in understanding the concept. The animator might even go so far as to offer tips of construction or provide constant feedback to the student to guide them towards successful completion of exercises.

Interactive animation software can present dynamic information in ways that are virtually impossible to do with traditional methods. Animation software also has the advantage of being "repeatable." Students are able to review lecture examples exactly as they were presented in the classroom.

Additionally, students using active learning animation software have the opportunity to explore a topic in greater detail and to further deepen their knowledge of a subject by trying examples that were not covered in class, or in the textbook. Carefully designed animation

software will not only demonstrate a concept, it will also capture students' attention and guide them toward a proper understanding by providing feedback as they progress through a session using the software.

Several studies on the effect that animation software has on learning have been carried out. Perhaps the most significant of these studies were carried out by Mayer [13, 14] and Stasko [23, Lawrence, 1994], who found that the difference in test results between students who used animation software to learn computer science concepts and those who did not were not necessarily significant, but that there was a significant difference in the results between those who used animation software that required the students' interaction with the software, and those that did not.

Grinder's FSA Simulator offers students several opportunities for active learning, by allowing students to interact with the system in different ways. Students can change the input string to the FSA and observe the way the animator changes its processing of the input to yield different results. Instructors can provide students with a starter FSA that will help by giving them a jump start to the construction of their FSAs. The instructor can even save a pre-built correct FSA in the background, which students can compare their answers against. This is a particularly useful feature of Grinder's FSA Simulator since it actively guides students through exercises toward a correction solution by prompting the student where a rectification needs to be made, and hence encourages students to keep working on a problem until they come up with the right result.

2.3 FSA Compare and Theoretical Foundations

The FSA Compare feature of the FSA Simulator is implemented using an interesting and unique (among the usual models of computation) property of FSAs: it is possible to detect whether two FSAs recognize the same language. This is possible due to the fact that regular languages are closed under union, concatenation and star (and, by implication, complementation) [22]. Furthermore, algorithms exist for creating an FSA from two given FSAs that recognizes the language resulting from the union or intersection of the languages recognized by the two given FSAs. Similar algorithms exist for the operations of complement and concatenation (see, for example [9]). There are also algorithms to determine whether an FSA recognizes only the empty set and, if not, whether the language recognized by the FSA is finite or infinite. Finally, given an FSA algorithms exist to identify strings that are accepted by that FSA. The FSA Simulator uses these algorithms to provide feedback to students doing exercises.

The student attempts to build an FSA M that recognizes a given language and when they think their solution is correct, they can check it against a target FSA M' that is preloaded in the background. The simulator first checks to see if the student's FSA M accepts any strings that the target FSA M' does not (this will be true if the FSA constructed to recognize the intersection of L(M) and the complement of L(M') is not empty. It then checks to see if there are any strings that the student's FSA M does not accept that are accepted by the target FSA M', which will be true if the FSA constructed to recognize the intersection of L(M') and the complement of L(M) is not empty. In either case, the student is prompted appropriately with

feedback regarding strings that their constructed FSA improperly does or does not accept. If neither case is true, then the student is prompted that their FSA is correct.

2.4 Defects in the FSA Simulator

The author would like to commend Grinder on the work he has done. In looking at the source code of the FSA Simulator tool, the amount of time and effort gone into producing it is evident. Grinder's FSA Simulator deserves all the praise and citations it has received. However, any software application's performance will suffer when it is left unmaintained for a decade. The objectives of this project center around remedying defects in the system resulting from advances in technology since Grinder's tool was put into production, ensuring that the tool can be run on the latest platforms and web browsers.

In 2002, when the FSA Simulator was first released, less than 1.8% of all computers were running Mac OS as their operating system [24]. Today that number has reached almost 10% and is growing, as Mac computers become more popular, especially amongst college students. In 2010, a New Jersey-based research firm called Student Monitor that has been tracking higher education computer purchases for 22 years, reported that, at 27%, Apple computers were the most used laptop by U.S. college students [6]. The report went on to state that among those who planned to purchase a new computer, 87% planned to buy a laptop. And among those students, 47% planned to buy a Mac. Of course statistics are only worth so much, as in the words of W. I. E. Gates, "Then there was the man who drowned crossing a stream with an average depth of six inches" [24]. Nonetheless, the growth in popularity of Mac computers can hardly be challenged. Hence it was deemed necessary not

only to enhance the model so that it ran on Mac computers, but in the process to modify the functionality of the system so that the actions for creating and manipulating FSAs were made more intuitive.

The simulator suffered from several signs of deterioration, some more apparent than others. A major one was the intermittent display of the input tape, details of which are described in section 4, Methods. Also, when implemented on Mac computers, the Alphabet window, and the window for selection of symbols for a transition would not indicate when a button was pressed. When loaded in Linux, the window size was too small, requiring the user to resize the window manually in order for the tape to display. A function that required modification due to improper design of the original model was the Clear functionality. In Grinder's system, there was only one Clear button, and upon clicking it, the system would completely clear the FSA diagram from the state diagram panel without prompting the user to confirm their action. In section 4, details of not only how this was fixed, but also how another feature along the same vein was added, are described.

3. RELATED WORK

Conducting a comprehensive search for FSA animator tools on the Internet actually led the author back to the FSA Simulator model from the Webworks laboratory at Montana State University on several occasions. There are several other attempts at trying to aid students with understanding the concept of a Finite State Automaton, but none of them appear to be of the active learning variety, except for one—JFLAP (Java Formal Languages and Automata Package) [10].

The following section is an analysis of JFLAP, an existing software tool that fosters active learning of topics within the Theory of Computing.

3.1 JFLAP

JFLAP is a software tool for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting a nondeterministic finite state automaton (NFA) to a deterministic finite state automaton (DFA) to a minimal state DFA to a regular expression or regular grammar [10].

JFLAP originated from FLAP (Formal Languages and Automata Package) after the release of the Java programming language. The latest version of JFLAP is 7.0, which was released on

August 28, 2009, and the last update was made on May 15, 2011. This analysis focuses on the finite automaton section of JFLAP, which is relevant to the FSA Simulator tool of this project, and not the other sections of JFLAP such as Mealy Machine, Moore Machine, Turing Machine, etc. The finite automaton section will henceforth be referred to as JFLAP for brevity's sake.

The first thing one would notice about JFLAP is that its user interface takes some getting used to, and requires that a student know what an FSA looks like, since the buttons on the toolbar simply contain icons for the creation of a state and transition. These buttons do have a tooltip text that pops up when you leave your mouse on the button for a while; however, a novice user might not be aware of this. Furthermore, this is time consuming. Having the text of the action on the buttons would be a more user-friendly design decision for students learning about FSAs.

The undo and redo buttons are unusual, in that clicking them does not do anything, except put the system into the respective mode. In order to actually undo or redo an action, the user has to click somewhere in the working area of the tool. This is an action sequence that is both unusual and non-intuitive.

A major drawback of this system over the FSA Simulator tool is the lack of real animation when testing an input string. The FSA Simulator tool clearly shows the transition from one state to the next by aid of a colored circle moving along the transition(s) from one state to the next. JFLAP does not do so. It simply highlights the state(s). This is particularly confusing in the case of a non-deterministic FSA where several states can be highlighted at a given time.

This is where the FSA Simulator really shines, showing animated transitions from each state of the FSA when a symbol in the string is read. The contrast is shown in Figures 3.1(a) and 3.1(b).

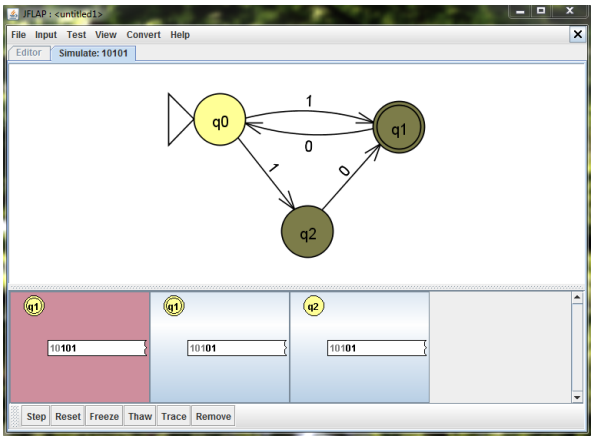


Figure 3.1(a): JFLAP in the middle of processing a string

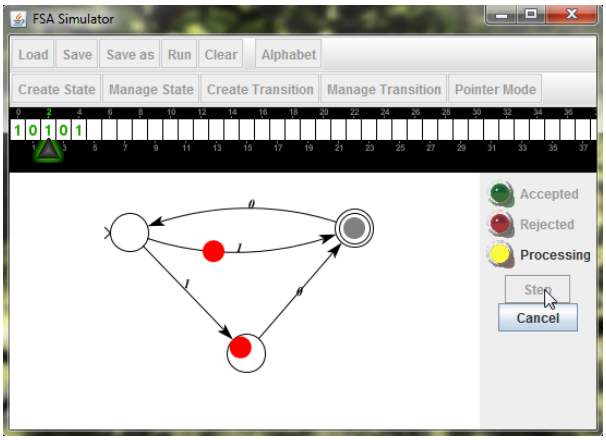


Figure 3.1(b): FSA Simulator tool in the middle of processing a string

In Figures 3.1(a) and 3.1(b) above, an example is shown where the same FSA is constructed in both JFLAP and the FSA Simulator tool. In both cases, the string 10101 is tested. In the image above, the screen is captured in the middle of the transition from the second 1 to the second 0 in the string, i.e. from the third to the fourth symbol in the string. As shown, JFLAP lacks any animation at all, and the states that are targets of the transitions are simply highlighted. In an FSA, particularly a non deterministic one with several states being reached on each symbol read, this can get very confusing for a student trying to understand what exactly is going on. The animation of the moving colored circle from one state to the next in the FSA Simulator tool greatly aids in the understanding process for a student.

Another drawback of JFLAP is its lack of clarity for whether a string was accepted or not, once the entire string is read. This is illustrated in Figures 3.2(a) and 3.2(b).

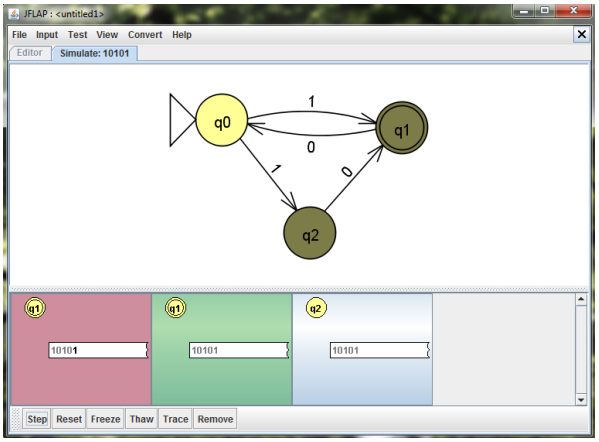


Figure 3.2(a): JFLAP at the end of processing a string

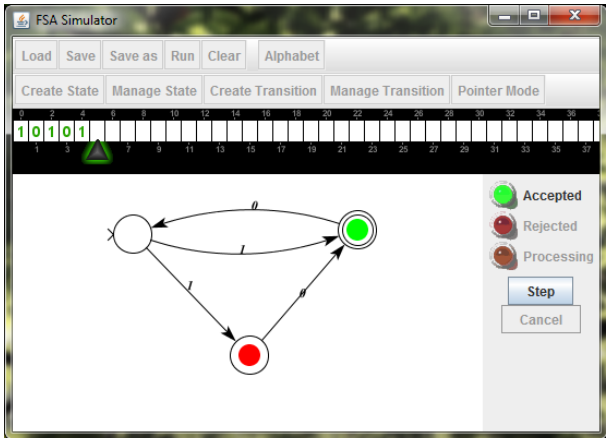


Figure 3.2(b): FSA Simulator tool at the end of processing a string

In Figures 3.2(a) and 3.2(b) above, the example used previously to read the string 10101 is continued to the end of the string in both JFLAP and the FSA Simulator tool. The FSA Simulator clearly shows the copies of the machine that are made from splitting non-deterministically when each symbol of the string is read. It also shows that at the end of the string, one of the copies ends on an accept state, and hence the FSA accepts the string, denoted by the green circle on the accept state. However, in JFLAP, this is not as intuitive as both states are marked in a shadow color, and it is unclear from looking at the colored boxes below what exactly is going on unless one already has an understanding of exactly how NFAs work. Hence this would not be the best learning for a student trying to grasp the concept of how NFAs function.

The FSA Simulator also provides a unique feature that greatly aids the understanding of the student, which is the compare feature. When run as an applet, the simulator allows the

student to compare the language of the FSA constructed in the state diagram panel to the language of the correct FSA that is preloaded in the background. Appendix B shows the different attributes that can be used when the simulator is loaded as an applet, of which targetfile is used to preload an FSA. When the student thinks that a correct FSA has been built, the Compare button can be clicked for verification. Whenever the Compare button is clicked, the simulator compares the language of the FSA that the student has constructed in the state diagram panel to the language of the correct FSA that was loaded in the background. If the languages are different, a dialog box will pop up to alert the student that either the displayed FSA accepts a string that is not in the target language (see Figure 3.3) or that it does not accept a string that is in the target language (see Figure 3.4). A specific "problem" string is simultaneously displayed to give the student an idea of where the constructed FSA falls short. Thus the student is guided toward a correct solution without being given the answer outright. When the student's FSA is correct, clicking the Compare button pops up a dialog box which informs the student that the constructed FSA is correct (see Figure 3.5).

