ANIMATING THE CONVERSION OF NONDETERMINISTIC FINITE STATE AUTOMATA TO DETERMINISTIC FINITE STATE AUTOMATA

by

William Patrick Merryman

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

January 2007

APPROVAL

of a thesis submitted by

William Patrick Merryman

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic content, and consistency, and is ready for submission to the Division of Graduate Education.

Dr. Rockford Ross

Approved for the Department of Computer Science

Dr. Michael Oudshoorn

Approved for the Division of Graduate Education

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis 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 thesis in whole or in parts may be granted only by the copyright holder.

William Patrick Merryman

January, 2007

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

LIST OF FIGURES

LIST OF FIGURES - CONTINUED

Figure Page

ABSTRACT

Many students may find the conversion of non-deterministic finite state automata into deterministic finite state automata to be difficult. Since standard computers are deterministic by nature, it is beneficial to understand how to convert nondeterministic finite state automata into equivalent deterministic versions. This conversion process is often inadequately presented in traditional textbooks, though, as static presentations rarely capture the dynamics of the process.

This thesis provides a Java applet that will help students better understand the conversion, and may help teachers to better present the conversion. This applet was designed to be beneficial to student learning; therefore, many different learning modes were included. These modes vary the speed and level of detail of the conversion so that students may proceed at their own pace. Each step of the conversion is animated and the student is constantly made aware of what has just happened and what is going to happen next, along with how the step will be performed. If the student is asked by the applet to supply anything, the applet will provide the student with feedback on his or her mistakes or successes.

This non-deterministic finite state automaton to deterministic finite state automaton conversion applet will be added to the Webworks hypertextbook project. This project is designed around the premise that animating algorithms will lead to better learning.

INTRODUCTION

Conversion algorithms for non-deterministic finite state automata to deterministic finite state automata can be difficult for students of computer science theory to learn. The goals of this thesis are (1) to develop a visual, animated software system to help students better learn and understand one such conversion algorithm, and (2) to develop a strategy for evaluating the efficacy of the developed system on student learning. The hope is that through a series of animated examples and exercises incorporated into a teaching and learning resource in support of computer science theory courses, students will be able to learn the selected conversion algorithm and be able to apply it on their own.

The software developed for this thesis is part of a much larger hypertextbook project by Ross on the theory of computing [2]. Ross describes a hypertextbook this way:

> The hypertextbook is a novel teaching and learning resource built around Web technologies that incorporates text, sound, pictures, illustrations, slide shows, video clips, and most importantly, active learning models of the key concepts of the theory of computing into a single, integrated source [2].

The theory hypertextbook will thus integrate active learning applets of various models of computation and their related grammars, conversions, and proofs into a cohesive whole suitable for use by both instructors and students. The hypertextbook will be accessible through a standard web browser and and the applets within will have a common interface.

To be published as *Theory of Computing: The Hypertextbook*, this novel resource will include active learning applets for finite state automata, pushdown automata, Turing

machines, and many other aspects of the theory of computing. The hypertextbook is intended to augment or replace traditional hardcopy textbooks. Although traditional textbooks can include descriptions of models of computation, the presentations are static and often not particularly effective learning tools for many students. It is true that an instructor can present models of computation and related algorithms in class on a board or an overhead projector, effectively animating the dynamic aspects of these models and algorithms; however, this approach does not seem to help students visualize these same processes outside of the classroom [1] [2] [8].

The effort required to integrate animated algorithms into an instructor's curriculum has always been one of the major inhibitors of the use of these animations [2]. Instructors often have packed schedules, which makes it difficult for them to find the time to learn about and integrate animated algorithm software into their curriculum. Since the theory hypertextbook will include active learning applets that animate dynamic theoretical computer science concepts, instructors will be able to spend less time preparing and integrating these applets into the classroom and more time focusing on the underlying theory. The hypertextbook is being designed to allow the active learning applets to be easily integrated into the classroom and homework assignments, as all the applets will have a standard interface that should be intuitive and easy to learn and use.

Students will also benefit from the inclusion of active learning applets within the hypertextbook. Since instructors are pressed for time they can rarely take time to go over more than one or two examples of a concept in the classroom. Using the applets, students will now be able to not only review the examples used in class, but to also view others

that were not covered in class, helping to increase the depth of understanding. Not only will students be able to review examples, they will also have exercises and homework assignments that will utilize the applets provided with the hypertextbook.

In section 2 related work is discussed. Section 3 provides an in-depth look at the conversion algorithm developed as part of this thesis. In section 4 the applet that performs the conversion is presented. Then section 5 discusses how to evaluate the effectiveness of the animation as a learning aid and how it can be used by instructors. Section 6 discusses future work.

THESIS

The purpose of this thesis is twofold. The first is to show that an animation applet can be built that demonstrates the non-deterministic FSA to deterministic FSA conversion algorithm. This purpose itself is multifaceted as this applet needs to have multiple ways that users may interact with it. It must have multiple example modes that run at different speeds and levels of detail that are fed back to the student. It must also have exercises that can be used by students for practice and by instructors as homework assignments. It must also have a means for instructors to create their own examples and exercises.

The second purpose of this thesis is to test the effectiveness of the aforementioned applet. This would be done most effectively within a class setting with both a test set and a control set of students. The test set would use the applet, but the control set would not have access to the applet. It would have to be incorporated into the classroom and out of the classroom for maximum testing effectiveness.

The applet constructed as the first part of this thesis was completed and is fully operational. The second part (class testing) was not completed as there was no theory class in progress in which the applet could be tested. An evaluation instrument was completed, however, and will be utilized in a future test of the effectiveness of the applet on student learning.

RELATED WORK

There are three types of animation software: program animators, algorithm animators, and concept animators [5]. Program animators work much like debugging software systems, displaying register, variable, and stack values of a program in execution, as the section of code being executed is highlighted. They usually have additional features that are generally not found in debuggers such as the capacity to back up through the processing during a code walkthrough, and a level of detail in the graphical presentation suitable to novice learners rather than expert programmers. Algorithm animators provide graphical, dynamic displays of how an algorithm works on a data set, such as a graphical display of a heap as a tree along with insert and delete operations being performed on learner-provided input. Concept animators are used to animate abstract concepts, such as automata for a computer science theory course, the instruction fetch and execute cycle of a computer for a computer architecture course, and virtually any other dynamic process in any academic discipline. In this thesis we will mainly be covering concept animators as that is what applies most to this particular project.

Webworks Laboratory Projects

The Webworks project of Ross [2] [3] has resulted in many concept animators for use in a computer theory or compilers course. The goal of the Webworks project is the development of a complete theory hypertextbook. This complete hypertextbook is to be disseminated on CD or DVD media and will be accessible through a standard web

browser. As a result, students and professors will have seamless access to the embedded concept animators without the hassle of locating and using individual animations on the web, which may not mesh well with the class textbook. This will solve one of the main problems restricting widespread use of animation software, namely that instructors feel that finding and incorporating web-based animations into their lessons is too time consuming. It will also help motivate students, who often view the animators that don't mesh well with a course as involving extra work without having a distinct and important role in the learning process [1].

The nondeterministic to deterministic finite state automaton conversion algorithm central to this thesis is thus not intended to be a standalone system (although it can be used in this fashion). Rather it has been designed to be integrated into the theory hypertextbook of Ross, along with many other concept animators.

FSA Animator

For his PhD dissertation Michael Grinder created an FSA animator applet. This applet forms the basis for the animator of this thesis and thus is covered in detail here. The FSA animator of Grinder was designed to be a learning tool for visualizing the operation of an FSA.

The FSA animator was designed with three main features in mind. The first two are closely tied together: to build FSAs and to be able to watch an FSA process an input string. The third feature provides for exercises that give the user a language and ask them to build an FSA that recognizes that language. A correct machine in the background is compared to the user's created machine to test for correctness.

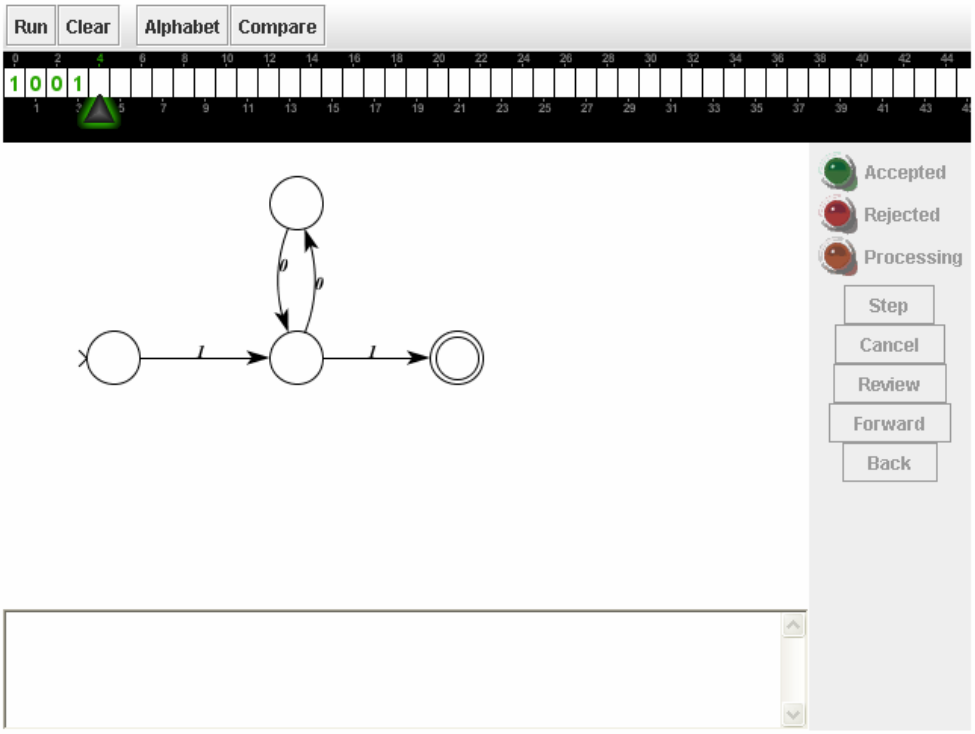

Figure 1. FSA Animator

As can be seen in Figure 1, the FSA animator has a simple interface. On the top is a series of clickable buttons; *Run* engages the machine to process the input string; *Clear* pops up a menu to clear the tape or the FSA; *Alphabet* pops up a menu to set the input alphabet for the FSA; and *Compare* allows users to check whether an FSA they have constructed is correct. Beneath these buttons is the input tape, with a green triangle that serves as the input tape head. All input for the tape is done through the keyboard, but key presses will only be accepted if they are symbols in the alphabet of the FSA. The main pane in the applet window contains the FSA.

To the right of the FSA panel is a control panel with some lights and buttons. If the input is accepted a green light is activated, if an input string is rejected the red light will be activated. While the tape is being processed a yellow light remains lit. The *Step*

button causes the FSA to process the next symbol on the input tape; the *Cancel* button will stop the processing of the FSA; the *Review* button activates the *Back* and *Forward* buttons and allows the user to review what the FSA has processed; they control the review process by allowing the user to move backwards and forwards through the currently processed tape. The bottom panel is a text box that can be used to give the user instructions or feedback when operating the FSA. Currently for the FSA animator the review state and the text box remain unused, although they are used extensively in the NFA to DFA conversion process.

Building an FSA is a straightforward process. To create a new state, the user need only press the Ctrl key and left click with the mouse on the desired position for the new state. Right clicking on the state brings up a menu that allows the user to label the state and establish it as a start and/or final state. In order to create transitions the user presses the Ctrl key, left clicks on the source state of the transition, and drags the mouse pointer to the target state of the transition. Right clicking on the transition allows the user to set the transition alphabet symbol.

An example of an FSA processing an input string can be seen in Figure 2. The green triangle on the input tape points to the symbol currently being processed by the FSA. As can be seen, the yellow processing light is activated and the *Step* and *Cancel* buttons have been highlighted. The red dot in the FSA indicates the current state of the FSA.