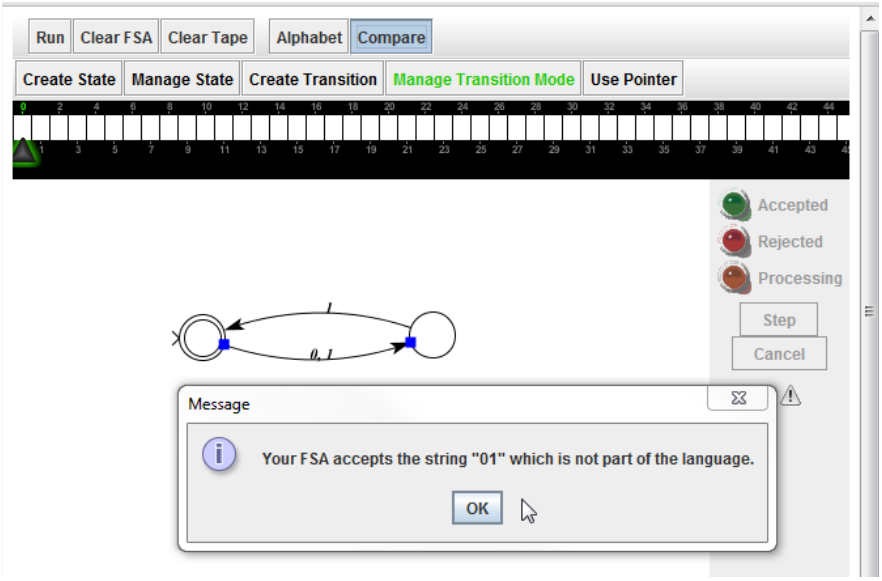


Figure 3.3: FSA comparison message when the student's FSA accepts a string not in the target language

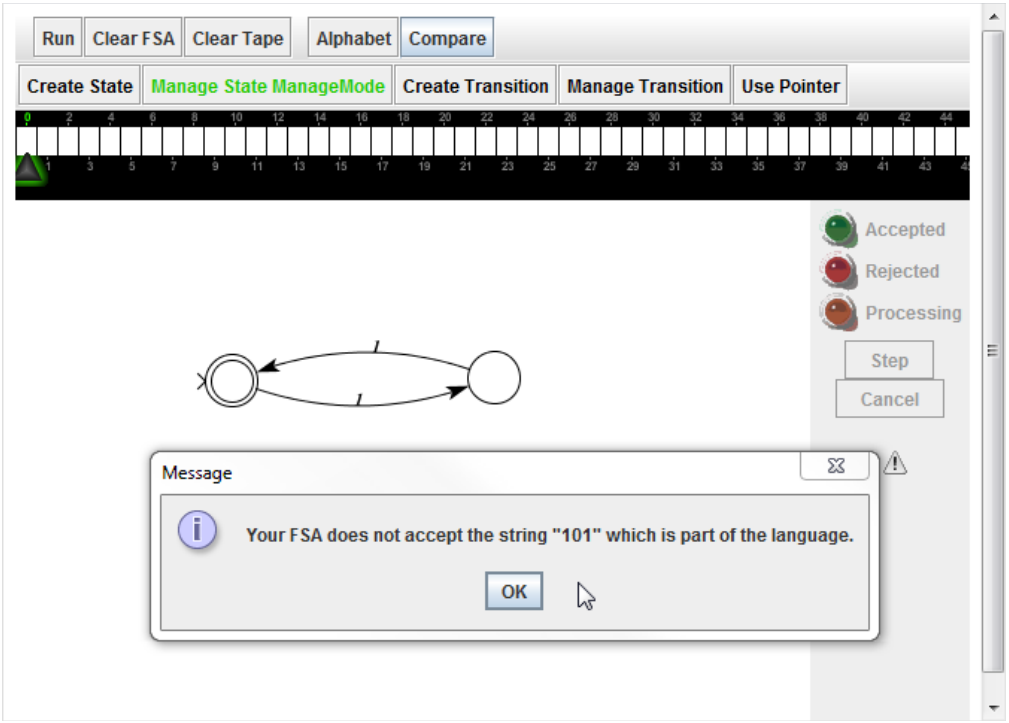


Figure 3.4: FSA comparison message when the student's FSA does not accept some string in the target language

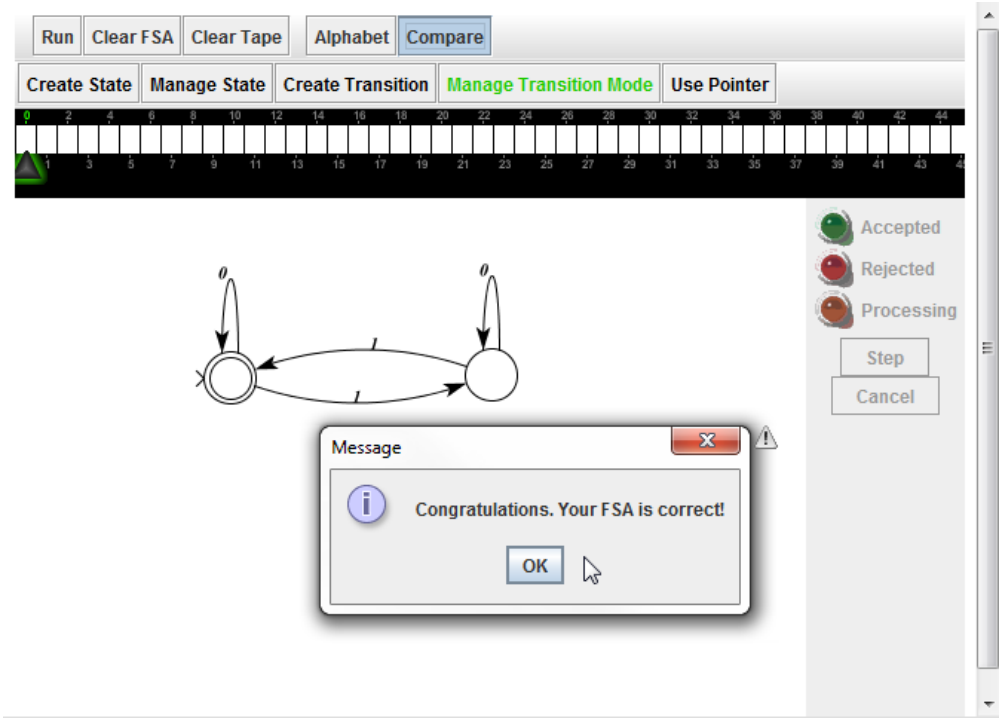


Figure 3.5: FSA comparison message when the student's FSA correctly recognizes the target language

Another feature of the FSA Simulator that sets it apart from JFLAP is a provision for the instructor to include a starter prebuilt FSA when the simulator is run as an applet. This is demonstrated in Appendix B, by using the fsafile attribute. The starter prebuilt FSA file contains a partial FSA that is provided by the instructor, with which the students can begin constructing their solutions. This FSA is displayed when the applet is started.

One great function that JFLAP does provide that the FSA Simulator does not is the conversion of an NFA to a DFA. This functionality is very useful for a student learning about how to convert NFAs to DFAs.

In our comparison, which is between the Finite Automaton tool of JFLAP and the FSA Simulator tool, even just comparing the menus that JFLAP provides will demonstrate its

overcomplexity. JFLAP's Finite Automaton tool has 6 top-level menus with 44 sub-menus, including options such as "Apply a Random Layout Algorithm", or "Highlight λ Transitions". The FSA Simulator tool is far more focused on aiding a student to understand how an FSA functions and only has 7 menu options in total.

In summary, JFLAP is a comprehensive tool for learning about concepts of the Theory of Computing, but in addition to lacking the animations to help a student grasp a concept easily, it can be overwhelming for a student who is just beginning to grasp these theories. Furthermore, the FSA Simulator includes unique featuers such as having a starter prebuilt FSA to aid the student in beginning the construction of the FSA, and more importantly, the compare function that aids the student in comparing the correctness of their FSA with a preloaded correct FSA. These features alone, more than prove the validity of the FSA Simulator model.

4. METHODS AND RESULTS

4.1 Objectives

The primary objective of Grinder's work [7] was to determine the feasibility of creating cross-platform, active-learning software as part of a comprehensive resource for teaching the theory of computing via the World Wide Web. He accomplished this objective, and the FSA Simulator model is a product of his accomplishment.

The main objectives of this project were to modify the model to ensure that it could run on all platforms both as an application and as an applet in the latest versions of up-to-date Web browsers. Bugs and defects in the system were remedied, such as the input tape problem. The features for constructing and manipulating FSAs were made more intuitive, and some features were added, such as clearing the tape. Additionally, the software was further documented to make future revisions, or even an entire translation from its current source language (Java) to a Web client side language (JavaScript), easier to manage. A detailed description of the objectives and methodologies to achieve them ensues, followed by a section on the results.

4.2 Methods

The primary objective of this project was to modify access to the core functionality of the FSA Simulator, i.e. constructing and manipulating FSAs, so as to make it accessible on all platforms both as an application and as an applet in the latest versions of up-to-date Web

browsers. This required several setup steps, before actual work could begin on making the necessary changes.

4.2.1 Setup and Prerequisites

Firstly, the source code was procured from Dr. Ross's former PhD student and the project was setup in an IDE. NetBeans was picked, because the author had some former experience in using it. Research into pre-existing documentation by Oracle [18, 19] proved particularly helpful in setting up the project in NetBeans. Appendix A outlines the setup of the project in NetBeans; it is a technical guide for anyone wishing to do future work on the project. A detailed description of what libraries are required to compile the code, where to find them and how to go about registering them is provided in Appendix A. After all the libraries were imported and registered, the code still would not compile due to a single error. The error and solution is highlighted in Appendix A.

While setting up the project, the author was involved in two other necessary prerequisites to the success of this project. The first involved learning about applet programming in Java. Researching this topic proved useful in understanding the concepts of action listeners [20] and mouse listeners [21]. The second prerequisite was a thorough read-through of Grinder's dissertation [7]. This provided an understanding of what Grinder's accomplishments were, and why he carried them out—a foundational requirement that proved necessary to carry this project forward.

The next logical step was to begin understanding the main components of the code. The first was the distinction between running the simulator as an application and as an applet. As shown below in the snippet of code taken from the FSAPanel class, if the simulator is being run as a stand-alone application, additional buttons for loading prebuilt FSAs from a file and for saving the currently displayed FSA to a file, labeled Load, Save, and Save As, are also displayed. This distinction is made in the following lines of code, found on line 328 of FSAPanel.java

```
if(!_isApplet)
{
   _toolbar.add(_loadAction).setText("Load");
   _toolbar.add(_saveAction).setText("Save");
   _toolbar.add(_saveAsAction).setText("Save as");
}
```

At this point in the project, the author's main aim was to familiarize himself with the inner workings of the code. Note that the source code for the simulator consists of 58 java files, with a total of 7,056 lines of source code. This number excludes blank lines, and comments. This also does not include the files in the Diva toolkit and other packages that the simulator depends on for its graphical representation. Put all together, the total number of files is 406, and the total lines of code, excluding comments and blank lines is 30,967. As one can imagine, deducing the inner workings of the simulator was no meager task.

4.2.2 A Shift in Paradigm

The main objective of the project was to modify the FSA Simulator to enable it to run on all major operating systems and web browsers. In attempting to achieve this goal, it was decided that the system would have to do away with the combination of mouse clicks used to perform

core functions of the FSA, such as creating a state or transition. The system relied on a combination of a left-click, right-click and ctrl-click actions to perform these functions. Instead, it was decided that each of these core functions would be performed when a related button was pressed, thereby setting the system in the relevant mode, and then simply clicking on the FSA Activity Panel. For example, whereas in the previous version of the system, a state was created by ctrl-clicking on the FSA Activity Panel, in the new version of the system, a state would be created by first clicking the Create State button, thereby setting the mode of the system to Create State, and then simply by left-clicking anywhere on the FSA Activity Panel to actually create a state.

This shift in paradigm would enable the system to run on any platform, by removing the dependency on a combination of mouse clicks that may not be possible to perform on some platforms. Also, the method of clicking a button to set the mode of the system and then using the mouse by clicking to perform the action is in keeping with several major applications such as Adobe Photoshop, Microsoft Office suite, and many others.

4.2.3 Adding the Buttons

The main goal was to understand the sequence of events that followed when a state was created, its properties changed, and when a transition was created and managed by adding symbols to it. In Grinder's version, these actions were created by a combination of ctrl-clicks and right-clicks, which were not only unintuitive, but also did not work on Mac computers. Even though it is possible to mimic a "right-click" on some Mac laptops by using two fingers at the same time, it does not fire the same event as a right-click on a PC, and hence the action

listener that is "listening" for the appropriate event (the button3_mask event in this case) would not get fired. The main aim was to extract the functionality from the ctrl-clicks and right-clicks and instead create four functional buttons: Create State, Modify State, Create Transition and Modify Transition, which when pressed, would perform the function specified when the user simply clicked on the appropriate object. This is better explained in table 4.1 below:

Function	Previously performed by	Proposed change
Create State	Ctrl-click anywhere on the state diagram panel	Press a "Create State" button. Then click anywhere on the state diagram panel
Modify State	Right-click a state	Press a "Modify State" button. Then click on a state to be modified
Create Transition	Ctrl-click a state, drag to the same state to create a transition to itself, or drag to another state to create a transition to that state	Press a "Create Transition" button. Click on a state, drag to the same state to create a transition to itself, or drag to another state to create a transition to that state
Modify Transition	Right-click a transition	Pressa "Modify Transition" button. Then click a transition to be modified

Table 4.1: Actions – Previous and Proposed

The author began tackling this task by first creating a new toolbar, called the edit bar, and then adding the appropriate buttons to it, as shown in Figure 4.1.