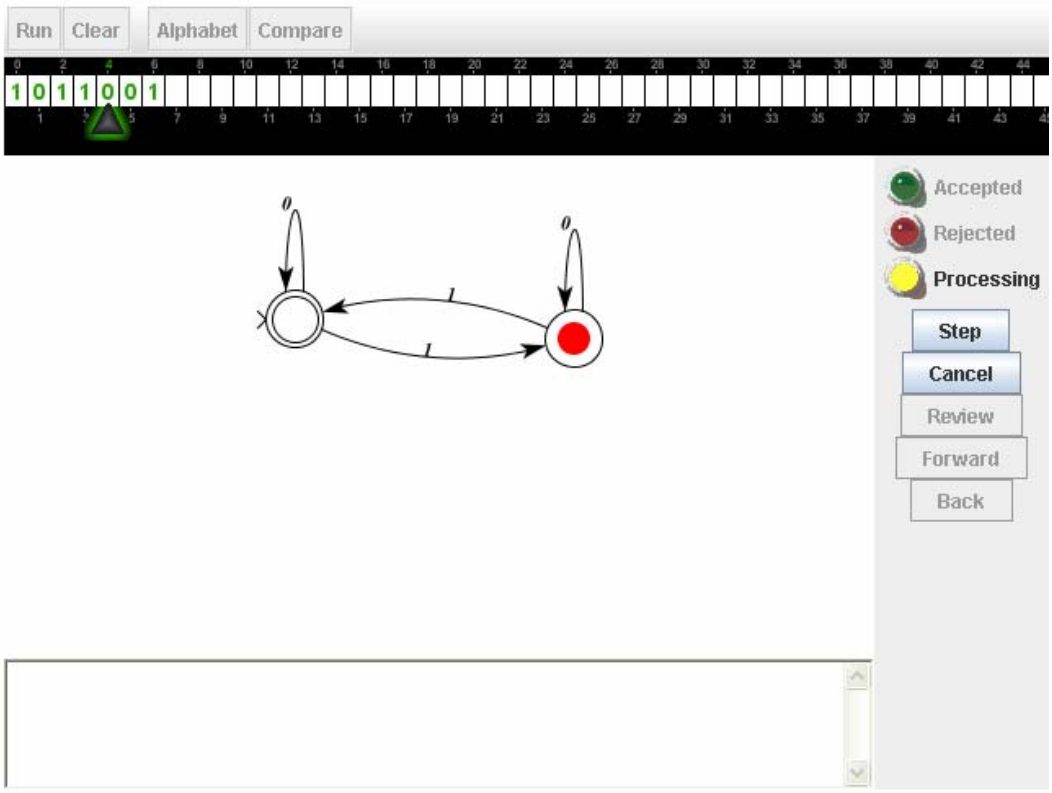

Figure 2. An FSA processing an input string

The *Compare* button provides a unique capability useful for exercises in which students are required to construct an FSA that recognizes a particular regular language. An instructor can develop such an exercise and provide a correct, unseen FSA as part of the exercise. Students then attempt to construct a correct FSA for the exercise, starting with a blank panel. At any point during the construction process they can click on the *Compare* button and a message will pop up telling them whether their FSA is correct or providing them with example strings for which their machine's acceptance differs from the background machine's. Since testing for equivalence between two FSA's is decidable, an algorithm exists that determines whether a student's constructed FSA does or does not recognize the language of the exercise. This is mentioned because the related

pushdown automaton applet (discussed below) developed as part of the theory hypertextbook project does not share this property.

An example of the FSA exercise applet in action is shown in Figure 3. The exercise in question requires students to construct an FSA that recognizes the language of binary strings that have even parity. Figure 3 shows an attempted FSA and the message that popped up when the *Compare* button was clicked. A correct FSA is shown in Figure 4.

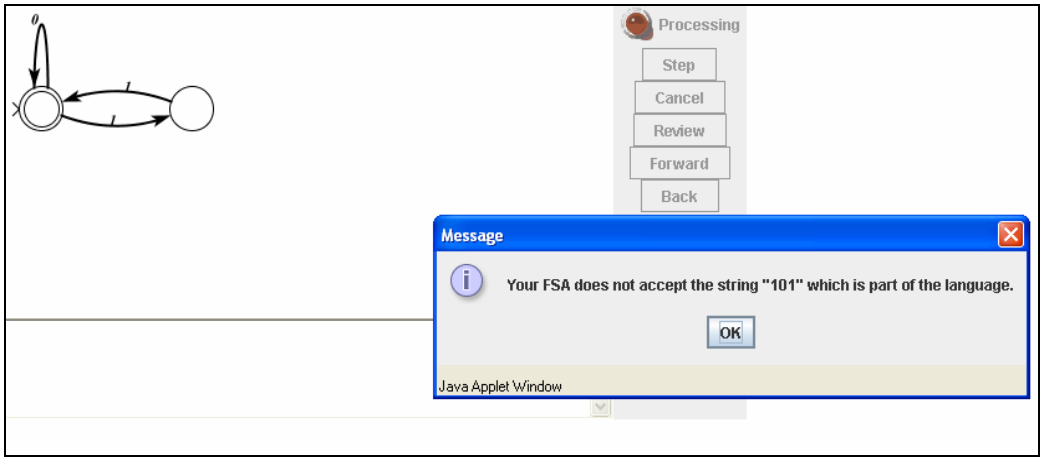

Figure 3. An example of an error when building an FSA to recognize a particular language

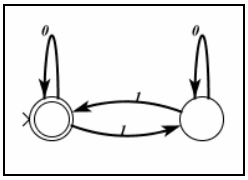

Figure 4. The FSA that recognizes the correct language

Pumping Lemma Animator

Josh Cogliati extended Grinder's FSA animator with features for animating the pumping lemma for regular languages. Cogliati's work added four new modes to the

FSA animator that allow users to visualize the basis for the pumping lemma. The first two modes are very similar; for an FSA M and a string $w \in L(M)$ the first illustrates a way of dividing w into x, y, and z sections, where $w = xyz$, such that pumping y results in new strings in the language as described by the pumping lemma [4] [7]. The second adds the capability for users to select a subsection of w to break into the x, y, and z sections. The third mode allows users to select a portion of w that will cause the FSA to loop. The applet will then check to see that the selected substring does indeed cause a loop and report whether the selection was correct or not. There is a sub-option for this mode which determines how much help the applet gives the user in finding these loops.

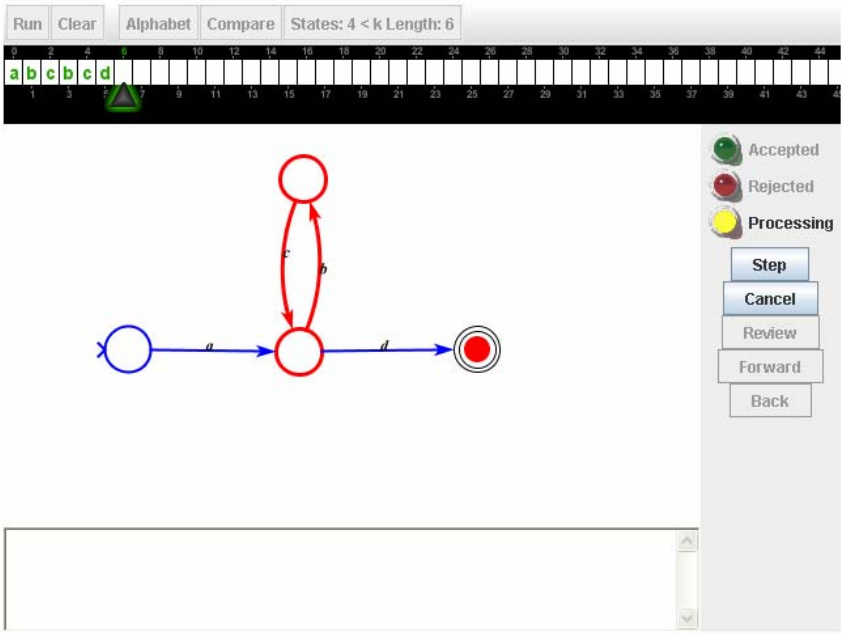

Figure 5. Pumping Lemma showing repeated states

The final mode allows the user to see states that have been repeated more easily, by highlighting states and transitions that have previously been encountered as the FSA

processes an input string. Figure 5 shows a snapshot of an FSA running in this mode. There are darker red lines on those states and transitions that have been repeated.

Pushdown Automata

Cheston Williams worked on an animator for pushdown automata. He has created two modes for his animator: an example mode and an exercise mode. In the example mode the user is shown a PDA and asked to give the PDA an input string to process. The applet shows both the finite state component and the stack of the animated PDA. Since the PDA applet allows for nondeterminism, if the PDA is in multiple states at once, stacks for each active state are shown. The stacks and the active states are color coded: red is active, grey is dead, and green means that the input was accepted. Dead stacks (resulting from unproductive nondeterministic branches) are removed. Figure 6 shows a PDA that accepts the language $(0^n1^n \mid n \geq 0)$ running on the input string 0011. The states q1, q3, and q4 are dead and states q2 and q3 are active. Notice that state q3 is both active and dead, which is caused by nondeterminism, (although the user can't see the dot on the state there is a grey dot beneath the red dot).

In the exercise mode, the user is asked to build a PDA that recognizes a given language. Since it is undecidable whether two PDA's recognize the same language, the applet runs the user's constructed PDA on a test suite of valid and invalid strings, providing feedback about the correctness of the PDA. Although this cannot guarantee that a user's given PDA is correct, it does provide some assurance that the PDA is correct if all strings pass the test.