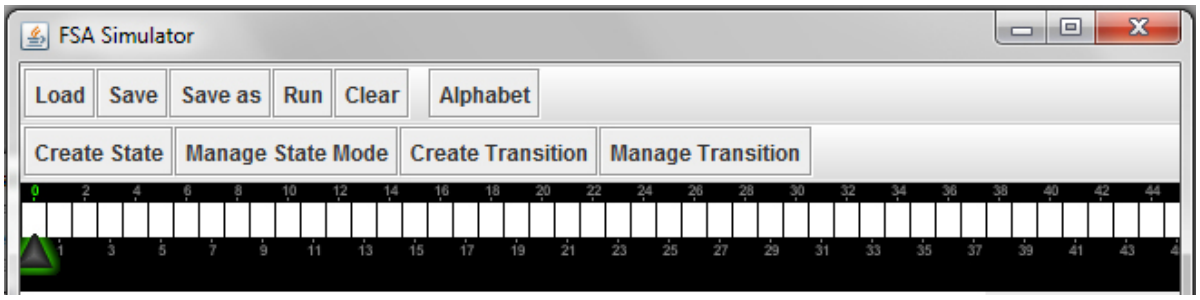


Figure 4.1: New buttons added to the Edit bar

An approach was adopted which is in keeping with standard practices of toolbars employed by larger applications such as Adobe Photoshop or Microsoft Office Suite, being that once a button is pressed, the simulator continues to perform the function of the button that is pressed, until another button is pressed, at which time, the simulator performs the function of the new button that is pressed. This is better illustrated in Figure 4.2.

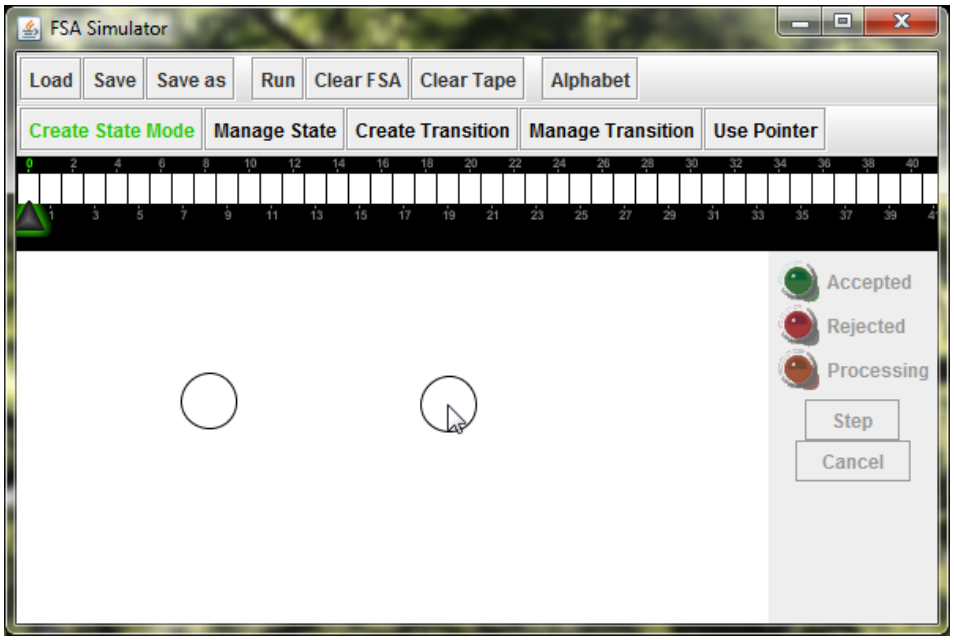


Figure 4.2: Illustration of buttons when pressed

As shown in Figure 4.2, a fifth button, Use Pointer, was added to the edit bar. The purpose of this button is to restore the functionality of the simulator to its default mode, which is to use the mouse simply as a pointer, and not in any of the other modes. Using the mouse as a pointer enables a user to select and drag states and transitions, simply by clicking the object and dragging it.

As illustrated in Figure 4.2, when a button is pressed, its text changes to include the word "Mode" appended at the end of the original text of the button, except in the case of the Use Pointer button, where the text changes to Pointer mode (and not Use Pointer Mode). Additionally, the color of the pressed button changes to a dark green to represent the mode that the user is currently in.

When a button is pressed the mode of the simulator is set, depending on which button is pressed. This is critical, since the mode of the simulator is checked before an event is carried out. For example, when the Manage Transition button is pressed, and a transition clicked, the system loads and displays the popup menu for the transition clicked. This will not happen if the Manage Transition button is not pressed, as the mode will not be set to that of managing a transition.

Once the buttons were added to the edit bar, and their functionality defined and coded, the next logical step was to start exporting the functionality from the ctrl-clicks and right-clicks to the buttons that were created. The author began tackling the issue of the functionality of buttons, one at a time. First in line, was the creation of states.

4.2.4 Create State

As stated in Table 4.1 above, in Grinder's version of the simulator, a state was created by ctrl-clicking anywhere on the state diagram panel. Note that ctrl-clicking an existing state does not create another state on top it, instead it selects the state by highlighting it, a function that makes sense and was maintained in the new version of the model. However, the ctrl-click function was changed to meet the new requirements of the project. In order to do this, first the author had to identify where the call to listen for a particular action was being made. This was found in the constructor of the StateCreator class.

```
setMouseFilter(new MouseFilter(InputEvent.BUTTON1_MASK,InputEvent.CTRL_MASK));
```

was modified to:

```
setMouseFilter(new MouseFilter(InputEvent.BUTTON1_MASK));
```

This changed the mouse listener to listen for BUTTON1_MASK, which is the left-click on the mouse, instead of BUTTON1_MASK and CTRL_MASK, which equates to the ctrl-click.

The next modification was made to the mousePressed function of the same class. A check was set to examine the global mode, which is set when one of the buttons on the edit bar is pressed. If the mode is CreateState, then the state is created.

```
public void mousePressed(LayerEvent e)
  {
     if (_fsapanel.mode == "CreateState")
     {
        Point2D position;
        State state;
        position = new Point2D.Double(e.getLayerX(), e.getLayerY());
        state = new State();
        _machine.addState(state);
        state.setPosition(position);
     }
  }
```

Once the author successfully managed to transfer the functionality of creating states from a ctrl-click on the state diagram panel to a click on the panel when the Create State button was pressed, it was assumed, and rightly so, that the remainder of the export of functionality would follow suit in the same manner. Unfortunately this was not to be, except for the case of exporting the function of managing a transition. The reason for this is explained shortly.

4.2.5 Manage Transition

In the case of managing a transition, previously this was done by right-clicking a transition. The aim was to export the functionality to a simple left-click on the transition, when the Manage Transition button was pressed. As in the case of creating a state, the author first identified where the call to listen for the right-click action was being made. This was found in the constructor of the TransitionPopupMenu class.

```
setMouseFilter(new MouseFilter(InputEvent.BUTTON3_MASK));
```

This was changed to:

```
setMouseFilter(new MouseFilter(InputEvent.BUTTON1_MASK));
```

This changed the mouse listener to listen for BUTTON1_MASK, which is the left-click on the mouse, instead of BUTTON3_MASK, which equates to the right-click.

The next modification was made to the mousePressed function of the same class. A check was set to examine the global mode, which is set when one of the buttons on the edit bar is pressed. If the mode is ManageTransition, then the system loads the popup menu for the transition.

```
if (_fsapanel.mode == "ManageTransition")
{
   _menu.show(e.getComponent(), e.getX(), e.getY());
}
```

This change successfully caused the popup menus for transitions to load when a transition was clicked. It appeared as though the same format could be followed to export the functionality of managing states and creating transitions as well, but as was hinted at earlier, there was yet to be a twist in the tale.

4.2.6 Manage State and Create Transition

The author then went back to trying to export the functionality of managing states. As indicated in table 4.1, previously, the popup menu for a state was loaded when a state was right-clicked. However, the objective was to export that functionality to a left click on a state, when the Manage State button is pressed. The author tried to take the same approach as in the two cases described above, i.e. the mouse listener for state popup menus was identified

and modified to meet the requirement. The mouse listener was found in the constructor of the StatePopupMenu class, and modified from

```
setMouseFilter(new MouseFilter(InputEvent.BUTTON3_MASK));
```

to

```
setMouseFilter(new MouseFilter(InputEvent.BUTTON1_MASK));
```

A check was put in place to examine the global mode so that the popup menu would only show when the mode was Manage State. However, this did not make the intended change, as it had in the two cases described above.

Considerable debugging of the code revealed that the setup of mouse filters was done in the initialization of the classes, long before the buttons were created or the user could set the global mode. The interactors defined and event listeners registered to these interactors hinted that at some point the event listener for the BUTTON1_MASK was already being set to the state. This was true, as clicking the state selected it, and enabled a user to drag it. But this was not set up in the constructor. Further debugging revealed that the mouse filter for the left click of states was set up in the AbstractInteractor and MouseFilter classes. In the MouseFilter class, an object of type MouseFilter was declared and initialized to listen for the left click, in the following manner:

```
public static MouseFilter defaultFilter = new MouseFilter(1);
```

Subsequently, in the AbstractInteractor class, this filter was set as the default filter, and hence setting the popup menu to appear on the BUTTON1_MASK was not functioning as intended. At this point, the author defined another mouse filter to listen for the right-click and assigned that to the state, so that the state could now be dragged by right-clicking. Note

that the user can drag a state when the system is not in the Manage State mode by simply clicking and dragging it. The modification was only made to the Manage State mode (and the Create Transition mode, as explained shortly) in order to be able to assign the left-click to another event.

Simultaneously, the author was monitoring the transition creation functionality of the system, since it is similar, in many ways, to the state management function. The major similarity is that both events fire from an action that happens on the state (later this was the cause of a further complication). In other words, in order to launch the popup menu for a state, a state would have to be clicked, and in order to generate the transition from a state, the same event (click) would have to be fired on the state. The author solved the problem of the overlapping left-click mouse listener for the transition creation function in the same way that the problem was solved for the state management function, by creating an additional listener on the state for the drag function.

However, on testing the changes, the system revealed that only one of the two functions—managing a state, or creating a transition—would function on a left-click action on a state, but not both. Even though a check was made for the global mode of the system, the system would perform only one of the two tasks. An in-depth investigation revealed that the task that would function depended on which object was initialized first in the constructor of the FSAGraphController class. This class creates a basic controller with nodes and edges, and as stated earlier, defines interactors that register nodes and events.

```
        ni = new NodeInteractor(this, sm);
…
        ni.addInteractor(new TransitionCreator());
        ni.addInteractor(new StatePopupMenu(machine));
```

In the code above, the TransitionCreator class is initialized first, and hence the event of left clicking a state would load the transition creator. However since the listener was already assigned to the left click for the transition creator, a state's popup menu could not be loaded on the same action, even though the mode of the system was being checked.

Further debugging led to finding the problem and eventually the solution. The StatePopupMenu and TransitionCreator classes were, like the other classes, calling the setMouseFilter function of the AbstractInteractor class. In the case of these two functions, the mouseFilter was being assigned to the same action, for the same object (left click for the state). Hence the solution was to call a different function from one of the two classes, so that the filter was not over-written. The author defined another function in the AbstractInteractor class to set the mouse filter for the state pop up menu, and the problem was solved!

The next step was to make the necessary checks for the mode that the system was running in. For the state management function, the mode is checked in the mousePressed function of the StatePopupMenu class, as shown below:

```
public void mousePressed(LayerEvent e)
  {
    if(_fsapanel.mode == "ManageState")
    {
      Figure source;
      Node node;

      source = e.getFigureSource();
      node = (Node)source.getUserObject();
      _target = (State)node.getSemanticObject();

      _startStateCheckBox.setState(_target.isStartState());
      _finalStateCheckBox.setState(_target.isFinalState());

      _menu.show(e.getComponent(), e.getX(), e.getY());
    }
  }
```

In the case of the transition creation function, the mode is checked in the mousePressed function of the TransitionCreator class. Only part of this function is shown below, for brevity's sake.

```
  public void mousePressed(LayerEvent event)
  {
    if(_fsapanel.mode == "CreateTransition")
    {
        //rest of the function
```

Finally, the four buttons performed as intended! The main objective of the project was accomplished. The next step was to fix the intermittent tape display.

4.2.7 Enhancement of the Tape display

The tape is the area where the user enters the string on which they wish to test the FSA. It is illustrated in Figure 4.3, as the area that is in focus, with the rest of the image blurred out.

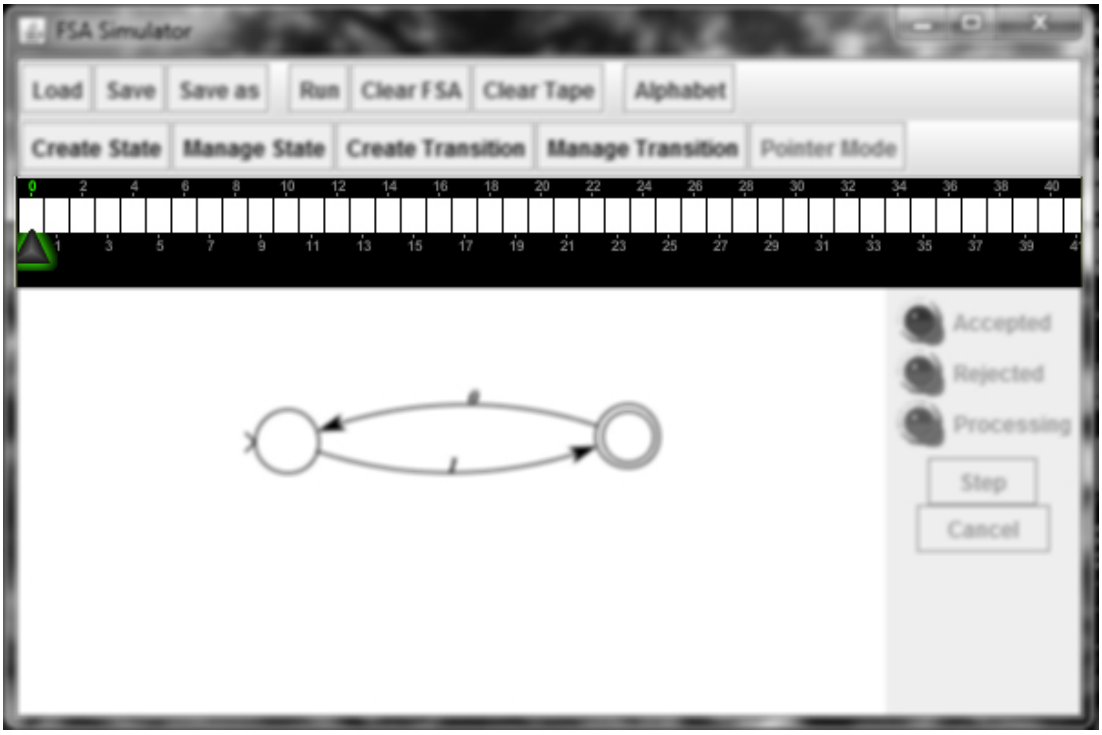


Figure 4.3: Highlighting the Tape

In order to enter a string, first the symbols that make up the language of the FSA are selected, using the Alphabet window, shown in Figure 4.4, and then the string is entered into the tape, as shown in Figure 4.5.