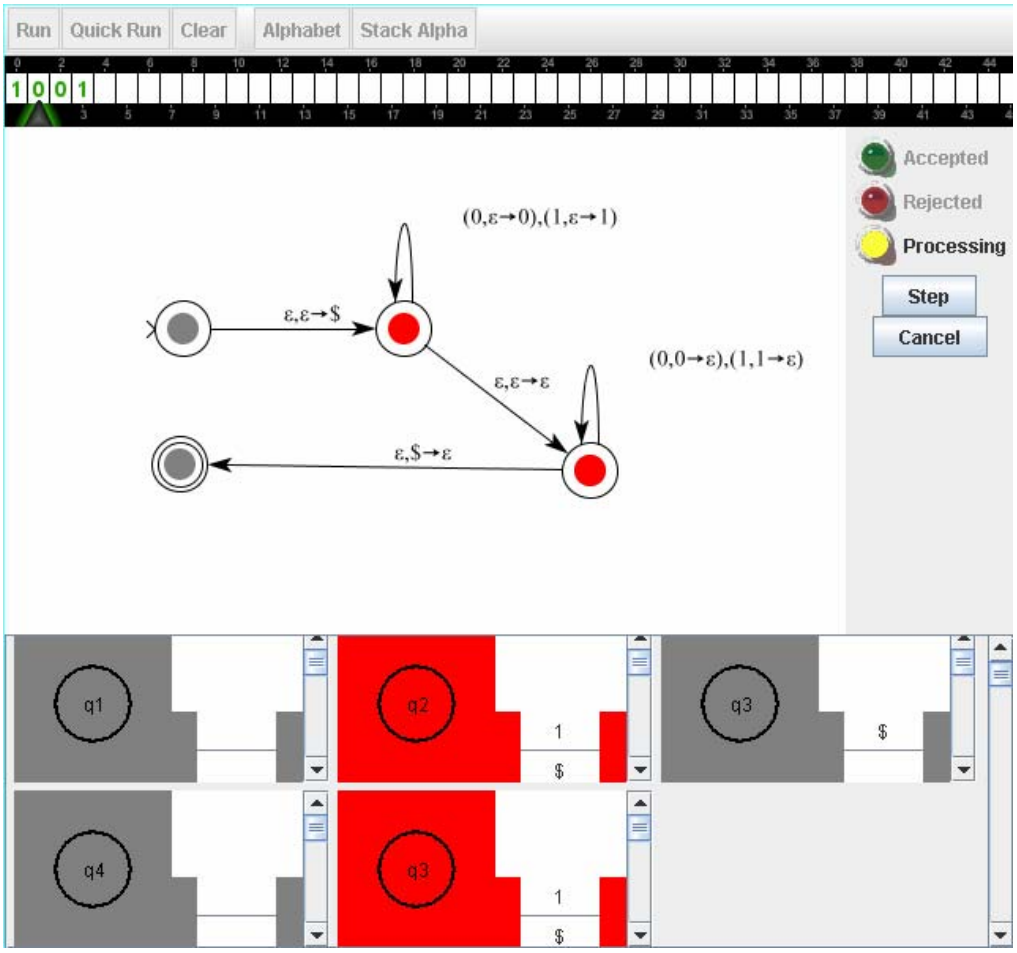

Figure 6. An Example of the PDA Applet

Regular Expression Animator

Katie Walsh developed an animator for regular expressions that was later extended by Brad Pascoe [2]. This applet has both a demonstration mode and an exercise mode. In demonstration mode the user is given a language and shown how the regular expression that captures that language could be built. The language is given in a text box near the top of the applet window. Each step that is performed in constructing the regular language is explained to the user in a pane at the bottom left of the applet window. As the regular expression is built it is displayed in the lower right pane. There is a tool bar at

the top left of the applet window with the buttons *Step*, *Reverse*, *Run*, and *Pause*. Figure 7 provides a snapshot of the regular expression animator in action. Clicking the step button causes the next phase of the construction of a regular expression to be displayed. Simultaneously the explanation is updated to tell the user what is happening and why. The reverse button backs up one step in the presentation each time it is pushed. The run button starts the applet running through the entire animation with short pauses at each step. The pause button halts the run mode.

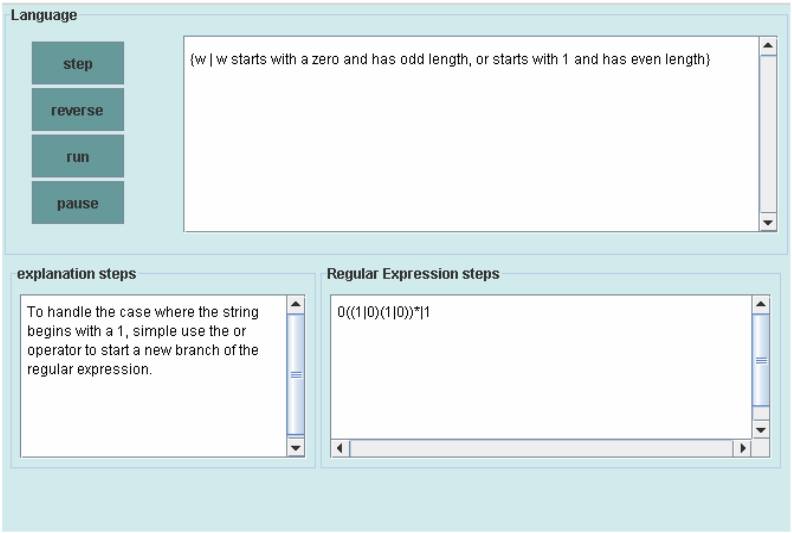

Figure 7. Regular Expression Animator

The exercise mode requires the user to construct a regular expression for a given language. It gives the user the option to build and use auxiliary regular expressions to use in their solution, such as digit=0|1|2|3|4. It also provides feedback on the correctness of a user's proposed solution.

Context Free Grammar Animator

Teresa Lutey developed an animator for building parse trees for context free grammars. This animator allows the user to use a provided grammar from a library or to supply one of their own. Options are available that allow the user to select a nonterminal for expansion and a rule to use for the expansion. Through this means a user can generate parse trees for strings in the grammar. Figure 8 is a snapshot of this applet in execution.

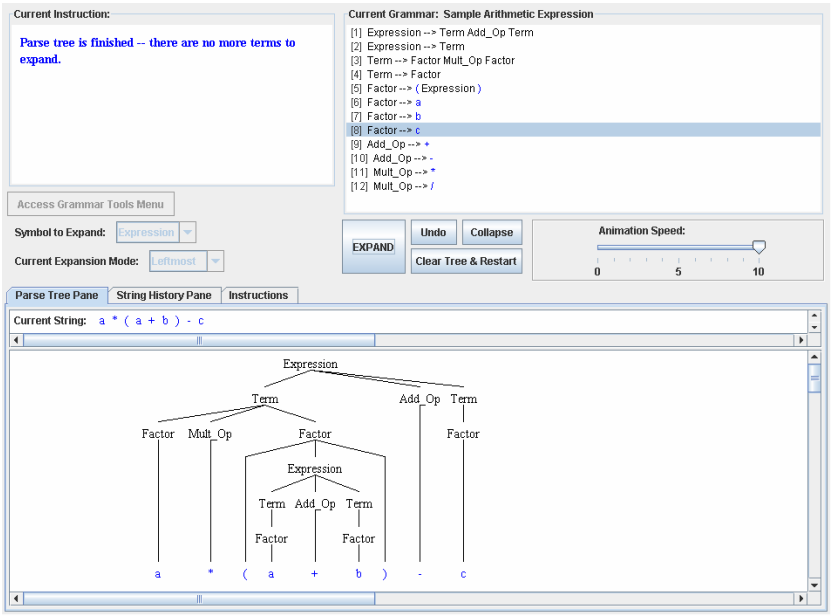

Figure 8. LL(1) Animator

There are many other tools that have already been built and included in the hypertextbook. However these other animation applets were not used as a basis for this thesis work. Many of these other tools have found their way successfully into the classroom and are being used by Ross in his compilers course.

Java Formal Language Automata Package

There is only one other comprehensive software system known to the author for helping students visualize various models and concepts from the theory of computing. Susan Roger has created a collection of automata animations for computer theory called JFLAP (Java Formal Language Automata Package) [6]. JFLAP is a fairly complete tool but is not implemented in a way that allows it to replace a traditional textbook. Although there are help menus that provide information on the use of each of the included models, there is no text for actually teaching the concept. It is more of a practice or homework application, which may not be ideal from a professor's perspective, as he or she must find ways to incorporate JFLAP use into the framework of their courses and make them mesh with a traditional textbook. Otherwise, JFLAP includes useful tools for helping students learn the theory of computing.