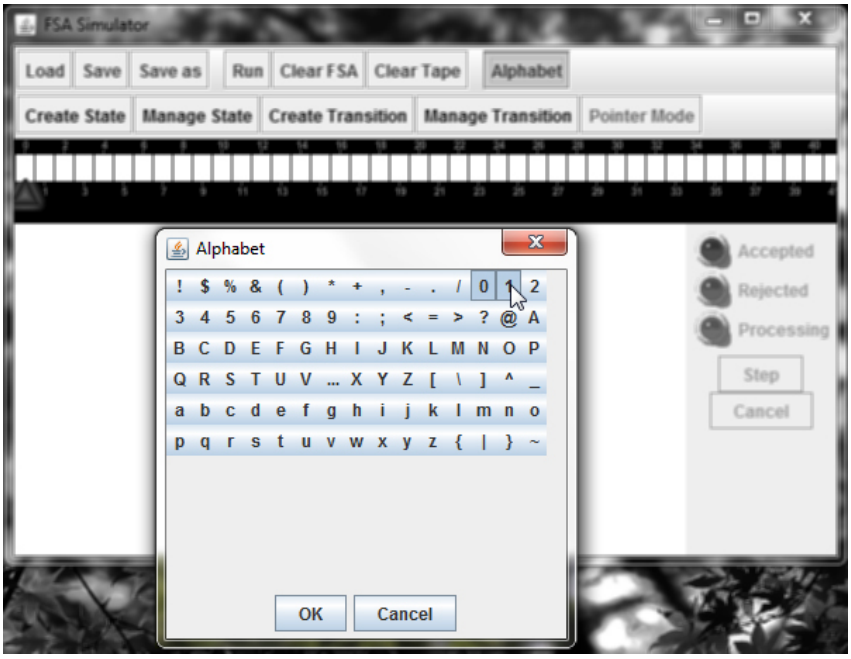


Figure 4.4: Alphabet window

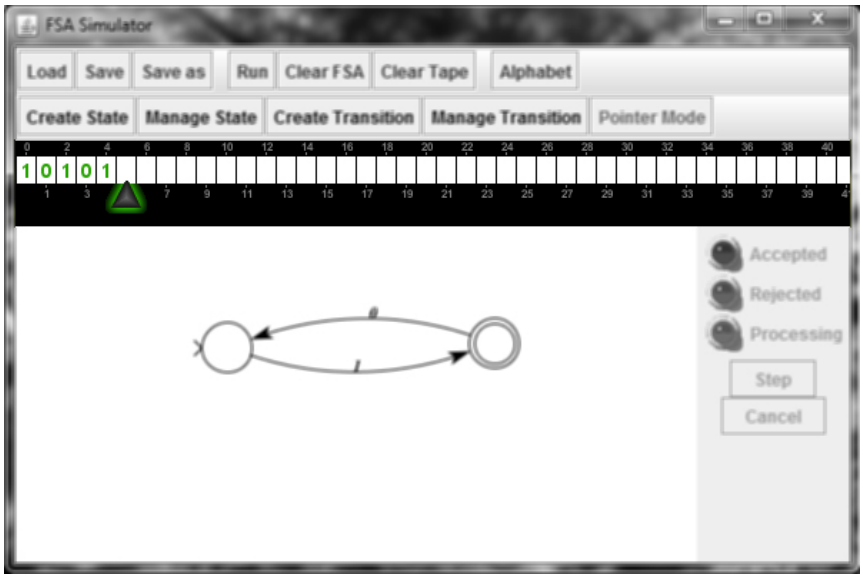


Figure 4.5: Tape with symbols

The issue with the tape was that when the simulator was loaded, the tape would not appear fully, as shown in Figure 4.6.

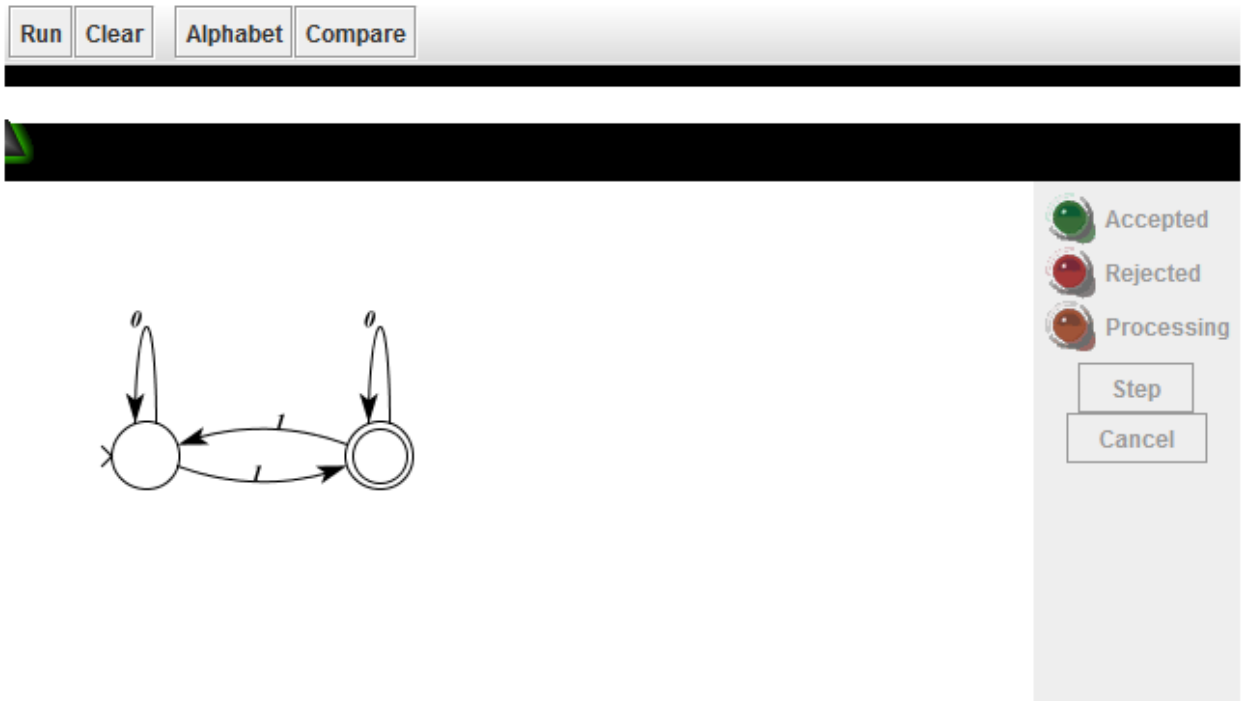


Figure 4.6: Illustrating the incompletely loaded tape

Dr. Ross mentioned that several attempts were made in the past to rectify this issue, but unsuccessfully. The tape would appear intermittently, but on a future release of the JRE, the fault would reappear. The fault has been fixed and tested on three different versions of the JRE (1.6.0_22, 1.6.0_31 and 1.7.0_03) and on a combination of operating systems and web browsers, details of which are included in the Results section. The description of how the issue was rectified is explained below.

The issue was thought to be resolved by introducing a call to the repaint() function in the paint() function of the Tape class, in the appropriate location in the function, after the lines for the tape are drawn and the colors assigned. However, when the system was tested on a Mac computer, the repaint call that was introduced, accompanied by another repaint call within the same function of the Tape class, was causing the tape to flicker out of control.

This was a difficult problem to troubleshoot, since it was a combination of two repaint calls within the paint function which caused this fault to happen. The problem was that removing these repaint calls would cause the tape to not appear properly in Windows, and yet having them, would cause the tape to flicker out of control on a Mac computer. There was only one thing to do.

To fix the problem, the author introduced a check for the operating system that the system is running on, and called the repaint function only if the operating system is Windows. This was done, as shown below.

This is how the check for the operating system is done.

```
private String nameOfOS = getOSDetails();

private String getOSDetails()
{
   String nameOS = "os.name";
   return System.getProperty(nameOS);
}
```

This is how the repaint function is called only when the operating system is Windows.

```
if(nameOfOS.contains("Windows"))
{
   repaint();
}
```

Note that the system performs on Linux just as it does on Mac OS, hence repaint() is called only on Windows.

4.2.8 Clear Tape

Another feature that was added was the Clear Tape functionality of the model. The previous version had only one clear button, which when clicked, would clear the contents of the FSA, without prompting the user for a confirmation. The confirmation dialog was added to the Clear FSA button. Additionally, a feature for clearing the tape was introduced. This was done by adding a second button called Clear Tape on the menu bar, shown in Figure 4.7, and assigning the function of clearing the contents of the tape to it. The previous Clear button was renamed to Clear FSA to make a distinction between the two clear functions. Both buttons ask the user to confirm the action before carrying it out.

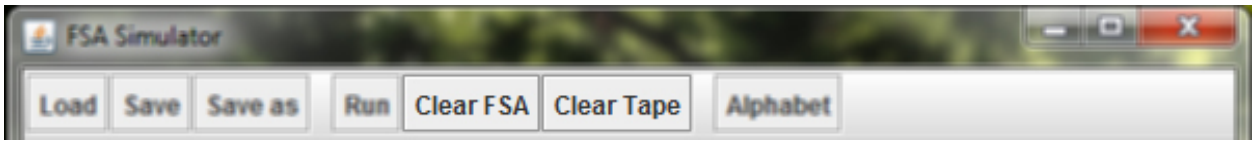


Figure 4.7: Clear FSA and Clear Tape buttons

The Clear Tape button, and indeed the Clear FSA button as well, are defined and created in the FSAPanel class, but the functionality for Clear Tape, shown below, was defined, rightfully, in the Tape class.

```
public void clearTape()
  {
    while(_head > _minLocation)
          {
                _tapeSymbols.removeElementAt(_head - _minLocation - 1);
                _head--;
                _maxLocation--;
                repaint();
          }
  }
```

It must be noted, that even though this feature was discussed early in the lifespan of the project, the actual inclusion of the "Clear Tape" functionality was done towards the end, as it was not a primary goal of the project.

4.2.9 Show Symbols Selected

A feature that was not discussed while defining the project objectives, but that came up when the system was tested on a Mac computer was that when a symbol was selected on the Alphabet window, or a transition popup menu window, on a Mac, it would not appear pressed. On a Windows PC however, the button would appear pressed. The contrast is shown in Figures 4.8(a) and 4.8(b).

Figure 4.8(a): Symbols selected on a Mac

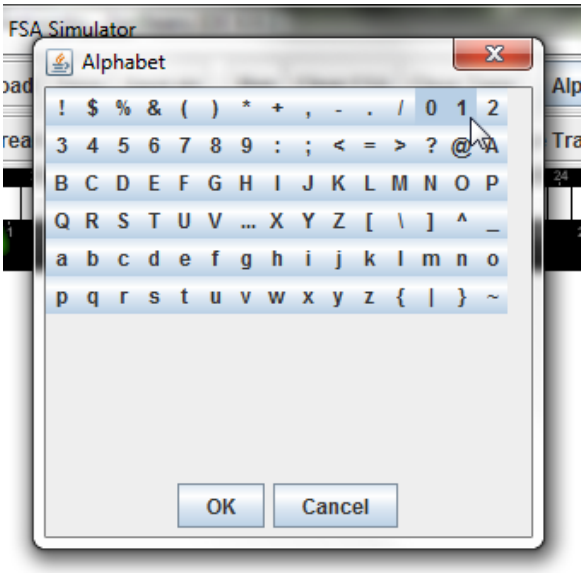


Figure 4.8(b): Symbols selected on Windows

In order to fix this problem, the author looked at several options, including changing the color of the button pressed, or increasing the size of the text of the button pressed, but it was

thought that the best way to tackle this issue would be to place a border around the button that is pressed. It turned out, that when tested on a Mac, this actually appeared as though the button was pressed, and the difference was not too apparent on a PC running Windows. This is illustrated in Figures 4.10(a) and 4.10(b).

Figure 4.10(a): Symbol selected on a Mac, after modification

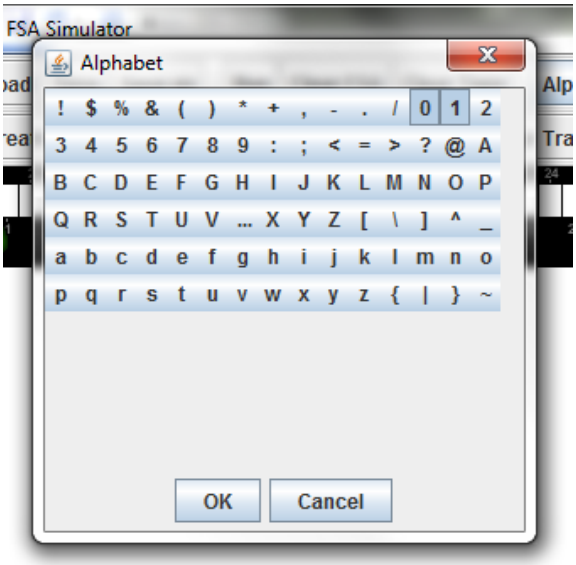


Figure 4.10(b): Symbol selected on Windows, after modification

The solution of placing a border around a button when it is pressed might appear trivial, but the author had to ensure that several factors were taken into consideration. Firstly, once a button was pressed in the Alphabet window, the border would have to appear until the button was pressed again. That could be at a later time, when the Alphabet window was loaded again. This was implemented in the following way:

```
public void toggled(ActionEvent e)
 {
  JToggleButton b = (JToggleButton)e.getSource();
  Character symbol = new Character(b.getText().charAt(0));

  if(b.isSelected())
  {
     if(selectedSymbols.contains(symbol))
     {
        removed.remove(symbol);
        b.setBorderPainted(true);
     }
     else
     {
        added.add(symbol);
        b.setBorderPainted(true);
     }
  }
  else
  {
     if(selectedSymbols.contains(symbol))
     {
        removed.add(symbol);
        b.setBorderPainted(false);
     }
     else
     {
        added.remove(symbol);
        b.setBorderPainted(false);
     }
  }
 }
```

In addition to this, the same functionality would have to be implemented when the transition pop up menu is loaded to assign a symbol to a transition. This was implemented in the refreshSelections() function of the SymbolPanel class.

```
while(iterator.hasNext())
  {
   mapEntry = (Entry)iterator.next();
   symbol = (Character)mapEntry.getKey();
   button = (JToggleButton)mapEntry.getValue();
   button.setSelected(selectedSymbols.contains(symbol));
   if(button.isSelected())
   {
      button.setBorderPainted(true);
   }
   else
   {
      button.setBorderPainted(false);
   }
  }
```

The trickiest bit was loading the correct pressed buttons for different transitions. In other words, if a symbol was selected for only one transition, and not another, it should not appear with the border when the symbol selection window for other transition is loaded. This was implemented in the symbolsAdded function of the SymbolPanel class.

```
while(iterator.hasNext())
  {
   symbol = (Character)iterator.next();

   b = new JToggleButton(symbol.toString());
   b.setBorderPainted(false);
   b.setMargin(margins);
   b.addActionListener(action);

   if(selectedSymbols.contains(symbol))
   {
        b.setSelected(true);
        b.setBorderPainted(true);
   }
```

Note that only part of this function is displayed here, for brevity's sake. Implementing these changes fixed the issue of the symbol button not appearing pressed on Mac computers.

4.2.10 Miscellaneous Modifications

A number of small, yet not insignificant, changes were made along the way. Some were cosmetic, and others yet functional. An example of a cosmetic modification made to the system, was to brighten the green color of the number above and below each cell of the tape, making it more visible.

A functional and perhaps more noteworthy modification to the system was made after testing the system on a Linux machine. It was noted that the size of the window (400 x 600) that is defined in the main function of the FSAPanel class caused the Pointer Mode button to be cut off by the edge of the window. More importantly, this caused the tape to not load, until the window was resized. Hence a modification was made to check for the operating system running, and to size the window accordingly. If the operating system is Linux, then the size of the window is set to 800 x 600, which is large enough to display all the elements of the system without having to resize the window. Yet another change made to the system was to increase the width of the buttons on the Alphabet window, since some buttons like the W character would appear as three dots because the character could not fit into the prescribed width of the button once a border was added to it.

In this manner, several miscellaneous matters affecting the esthetics and functionality of the program were resolved.

4.2.11 Comments

One noteworthy addition to the project was the inclusion of comments in the code. Every component of code that the author introduced into the system is well documented with comments alongside the code itself. An example of this is given below:

```
/**
   * This function is called from FSAPanel.java when the clear tape button is pressed
   */
  public void clearTape()
  {
     /**
      * This loop checks to see if there are characters left on the tape. If there are,
      * the character is removed and the position of the head moved. This loop is repeated
      * until there are no more characters to delete.
     **/
     while(_head > _minLocation)
            {
                   _tapeSymbols.removeElementAt(_head - _minLocation - 1);
                   _head--;
                   _maxLocation--;
                   repaint();
            }
  }
```

Before the author started work on the project, there were 7,056 lines of code and 702 lines of comments, at a ratio of more than 10 lines of code per comment line. The code added by the author was commented at a ratio of 1.4 lines of code to one comment line. These statistics are meant merely to give an estimation of the work done in terms of documenting the code that the author wrote. The statistics were gathered by using a third party tool – CLOC [3].

4.2.12 Documentation

It is the hope of the author that the comments in the code, and the documentation provided in this report, reinforced by the Appendices, will support future work done on this system, including the migration of the system from Java to JavaScript, a topic that is discussed in Section 5 of the report.

4.3 Modifications to the code

The following is a detailed description of the modifications that were made to the source code, followed by a UML diagram of just the classes that were modified:

4.3.1 FSAPanel.java

Lines 156-220 – Implements the Clear Tape action, and enhances the Clear FSA action to prompt user for confirmation dialogue.

Lines 369-491 – Creating and placing the edit bar and added the buttons to the bar.

Lines 494-537 – Increase contrast of the animation panel on the right side of the window.

Lines 539-547 – Eliminates error shown on Java console when a starter file is provided by no target file.

Lines 639-674 – Increase the size of the window when the system is loaded on the Linux OS, because by default, the size of the window loaded in Linux is not large enough, and this causes the tape to not appear at all.

Lines 702-790 – Implements action listener functions for each of the added buttons.

Lines 791-856 – Changes and restores colors and text of buttons when one is pressed.

Lines 858-870 – When the Cancel button is pressed, enable the edit bar.

Lines 873-886 – When the Run button is pressed, disable the edit bar.

4.3.2 StateCreator.java

Line 23 – Changes action listener from control-click to left click.

Line 30 – Checks the global mode before executing state creation action.

4.3.3 StatePopupMenu.java

Line 32 – Changes action listener from right-click to left-click.

Line 62 - Checks the global mode before executing state popup action.

4.3.4 SymbolPanel.java

Line 103 – Increase size of symbol width since characters like ‘W’ do not appear fully.

Lines 141-153 – function called when a symbol is selected from the transition popup menu.

Lines 198-227 – Setting and removing borders when symbols are selected and deselected.

4.3.5 Tape.java

Lines 155-160 – Fixes the intermittent tape display by checking if the OS is Windows and repainting if it is.

Lines 214-218 – Checks the OS for Windows and repaints if it is.

Lines 241-254 – Gets the OS that the system is running on.

Lines 746-765 – Function called from the FSAPanel.java when the Clear Tape button is pressed. It clears the tape.

4.3.6 TransitionCreator.java

Line 38 – Changes action listener from ctrl-click to left click.

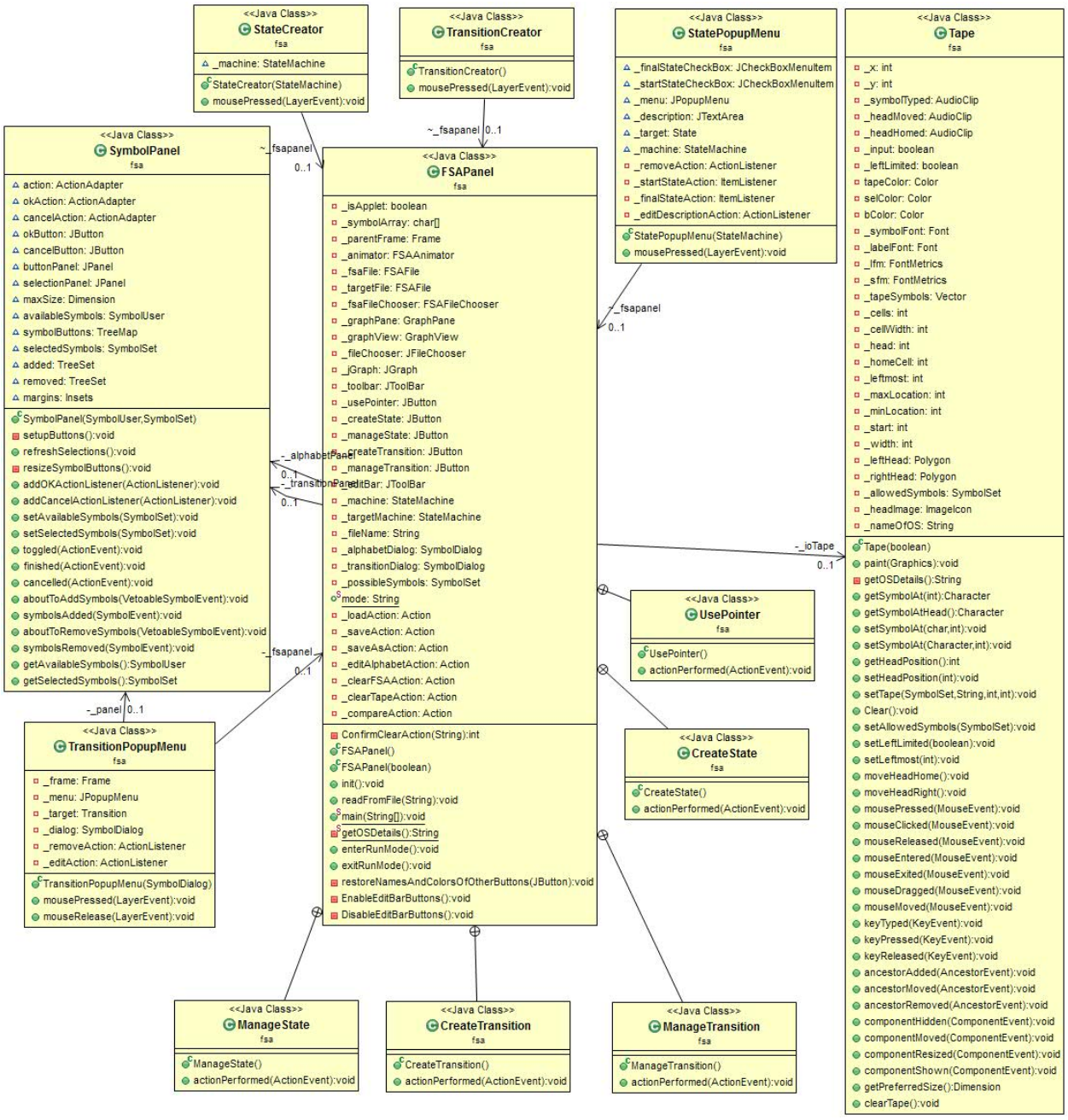
Line 44 – Checks the global mode before executing transition creation action.

4.3.7 TransitionPopupMenu.java

Line 40 – Changes action listener from right click to left click.

Line 62 – Checks the global mode before executing transition popup menu action.

4.3.8 UML Class Diagram of the classes which were modified



4.4 Results

4.4.1 Background

The primary objective of Grinder's work [7], as mentioned in the Objectives section above, was to determine the feasibility of creating the FSA Simulator tool that is a cross-platform, active learning software as part of a comprehensive resource for teaching the theory of computing on the World Wide Web. A secondary objective was to begin an evaluation of the effects of this software on student learning, and he has done so by providing two preliminary evaluations.

The two preliminary evaluations were conducted in computer science labs during the spring semester of 2002. Both experiments demonstrated that use of the FSA Simulator can significantly improve students' performance on FSA construction exercises and the results of the second evaluation suggest that use of the simulator may increase students' success at constructing FSAs without the assistance of the simulator applet.

Experiment 1 was performed in a first-year computer science course at Montana State University. Demographic information such as each student's age, sex, major, year in college, approximate grade point average, etc. was collected, so that the two groups formed – test and control, could be compared for significant differences. Fifty-two students participated in the first experiment. Twenty-eight students made up the test group, while twenty-four students made up the control group. Overall, the demographics of the two groups appeared to be equivalent.

The lab exercise was divided into two parts. In the first part, the students read through a web-based tutorial based on material from the Webworks Laboratory's theory of computing hypertextbook [2]. Examples in the test group's tutorial included the FSA Simulator applet. In place of the applet, the control group viewed static images that conveyed equivalent information. While they were reading through the tutorial, the students completed four exercises that tested their knowledge on FSAs, such as identifying the start and final state of an FSA, listing the sequence of states the FSA would enter while processing a particular string, determining whether the FSA would accept or reject 4 different strings, and constructing a state diagram of an FSA from its formal description, etc.

The test group was allowed to use the FSA Simulator (with the comparison feature enabled when applicable) during all of the exercises. Images of solutions to the exercises were made available to the control group to use in checking their solutions as links from the exercise pages. Both groups were asked to write down their answers to the exercises on a separate sheet of paper.

The second part of the lab was a paper-and-pencil test with problems similar to the exercises in the tutorial. All students took the same test without reference to any of the online materials from the tutorial. The lab had to be completed within 110 minutes.

It was observed that the students in the test group took longer to complete the exercises due to several factors, namely, they had to learn how to use the FSA simulator, there were some bugs in the applet that forced some students to restart in the middle of an exercise, and because they received feedback about their solutions, which encouraged them to keep

working on a solution until it was correct. As a result, nearly a third of them (8 of 28) were not able to complete the exercises. The control group, on the other hand, completed the exercises (although not always correctly) and the test much more quickly.

It was observed that many of the students in the test group appeared to enjoy working with the FSA Simulator applet. The ability to check the accuracy of solutions and receive hints when incorrect solutions were submitted made the exercises seem like a puzzle game. A few of the students even competed with each other to see who could complete the exercises first. On the other hand, the control group was much more subdued. The students in this group asked fewer questions and appeared to be much less enthusiastic about the lab.

The data collected from the results of the experiments clearly demonstrates that the comparison feature of the FSA Simulator does improve student performance on exercises. Although the control group appeared to have a slight edge when identifying the parts of a finite state automaton, on exercises that required FSAs to be constructed from a language description, the test group had a definite advantage. The percentage of students in the control group who successfully completed exercises dropped steeply as the exercises increased in difficulty, which was in contrast to the students in the test group.

Experiment 2 – The experiment lab was performed in a second-year computer science course at Montana State University. As in the first experiment, demographic information of the students who participated in the experiment was collected, so that the two groups formed – test and control, could be compared for significant differences. The control group comprised 17 students, and the test group 27 students.

The lab for Experiment 2 was nearly identical to Experiment 1. The online tutorial was modified slightly to correct some typographical errors and to reword the questions about the start and final states in Exercise 1. Also, the bugs discovered during Experiment 1 were fixed before Experiment 2. Since the labs for this class were not as structured as the first lab, the two-hour time limit was not as strictly observed. As in Experiment 1, the test group tended to do better than the control group on the exercises. The test group did better than the control group on some of the problems, but not significantly better.