JFLAP is made of many computer theory animation applications, which are listed here:

- Finite state automata
- Pushdown automata
- Turing machines
- Multi-tape Turing machines
- Grammars
- L-Systems
- Regular expressions

JFLAP: Converting An NFA Into A DFA

As the focus of this thesis is on the visualizing the conversion of nondeterministic finite state automata to deterministic finite state automata we examine this feature of JFLAP in more detail. The conversion algorithm animated in JFLAP is very straightforward. One creates an NFA and then selects (*Convert to DFA).* Creating the NFA is a simple process in JFLAP.

To create an NFA, the first thing a user does is click on the *State Creator* button in the editor buttons menu. After this, the user simply clicks where he or she wants new states to appear on the screen. Once this is done, clicking the *Attributes Editor* button allows the user to identify and set the final and start states and add any labels for the states that were created. Transitions are added at this point by using the *Transition Creator* button. Transitions are created by clicking on the state representing the start of the transition, and then clicking on the end state for the transition. A label editor pops up, allowing the user to input a symbol for the transition. One inconvenience of JFLAP is that this process must be repeated for each transition desired between the two same states. For example, if the user wants a transition for the symbols 0 and 1, two transitions must be created, one labeled 0 and one labeled 1. Also JFLAP does not require the user to specify an input alphabet. Instead, the alphabet is determined from the labels given to transitions. This would seem to make it more difficult for students to understand the importance of the alphabet. Furthermore, an accidental press of the space bar when entering a transitions symbol, say 0, causes the transition to be labeled with a 0 followed by a space unbeknown to the user, which can lead to great confusion.

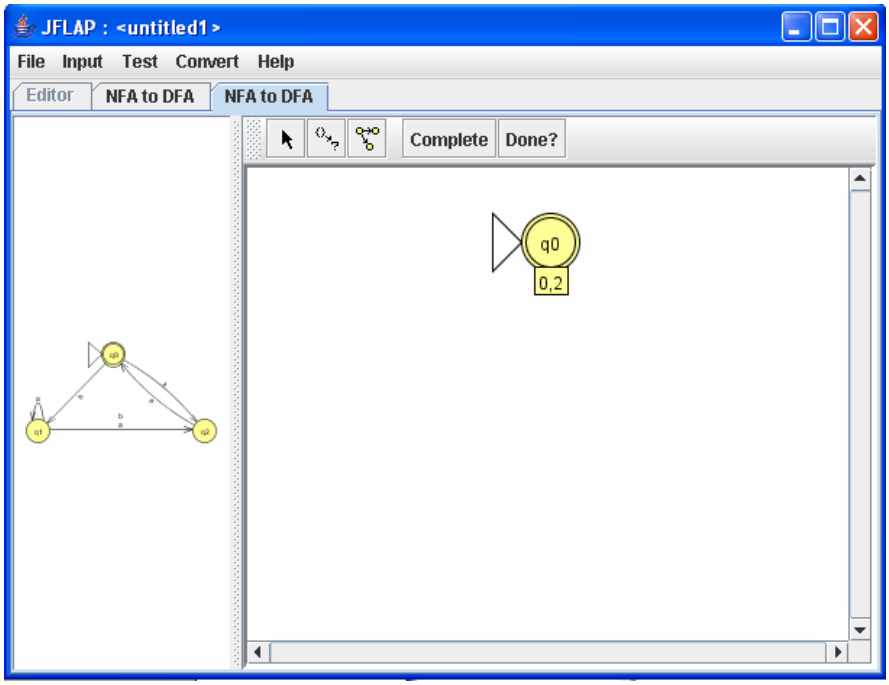

Figure 9. JFLAP at the start of the NFA to DFA conversion

Once an NFA has been created, the user can select *Convert to DFA* from the *Convert* menu (see Figure 9). This will open a new window for doing the conversion. The constructed NFA is in a pane on the left and the current workspace is in a pane on the right. The start state is created automatically; notice the box with 0, 2 in it right below the state. This represents the set of states that the start state is made of.

The algorithm to do the NFA to DFA conversion that Roger is using requires that each state be created as it is needed. If a state has a transition going to multiple states on the same input symbol *x* a new state *A* will be created. This new state represents the set of states that were entered on that input symbol *x*. The transitions that leave *A* will be determined by the set of states that *A* represents. For each input symbol *y*, the states that are entered by each state represented by state *A* will be brought together into a new state

B through a union operation, and *A* will have a transition leaving it and going to *B* with symbol *y*.

There are two options for completing the conversion. The first is the *Expand Group on Terminal* button ( ). Using this option, the user clicks on a state then drags the mouse away from this state. This will open up a dialog box that prompts the user for a terminal symbol, which is a transition label. Once this is entered another dialog box will open asking which state combinations the new transition will go to. If the state combination is new, the state will be created for the user, with a box below it showing the states it is created from. This process is repeated for each terminal symbol on each state. Figure 10 shows JFLAP in action. This screenshot was taken after q0 had transitions added for symbols *a* and *b*, and q1 had a transition added for symbol *a*.