The data collected from the results of the two groups showed that the test group's advantage over the control group grew as the hardness of the exercises increased. The control group's performance dropped at a steady rate as the difficulty of the exercises increased. Although not quite as dramatic as in Experiment 1, the test group clearly had an advantage over the control group for the exercises.

As in the exercises, the students in Experiment 2 scored higher on most of the problems on the test. However, except for on the hardest problem, the difference between the scores was insignificant. On one of the questions, the test group did much better than the control group, although the difference was not quite large enough to be considered statistically significant.

Grinder concludes that the results suggest that there may be some minimum amount of knowledge needed for the FSA Simulator applet to significantly affect learning. Much more evaluation needs to be done to confirm this hypothesis.

4.4.2 Test Cases

Grinder's evaluation on the effect of the FSA Simulator on students' learning of the concept is comprehensive and sufficient for the purpose of this project. Moreover, it was deemed unnecessary to repeat tests similar to the ones conducted by him, since repeating the tests would not measure the achievements of this project. Instead, an evaluation was carried out to test the main objective of this project, which was to modify the model so that it could run on all platforms both as an application and as an applet in the latest versions of up-to-date Web browsers. Tests were also carried out to verify that the changes made to rectify faults in the system, and to enhance the system, worked, and that they did not introduce new errors.

Tests were carried out in the laboratory of the Computer Science Department at Montana State University. The lab has both PC and Mac computers, and hosts Windows, Linux and MacOS on its machines. The FSA Simulator was tested as both an application and an applet on all installed web browsers on each of the three operating systems. The following test cases were carried out. On each test, the following actions were tested:

Test case 1: Select the symbols that comprise the language of the FSA

- Load the Alphabet window. Select the symbols that comprise the language of the FSA. Close the Alphabet window. Reload the Alphabet window to verify that symbols that were previously selected appear selected.

Test case 2: Basic FSA actions

- Create state
    - Click the Create State button. Click anywhere on the panel. Verify that state is created.
    - Click on an existing state. Verify that an additional state is not created on top of an existing state.
    - Click the Use Pointer button. Click anywhere on the panel. Verify that a state is not created.
- Assign start and final states
    - Click the Manage State button. Click a state. Verify that the state popup menu loads.
    - Click the Use Pointer button. Click a state. Verify that the state popup menu does not load.
- Create transitions from one state to another, and from a state to itself
    - Click the Create Transition button. Click a state and drag the mouse from it to another state. Verify that a transition is created.
    - Click a state and drag the mouse from it to a state that it already has a transition to. Verify that a duplicate transition is not created.
    - Click the Create Transition button. Click a state and drag the mouse from it to itself. Verify that a transition is created from the state to itself.
    - Repeat the step above for the same state. Verify that a duplicate transition is not recreated.
    - Click the Use Pointer button. Click a state and drag the mouse from it to another. Verify that a transition is not created.

- Assign symbols to transitions
    - Click the Manage State button. Click a transition. Click the Edit Symbols options on the transition popup menu. Assign symbol(s) to the transition. Verify that the symbol(s) assigned appear as labels on the transition.
    - Click the Use Pointer button. Click a transition. Verify that the transition popup menu does not load.
- Drag components of the FSA around on the working space
    - Click the Use Pointer button. Click a state and drag it around. Verify that you cannot drag it outside the borders of the working space (FSA Activity Panel).

Test case 3: Enter symbols onto the tape

- Enter symbols onto the tape. Verify that only symbols that are selected in the Alphabet window can be entered onto the tape.
- Click the Run button. Verify that symbols cannot be entered onto the tape when the FSA is in run mode.

Test case 4: Modify the contents of the tape

- Press the left and right keys on the keyboard to navigate on the tape. Verify that entering a valid symbol on an existing symbol replaces it. Verify that pressing the backspace button deletes the previous symbol on the tape.

Test case 5: Run the FSA on a string that it accepts

- Create an FSA. Enter a string on the tape that the FSA should accept. Click the Run button. Verify that the FSA accepts the string.

Test case 6: Run the FSA on a string that it rejects

- Create an FSA. Enter a string on the tape that the FSA should reject. Click the Run button. Verify that the FSA rejects the string.

Test case 7: Add more states and transitions to make the FSA non-deterministic

- Create an NFA by adding states and transitions, whereby the FSA can have more than one transition for a single symbol of the alphabet. Also add empty string transitions.

Test case 8: Test the NFA on strings its accepts and rejects

- Enter a string on the tape that the NFA created in the step above should accept. Click the Run button. Verify that the NFA accepts the string.
- Enter a string on the tape that the NFA created in the step above should reject. Click the Run button. Verify that the NFA rejects the string.

Test case 9: Save a new FSA to the local disk

- Click the Save as button. Verify that the Save Window loads, and that the FSA is saved on the local disk in the location specified.

Test case 10: Load a saved FSA from the disk

- Click the Load button. Verify that a saved FSA is loaded.

Test case 11: Update a saved FSA from the disk

- Load an existing FSA from the disk. Make the necessary changes to the FSA. Click the Save button. Verify that the changes have been saved.

Test case 12: Compare the created FSA to an existing one

- Load the system as an applet. Verify that the target FSA is specified in the TargetFile parameter of the Object tag. Click the compare button to verify that the system compares the language accepted by the target FSA to that of the FSA created by the user.

Test case 13 Clear the FSA

- Click the Clear FSA button. Verify that the system prompts the user to confirm the action. Verify that pressing the Cancel or No buttons does not clear the FSA. Verify that pressing the Yes button clears the FSA.

Test case 14: Clear the Tape

- Click the Clear Tape button. Verify that the system prompts the user to confirm the action. Verify that pressing the Cancel or No buttons does not clear the tape. Verify that pressing the Yes button clears the contents of the tape.

4.4.3 Current Results

On the Windows operating system, the simulator performed to standard, as an application, by simply double-clicking the jar file to execute the application, whereas previously, double-clicking the jar file would launch an error that stated that the manifest file did not contain the main class. The simulator was then tested as an applet on each of the installed web browsers on the machine, namely Google Chrome, Mozilla Firefox, Opera, Internet Explorer, and Sea Monkey. It must be noted that the tool did not work on an older version of Internet Explorer, but did so flawlessly on the latest version (Internet Explorer 9). Safari is not installed on the Windows operating system of any of the computers in the lab, but a test of the applet on Safari running on Windows on another computer yielded perfect results.

On the Mac OS, the simulator performs just as well as it does on the Windows OS. The FSA Simulator was first tested as a stand-alone application. Double clicking the jar file launched the application and all the tests were carried out yielding perfect results. Note that previously the application could not be launched by double-clicking the jar file. As an applet, as well, the simulator performed to standard. Tests were carried out on all the web browsers that are installed on the Mac OS, namely Safari, Google Chrome, Mozilla Firefox, Opera and Sea Monkey. It should be noted that Internet Explorer was not installed on the Mac computer in the lab, and hence has not yet been tested on that combination. It should also be noted that previously, the FSA Simulator could not be used as an applet on Macs, since key functions depended on the right click of a mouse, such as assigning start and finish states, and assigning symbols to transitions. This has now been resolved by moving all functionality to simple clicks, hence making the applet usable on a Mac.

Testing on Linux-Fedora yielded similar results. The application was able to be loaded by double-clicking the jar file and performed to standard. The applet also performed to standard on all the major browsers installed on the Linux operating system, including Google Chrome, Mozilla Firefox, and Sea Monkey.

5. SUMMARY AND FUTURE WORK

5.1 Summary

The main objective of this project was to modify the FSA Simulator to make the functions of creating and manipulating FSAs more intuitive, in turn enabling the simulator to run on all major platforms and web browsers. This objective was achieved. During the process, some new features were added and others fine-tuned. The documentation produced is aimed at making future revisions or even an entire translation from the current source language (Java) to a Web client side language (JavaScript), easier to manage.

The FSA Simulator was modified to enable it to run on all major operating systems. In the process of doing so, the actions for handling the basic functions of the FSA were made more intuitive. In the previous version of the system, a combination of clicks, ctrl-clicks and right-clicks caused the system to perform the basic creation and management of states and transitions. This was modified in the current version of the system. A toolbar with buttons for each of the actions was added and the functionality of the system modified so that when a button is pressed, simply clicking on the FSA activity panel performed that action.

In addition to modifying the actions for the core functionality of an FSA, enhancements were made to the system and errors that existed previously as well that arose as part of the modification of the system, were rectified. One such error that existed in the previous version of the system was that the tape displayed intermittently in the Windows operating system. This was rectified.

An enhancement that was made was that in the previous system, when the Clear FSA button was pressed, the system would clear the FSA without prompting the user for a confirmation. This confirmation was added to the system. Also, a Clear Tape button was added which clears the contents of the tape after the user confirms the action when prompted to do so.

When the system was tested on the Mac OS, it was noticed that when the user selects a symbol in the Alphabet window, the symbol does not appear to be selected. This was rectified by adding a border to selected symbols, both in the window that is used to set the accepted symbols of the alphabet of the FSA, as well the window that is used to select the symbols of a transition.

Another change made to the system was to increase the width of the buttons on the Alphabet window, since some buttons like the W character would appear as three dots because the character could not fit into the prescribed width of the button once a border was added to it.

It was noted that the size of the window in Linux caused the Use Pointer button to be cut off by the edge of the window, as well as, the tape to not load, until the window was resized. Hence a modification was made to check for the operating system running, and to size the window accordingly. If the operating system is Linux, then the size of the window is set to be large enough to display all the elements of the system without having to resize the window.

An error was noticed when the Java console window is turned on, and a starter file provided, but no target file specified. This error was rectified as well.

5.2 Future Work

One of the features that may be looked at in a future revision is the Save/Save as function. There are several issues with it. Firstly, it needs to be made clear that the Save function is meant only to overwrite an FSA that is already saved. In order to save an FSA that the author has just created, the Save as button needs to be clicked. This either needs to be changed, or made obvious. Secondly, when a user opts to save a newly created FSA, they have the option to enter the name of the FSA, but this works intermittently. At times, the textbox for the name of the file does not allow the user to enter a name. Additionally, even though the combo box for specifying the type of file is defaulted at "FSA File (.fsa)", if the user does not enter the .fsa extension after the file name, the system does not save the file as an fsa file. Yet another issue with the Save function is that, while saving the FSA, if there are transitions that are not labeled with a symbol, they are not saved. While these are issues that require fixing, it must be noted that the Load and Save/Save as features are not available when the simulator is run as an applet, which is the mode that all hypertextbooks will run the simulator in, and the means by which most students access the system.

A feature that would be nice to have and that may be implemented in the next release of the system is an Undo/Redo function. Even if it is limited to undoing or redoing a single action, it might still save the user a considerable amount of time and heartache should the user make a mistake and wish to undo it. From the author's understanding of the inner workings of the system, this would not be a meager task to implement.

Yet another feature that could be implemented in the next release of the system is the ability to convert an NFA to a DFA. In the author's estimation, this would almost comprise of a project in itself.

Finally, Grinder's evaluation [8] provides a preliminary analysis of the effect of the system on student's learning. The author believes that the tests carried out can be built upon and repeated under different circumstances to improve upon and possibly increase the conclusions from the evaluations. Having said that, the evaluations carried out by Grinder are in keeping with studies of this nature, and are similar to several such studies carried out [13, 23, 12].

Ultimately, a complete rewrite of the system in a web-based client-side language such as JavaScript would eliminate most of the issues that warranted this project's existence. Many of the issues caused by the system are because of its reliance on the Java Virtual Machine (JVM). A user wishing to user the FSA Animator has to ensure that the JRE is installed on his/her personal computer. For a novice user nervous about installing an unknown application on their computer, or for a user who does not have administrative privileges on the computer they are wishing to run the tool on, this might be a deal-breaker and render the system useless. Even if the JRE is installed on the computer, it may not be the latest version released by Oracle, and may therefore react differently to the tool than what the user expects. Furthermore, Apple deprecates the version of the JVM that it distributes with the Mac OS. Hence, the user is dependent on the version of the JVM that Apple provides on their computers. Ultimately, all of these issues would be eliminated if the system ran on a web-based client-side language such as JavaScript.

The advantage of rewriting the system in JavaScript is that, like Java, it is platform independent, that is, JavaScript will compile and execute on any hardware/operating system pair that the JavaScript virtual machine is installed on and compiled for. The difference between Java and JavaScript is that the compiler for JavaScript is included with every major browser and does not need to be preinstalled by the user. The compiler either compiles the JavaScript source code into an Intermediate form (IR, like byte codes for Java) and then interprets the IR on the machine, or, in the case of the V8 VM from Google for the Chrome web browser, directly translates JavaScript source code into machine code that is then executed on the native machine [11]. In either case the JavaScript Virtual Machine is included with all major web browsers and this holds the edge over the JVM.

The challenge of implementing the system in JavaScript is to be able to find a graphics package that will meet the needs of the system. WebGL [25] is a cross-platform web standard for a low-level 3D graphics API, exposed through the HTML5 Canvas element, that might meet, and indeed exceed the needs of this system. Such a package might be just what is required to replace the Diva toolkit [4], which is a graphical framework developed at the University of California-Berkeley, chosen for building the graphical user interface of the FSA Simulator. The benefit of using the Diva toolkit is that in addition to taking advantage of Java's Java2D API, it also provides important graph display classes necessary for constructing the state diagrams used to display FSAs. The developer in charge of rewriting the system in JavaScript would have to investigate WebGL, or whatever graphics package is chosen at the time, to ensure that it meets all the needs of this system.

6. REFERENCES

[1] Bonwell, C. C., and Eison, J. A. Active Learning: Creating Excitement in the Classroom. ASHE-ERIC Higher Education Report No. 1. Washington D.C.: The George Washington University, School of Education and Human Development, 1991.

[2] Boroni, C. M., Goosey, F. W., Grinder, M. T., and Ross, R. J. Engaging students with active learning resources: Hypertextbooks for the web. In The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education [March 2001], vol. 33, pp. 65–70.

[3] CLOC. [2012] Count Lines of Code <http://cloc.sourceforge.net/>

[4] University of California-Berkeley "Welcome to Diva" [2001] <http://embedded.eecs.berkeley.edu/diva/>

[5] The Eclipse Foundation <http://www.eclipse.org/>

[6] Philip Elmer-DeWitt. Tech Fortune in association with CNN Money [August 7, 2010] "Big Macs on campus" <http://tech.fortune.cnn.com/2010/08/07/big-macs-on-campus/>

[7] Michael T. Grinder. 2002. Animating automata: a cross-platform program for teaching finite automata. In Proceedings of the 33rd SIGCSE technical symposium on Computer science education (SIGCSE '02). ACM, New York, NY, USA, 63-67.

[8] Grinder M. T. 2003. A preliminary empirical evaluation of the effectiveness of a finite state automaton animator. In Proceedings of the 34th SIGCSE technical symposium on Computer science education (SIGCSE '03). ACM, New York, NY, USA, 157-161.

[9] Hopcroft, J. E., and Ullman, J. D. Introduction to Automata Theory, Languages, and Computing. Addison-Wesley, 1979.

[10] JFLAP Website. "JFLAP Version 7.0" [May 30, 2011]. <http://www.jflap.org/>

[11] Lars Bak channel9.msdn.com "Expert to Expert - Erik Meijer and Lars Bak: Inside V8 - A Javascript Virtual Machine" [April 29, 2009] <http://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Erik-Meijer-and-Lars-Bak-Inside-V8-A-Javascript-Virtual-Machine>

[12] Lawrence, A. W., Badre, A. N., and Stasko, J. T. Empirically evaluating the use of animations to teach algorithms. Tech. Rep. GIT-GVU-94-07, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, 1994.

[13] Mayer, R. E. Systematic thinking fostered by illustrations in scientific text. Journal of Educational Psychology 81, 2 [1989], 240–246.

[14] Mayer, R. E., and Anderson, R. B. The instructive animation: Helping students build connections between words and pictures in multimedia learning. Journal of Educational Psychology 84, 4 [1992], 444–452.

[15] McConnell, J. J. Active learning and its use in computer science. In Proceedings of the conference on Integrating technology into computer science education [1996], ACM Press, pp. 52–54.

[16] ObjectAid.com “The ObjectAid UML Explorer for Eclipse” <http://www.objectaid.com/>

[17] Oracle Java Applet Documentation. “The APPLET Tag”. Oracle Documentation. Oracle and/or its affiliates. [1995-98]. <http://docs.oracle.com/javase/1.4.2/docs/guide/misc/applet.html>

[18] Oracle Corporation and/or its affiliates “Getting Started” Oracle Documentation [2012]. <http://netbeans.org/kb/docs/java/quickstart-gui.html#getting_started>

[19] Oracle Corporation and/or its affiliates “Integrating an Applet in a Web Application” Oracle Documentation [2012]. < http://netbeans.org/kb/docs/web/applets.html>

[20] Oracle Corporation and/or its affiliates “How to Write an Action Listener” Oracle Documentation [2012]. <http://docs.oracle.com/javase/tutorial/uiswing/events/actionlistener.html>

[21] Oracle Corporation and/or its affiliates “How to Write a Mouse Listener” Oracle Documentation [2012]. <http://docs.oracle.com/javase/tutorial/uiswing/events/mouselistener.html>

[22] Sipser, M. Introduction to the Theory of Computing, Second Edition. 2006 Courser Technology, Cengage Learning. [23] Stasko, J. T. Using student-built algorithm animations as learning aids. Tech. Rep. GIT-GVU-96-19, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, August 1996.

[23] Stasko, J. T. Using student-built algorithm animations as learning aids. Tech. Rep. GIT-GVU-96-19, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, August 1996.

[24] w3schools.com OS Platform Statistics [2012]
<http://www.w3schools.com/browsers/browsers_os.asp>

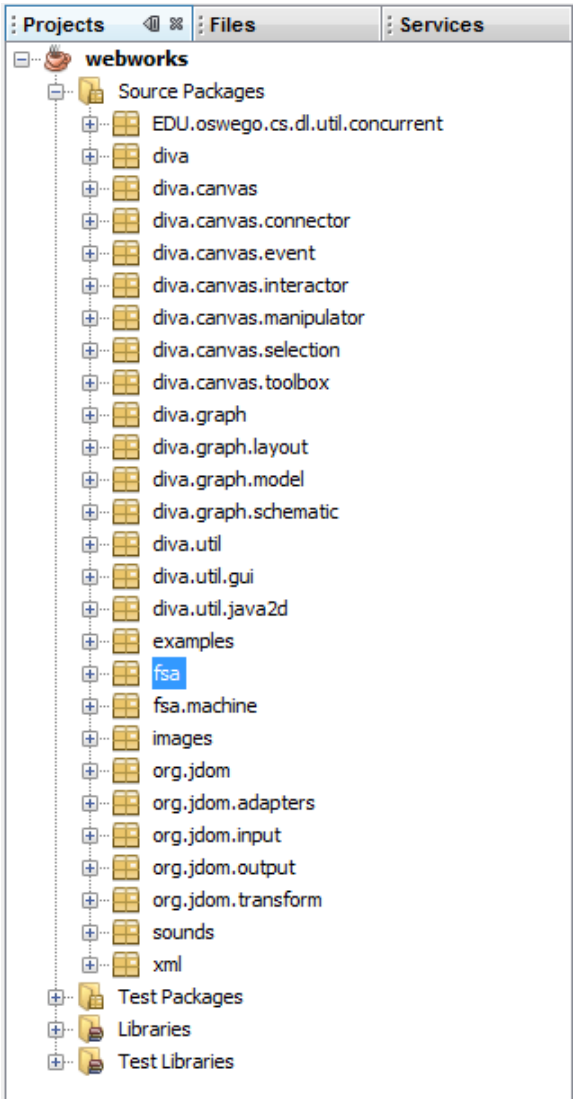
[25] Khronos.org “WebGL - OpenGL ES 2.0 for the Web” [2012]
<http://www.khronos.org/webgl/>

APPENDIX A: DEVELOPER'S GUIDE

Dependencies on Libraries

The successful compilation of the project depends on two libraries: jdom.jar and jython.jar. The following is a description of how to add these libraries to the project so that the code compiles.

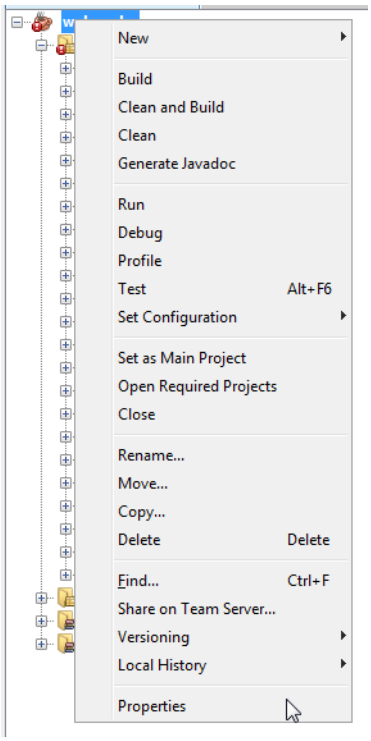
First, the setup of the project, in terms of packages, in NetBeans should be as in the image below:



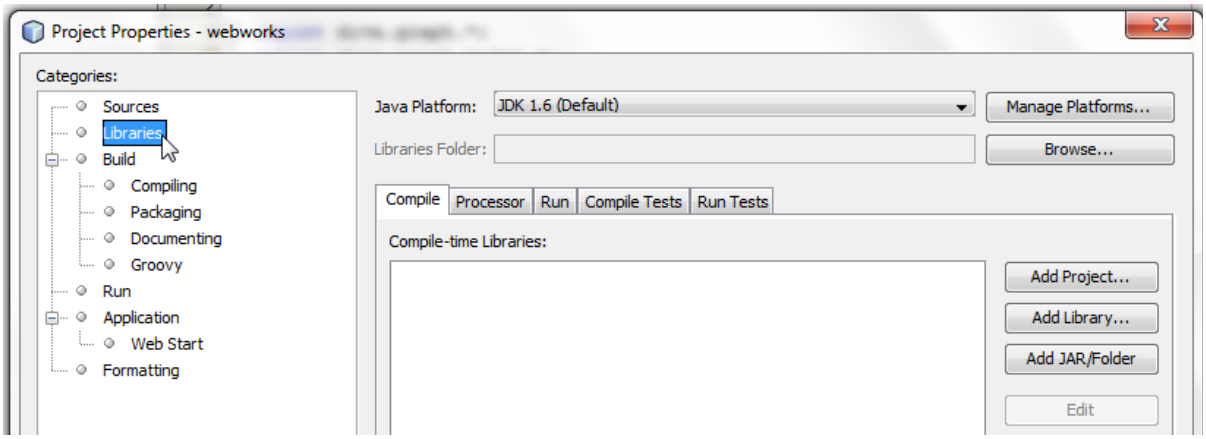
This is critical for the project to compile and run in NetBeans. The diva, EDU, examples, images, sounds and xml folders can be found in the build subfolder. The fsa and fsa.machine folders are taken from the src folder, since it is this folder that contains the .java files.

In order to import the required libraries, follow the following steps, as an example for the jdom libraries:

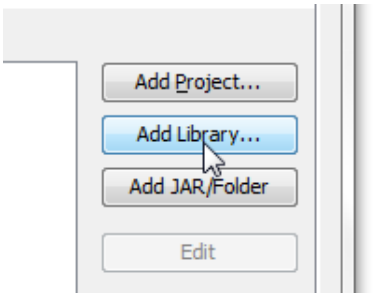
1. Right click the project and select properties



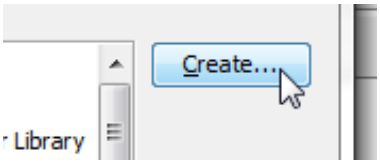
2. Navigate to the Libraries pane



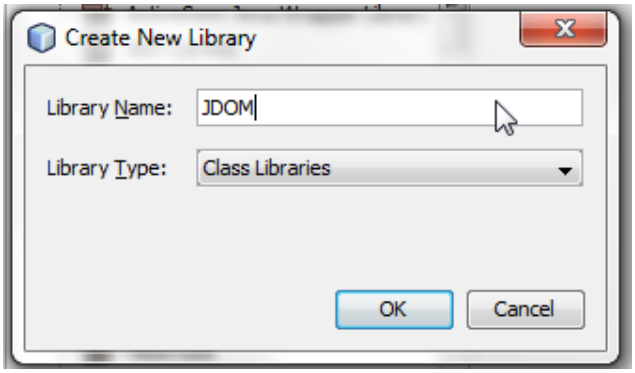
3. Click Add Library



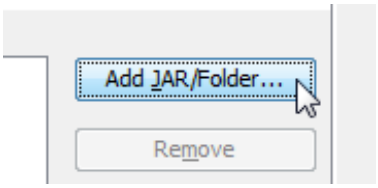
4. Click Create



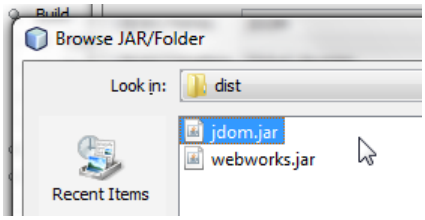
5. Give the library an appropriate name



6. Click Add JAR/Folder

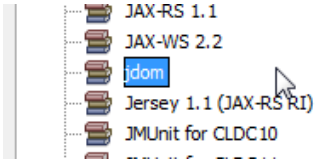


7. Identify the location where the JAR file is located, and select it

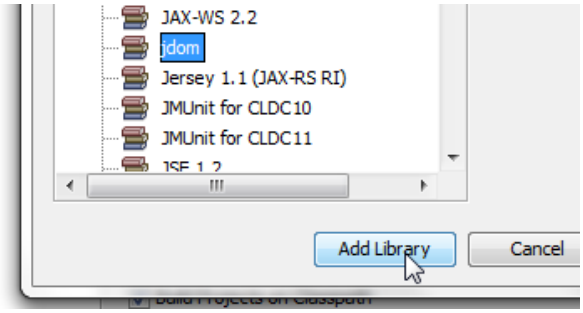


In this case, the file is called jdom.jar, and it located in the lib subfolder.