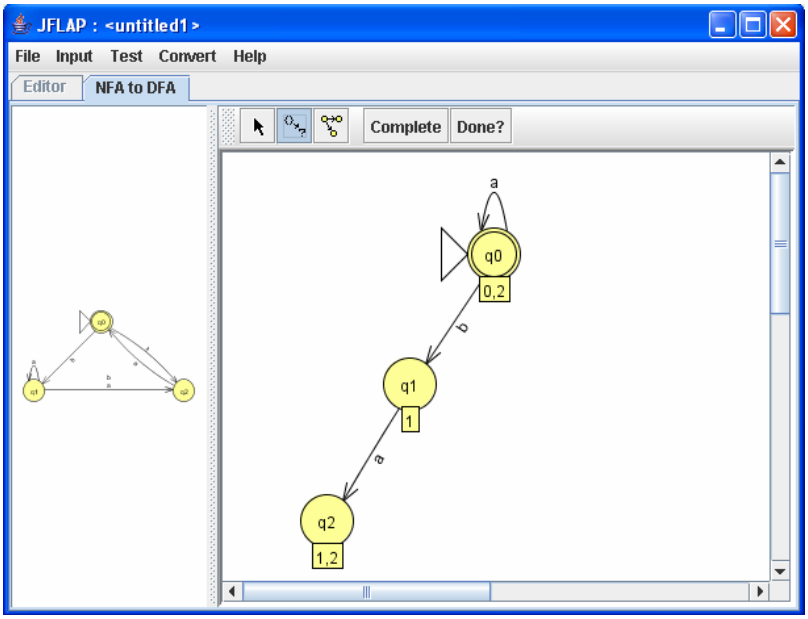

Figure 10. JFLAP using the Expand Group on Terminal button

The second option for doing the conversion results from selecting the *State Expander* button ( ). After doing this, clicking on any state will automatically add all the transitions that this state or state combinations would have. This will also add any new state or state combinations that are needed. Figure 11 shows the machine after states q0 and q1 were selected.

Although JFLAP's presentation of the conversion is quite valid, it does present some awkwardness for professors. Since JFLAP is not designed as part of a textbook, professors who use JFLAP will need a textbook for the class in addition to JFLAP[1]. Most textbooks on the theory of computing would have slightly different algorithms, different notation, and different approaches to the presentation. This might possibly make it hard for students to gain much from JFLAP[2].

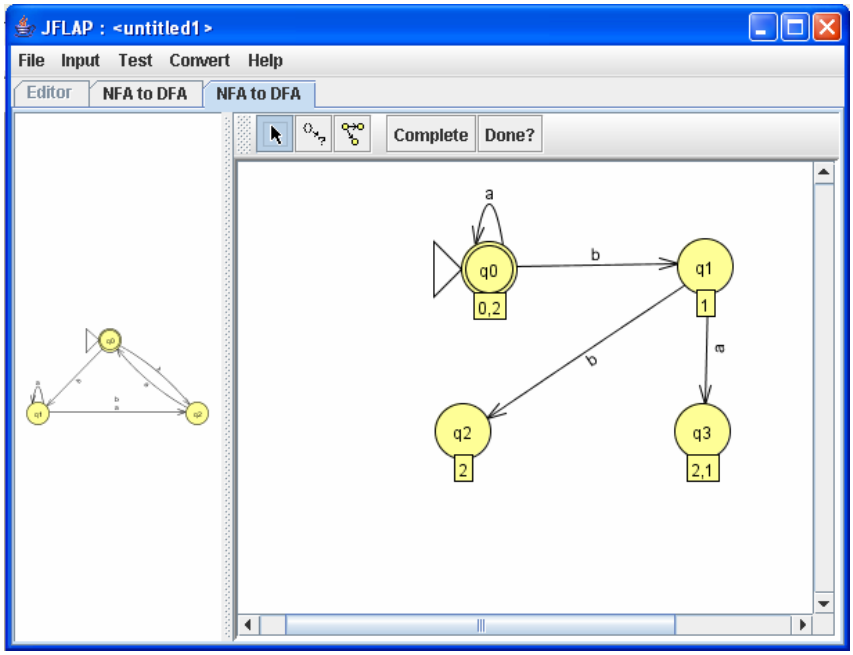

Figure 11. JFLAP after the State Expander button has been pressed

[1] This issue has been partially addressed through the recent publication of a JFLAP manual [9].
[2] This issue has also been partly addressed in that JFLAP has been designed in conjunction with the theory textbook by Peter Linz [10].

Besides Roger's JFLAP, there seems to be little work done for NFA to DFA conversion animation software. There have been a few in the past, but they all appear to be discontinued and the animators have fallen into disrepair.

ANIMATING THE CONVERSION FROM A NONDETERMINISTIC FINITE STATE AUTOMATON TO A DETERMINISTIC FINITE STATE AUTOMATON

In this chapter the scheme developed for this thesis for animating the conversion of a nondeterministic finite state automaton to a deterministic finite state automaton is presented.

Background

Nondeterminism is defined as the opposite of determinism, which Webster defines as a theory or doctrine that acts of the will, occurrences in nature, or social or psychological phenomena that are causally determined by preceding events or natural laws. For a deterministic finite state automaton (DFA) this means that the current state and input symbol uniquely determine the next state. On the other hand, the next state in a nondeterministic finite state automaton (NFA) cannot always be uniquely determined from the current state and input symbol. Nondeterminism is represented in an NFA through one or more empty transitions, more than one transition from a single state with the same non-empty label, or any combination of the two.

There are two commonly used models for animating the operation of an NFA. The first model spawns a new NFA for each nondeterministic branch that can be taken from the current state; all spawned NFA's continue to execute concurrently. The second model maintains just one NFA but allows multiple states to be active at any time. The second model is the basis for Grinder's simulator [4] and is the model used in this thesis.

The Theory

The conversion algorithm presented here follows the one given in Sipser [7]. It is based on the following underlying theory.

Definition 1

A nondeterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states.
2. Σ is a finite alphabet.
3. $\delta : Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function, where *P(Q)* represents the power set of Q and $\sum_\varepsilon = \sum \cup \{\varepsilon\}$, ε being the empty string.
4. $q_0 \in Q$ is the start state.
5. $F \subseteq Q$ is the set of accept states [7]. ⟨

Given an NFA *N*, there are two main approaches to the conversion of *N* to an equivalent DFA *M*. The first, followed in this thesis, is to create $2^{|Q|}$ states, each corresponding to a subset of the states of *N*, and then to build the DFA up from there. The second, which is the basis for the NFA to DFA conversion algorithm of JFLAP, is to build each state in the DFA as it is needed. The second is more efficient, but does not, in the opinion of the author, impart as strong of a sense of the underlying principles behind the conversion as the first approach.