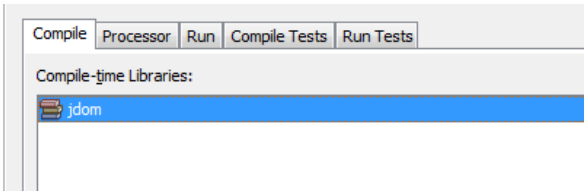
8. Notice that the library will appear in the list



9. Select the library and click Add Library

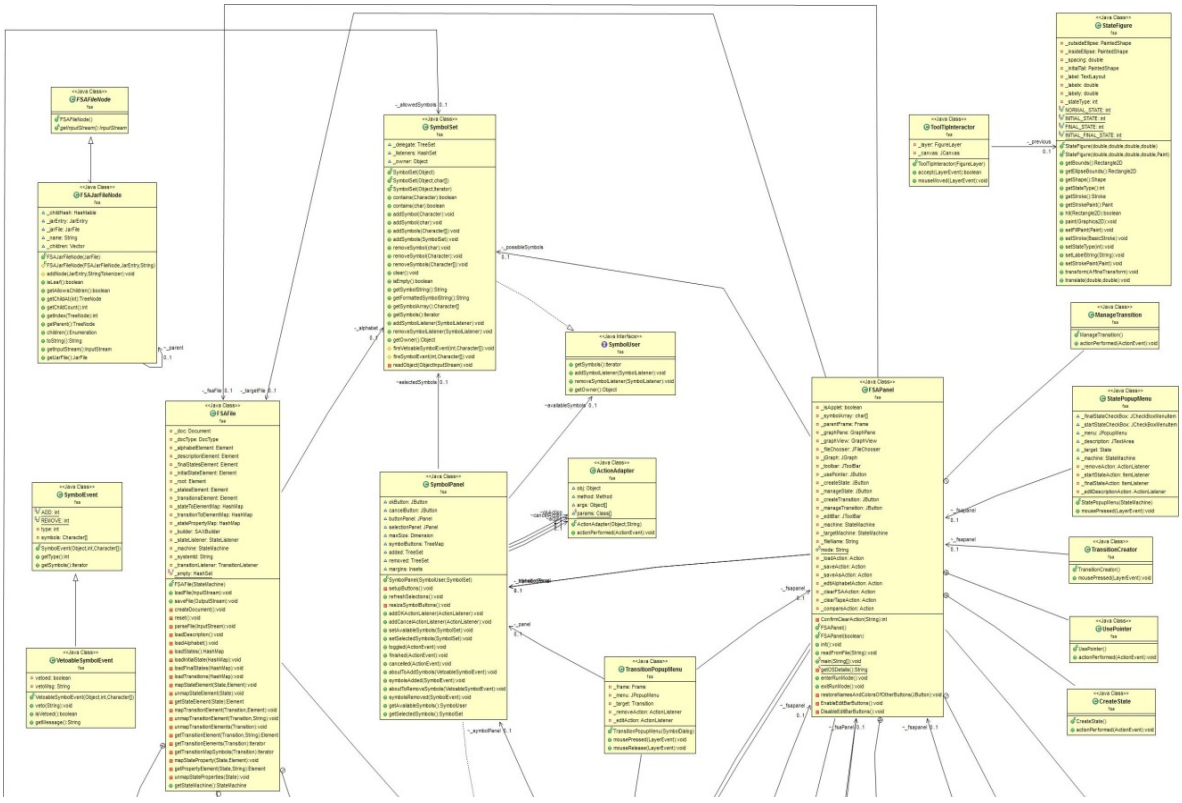


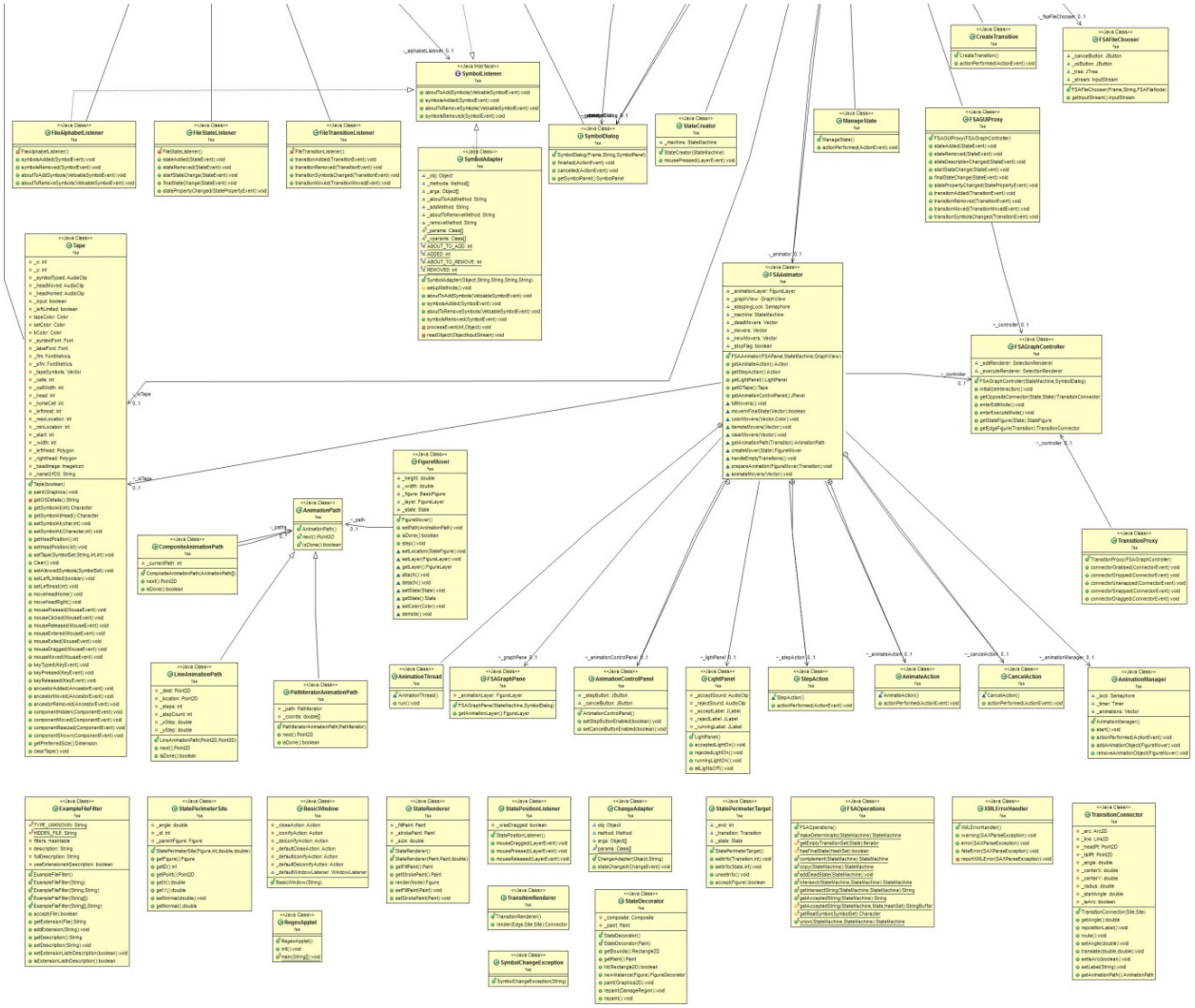
10. Notice the library appears in the list of Compile-time Libraries



UML Class Diagram

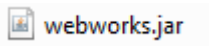
The UML Class diagram below is auto-generated by the ObjectAid plugin [16] for Eclipse [5]. The author spent a considerable amount of time rearranging the classes on the diagram in order to maintain a level of order and readability. The aim of producing this diagram is to aid in the eventual rewrite of the system by showing the association and dependency between the classes of the simulator. Even though the class diagram is virtually unreadable in this report, a high-resolution image along with the .ucls file, which is the class diagram file generated by ObjectAid, will be submitted along with the project deliverables.





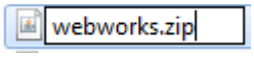
Tip for Viewing the Contents of a jar File

You can treat a .jar file as a zip file. Simply change the extension of the file from .jar to .zip (by renaming it). You can then extract the zip to reveal all its contents.

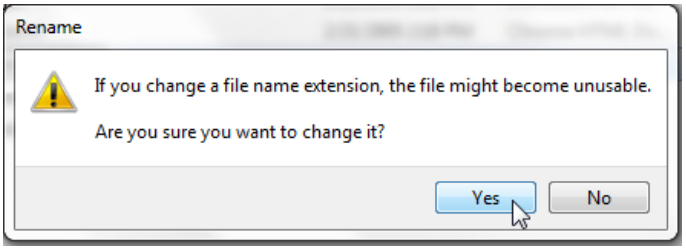
Identify the jar 

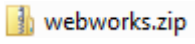
file

Create a copy of it and then rename it changing it the extension from .jar to .zip



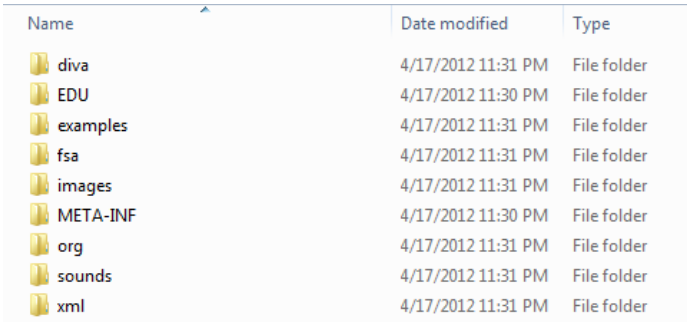
Don't let this intimidate you:



You now have your zip 

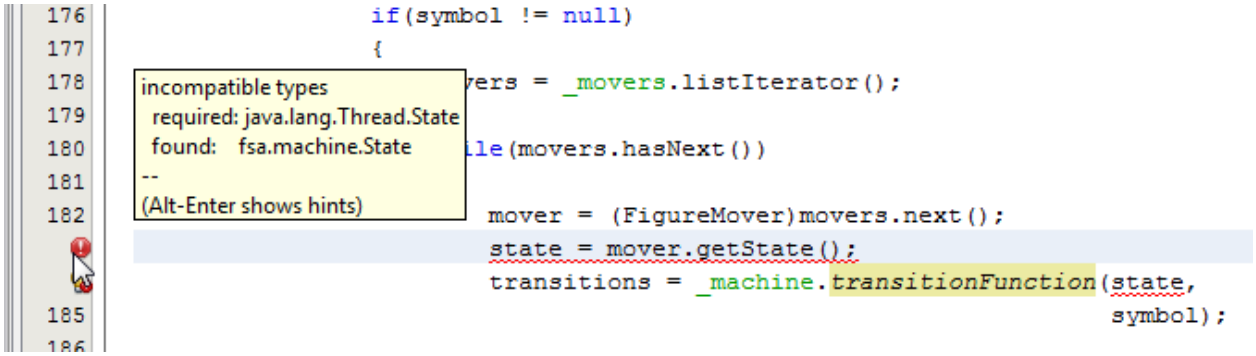
file

Which you can extract into a folder to reveal its contents:



Compile Error

One of the compile errors faced by the author in setting up the project source code in NetBeans is shown below. An example of where this error occurred was on line 182 in FSAAnimator.java.



This error was resolved by changing State state; to fsa.machine.State state on line 143 of FSAAnimator.java

APPENDIX B: LOADING THE APPLET IN A WEBPAGE

The FSA Simulator can be executed as a stand-alone application or embedded as an applet in a webpage and displayed in a web browser. When run as a stand-alone application, the simulator can read and write files on the local file system. When used as an applet, the security restrictions imposed by browsers (at the time this tool was developed) prevent the simulator from reading or writing files on the local file system. Other than this difference, the operation of both versions is the same.

The FSA Simulator contains a toolbar that has a set of buttons displayed. In both application and applet versions of the Simulator, buttons for executing the FSA on the contents of the input tape, for clearing the state diagram panel, for clearing the input tape, and for modifying the FSA's alphabet—labeled Run, Clear FSA, Clear Tape, and Alphabet, respectively—are available. If the simulator is being run as a stand-alone application, additional buttons for loading prebuilt FSAs from a file and for saving the currently displayed FSA to a file—labeled Load, Save, and Save As—are also displayed.

To load the Simulator as an applet, create a folder with an HTML page and the webworks.jar file in it. The HTML page should contain the following code embedding the applet into the page.

```
<object type="application/x-java-applet" height="580" width="790">
      <PARAM NAME="code" value="fsa.FSAPanel" />
      <PARAM NAME="archive" value="jdom.jar,support.jar,sax-
             utils.jar,webworks.jar" />
      <PARAM NAME="align" value="baseline" />
      <PARAM NAME="FSAfile" value="../StarterFSAs/attempt.fsa" />
      <PARAM NAME="TargetFile" value="../TargetFSAs/solution.fsa" />
</object>
```

Now let us examine each attribute of the code.

```
NAME="code" value="fsa.FSAPanel"
```

The code attribute gives the name of the file that contains the applet's compiled Applet subclass. This file is relative to the base URL of the applet. It cannot be absolute.

```
NAME="archive" value="jdom.jar,support.jar,sax-utils.jar,webworks.jar"
```

This OPTIONAL attribute describes one or more archives containing classes and other resources that will be "preloaded". The classes are loaded using an instance of an AppletClassLoader with the given CODEBASE. For security reasons, the applet's class loader can read only from the same codebase from which the applet was started. This means that archives in *archiveList* must be in the same directory as, or in a subdirectory of, the codebase.

```
NAME="align" value="baseline"
```

This OPTIONAL attribute specifies the alignment of the applet. The possible values of this attribute are the same as those for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.

```
NAME="fsafile" value="../StarterFSAs/attempt.fsa "
```

This attribute specifies the FSA file that is preloaded when the applet is run. In this case it is called attempt.fsa, and located in the xml subfolder. This can be used to give students a head start with an assignment, or ensure that all the students start off on the same page. Fsafile is an optional attribute.

```
NAME="targetfile" value="../TargetFSAs/solution.fsa "
```

The targetfile attribute specifies the name and location of the FSA file that is used to compare the newly constructed FSA file against. For instance, when a professor assigns a homework assignment, this would be the correct FSA that he/she prepares, and against which students will compare their FSAs to check the validity of their solutions. As in the case of the fsafile attribute, above, this file is also located in the xml subfolder. targetfile is an optional attribute.

[17]

Note to Instructor: Initially it was thought that the starter and target fsa files had to be included in a folder and compiled with the jar file. However, this is not the case. The starter and target fsa files can be placed anywhere, with the correct path placed in the value of the fsafile and targetfile parameters, as shown above.

In this example, `NAME="fsafile" value="../StarterFSAs/attempt.fsa"`, the starter file is called attempt.fsa and is placed in the StarterFSA subfolder which is located in the same folder where the webworks.jar and html webpage are located. Note that the ".." causes the

system to step out of the jar file (which is a zip folder) and into the folder where the webworks.jar file is placed.

Similarly, in this example, `NAME="targetfile" value="../TargetFSAs/solution.fsa"`, the correct solution preloaded into the applet, against which students will compare their answers is called solution.fsa. The file is placed in the TargetFSAs subfolder which is in the same folder that contains the webworks.jar file and the html webpage containing the applet.

This is important because initially it was assumed that the fsa files had to be compiled with the webworks jar file. This would have meant that the instructor would have to recompile the webworks jar file every time they wanted to change the fsa file, or, more likely, find a way to replace the fsa files in the webworks.jar file, by treating it as a zip folder. That is not necessary any more, hence greatly reducing the amount of time and effort that an instructor would have to spend to set up the starter and target files for the students.