The construction to convert an NFA into a DFA from Sipser [7] is presented here. Suppose that we have an NFA $N = (Q, \Sigma, \delta, q_0, F)$. The algorithm will produce a DFA $M = (Q', \Sigma, \delta', q_0', F')$ such that L(*M*) = L(*N*). The individual states of *M* represent sets of

states in *N*. Intuitively, the single states of *M* represent the collection of states that *N* can be in at any particular stage of a nondeterministic computation on its input.

To proceed we need the following definition.

Definition 2

Let $N = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite state automaton. For any subset $R \subseteq Q$, define $E(R) = \{q \in Q \mid q$ can be reached from any r $\in R$ by traveling along 0 or more empty transitions$\}$ 〈

This definition captures the fact that the set of next states that an NFA can be in given its current set of states and the current input symbol is not only the set of states arrived at by applying the current input symbol to each of the current states, but also any additional states that can be reached from those next states by empty transitions.

We can now state the following lemma.

Lemma 1

Let $N = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic finite state automaton. Construct $M = (Q', \Sigma, \delta', q_0', F')$ such that

1. $Q' = P(Q)$
2. $q_0' = E(\{q_0\})$
3. $F' = \{R \in Q' \mid R$ contains an accept state of $N\}$
4. For $R \in Q'$ and $a \in \Sigma$ let $\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a))$ for some $r \in R\}$ (alternatively this could be represented as: $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a))$)

Then *M* is deterministic and *L*(*M*) = *L*(*N*).

The proof of this lemma is given in Sipser [7].

Note again that in the DFA *M*, states *R* in Q' are elements of the power set of *Q* and hence may represent multiple states in *Q* (i.e., single states *R* in Q' represent multiple states that are subsets of *Q)*. This is key to understanding the conversion algorithm.

The Construction of Lemma 1

An algorithm for converting an NFA *N* to an equivalent DFA *M* can be created by implementing the construction given in Lemma 1. The construction can be better understood through a more descriptive process.

First, it is important to note that the conversion algorithm constructs *M* by modifying *N* rather than constructing an entirely new deterministic FSA from scratch as many other algorithms do. Also, in the conversion algorithm developed for this thesis no distinction is made between the set containing a single state (e.g. $\{q_1\}$) and the state itself (e.g. q_1). This is because the algorithm proceeds by constructing *M* from *N* by including the states of *N* in *M* and adding new states that correspond to non-singleton sets in $P(Q)$.

Step 1

Step one of the construction given in Lemma 1 describes the states that will be constructed for *M* based on *N's* states. Each state of *M* is a subset of the states of *N*. For example, if *N* has states $Q = \{q_0, q_1\}$, *M* would have states $Q' = \{\{\}, \{q_0\}, \{q_1\}, \{q_0, q_1\}\}$.

Although this algorithm creates all possible combinations of states, they are usually not all used.

Step 2

The second step sets the initial state of *M*. This state is the start state (q_0) of *N* unioned with the set of states that can be reached by following one or more ε transitions starting at q_0.

Step 3

The third step of the construction establishes the final states for *M*. Any state *R* in Q' that contains a final state of *N* is a final state in *M*.

Step 4

The fourth step determines the transition function δ' for machine *M*. For a state $R_1 \in Q'$ and an input *a*, $\delta'(R_1, a)$ returns $R_2 \in Q'$, where $R_2 = \bigcup_{r \in R_1} E(\delta(r, a))$, as shown in definition 2. In other words, we must construct the state R_2 that will be entered by the DFA *M* when *M* is in state R_1 and *a* is the next symbol on the input tape: state R_2 is the union of all states entered by *N* from each state $r \in R_1$ on input symbol *a*.

This is the complete construction for converting a NFA into a DFA. Although it can be difficult to read initially, once understood it is a fairly simple construction. Many students, however, have difficulty reading mathematical descriptions of an algorithm and may not be able to interpret what is truly being done. In the next section the process that the software developed for this thesis uses will be discussed. It presents a simple process

through which this algorithm may be more easily understood, along with a way to actively view the process taking place.

The Conversion Algorithm

The algorithm developed for converting an NFA *N* to a DFA *M* is presented here. It follows the construction described in the previous section in an obvious fashion, changing only where necessary for pedagogical or technical reasons.

The algorithm starts by assuming that the individual (singleton set) states of *N* are automatically states of *M*. The algorithm then creates the extra states needed by the DFA *M*, following Lemma 1 from the previous chapter. A null state will also be added representing the empty set of states. *M* will thus have $2^{|Q|}$ states, where *Q* is the set of states in *N*. All of these states may not be used, but following Lemma 1, all states are created; some will later be removed if they are not used. (The exponential number of states that are thus created and the limits of the applet size make it unwise to attempt to use the applet on an NFA that has more than four states in this conversion.)

The next step is to change the start state. The new start state q_0' is created by finding every state $q \in Q$ that can be reached from q_0 by following ε transitions. Each of these states unioned with q_0 represent the state q_0' that is the start state for *M*.

Once the start state is constructed, the final states must be built. According to the algorithm the final states of *M* are any states $R \in Q'$ where *R* contains an accept state $r \in Q$.

At this point the algorithm has reached the stage where the transitions of *M* must be determined. This is done in two stages. First, the applet removes ε transitions. In order to do this, new transitions must be added to replace the empty transitions. This is the first place where the construction implied by Lemma 1 and the algorithm presented here differ. It was felt by the author that the conversion would be better understood if the transition determination was done in stages. Therefore removing empty transitions and adding transitions that will replace the empty transitions is done first. After this there is a step to add and remove all other transitions as necessary.

Remember that at this point in the construction of DFA *M* from NFA *N*, *M* consists of the states of *N* (where these states of *M* are considered to be the singleton sets of $P(Q)$), the remaining non-singleton states in *P(Q)*, and the original transitions of *N* that only go between the singleton states in *M*. No new transitions involving the newly constructed non-singleton states of *M* have yet been built. Thus, to remove ε transitions in *M*, each singleton state $q \in Q'$ that has an empty transition leaving it will need to be found, along with that transition's singleton destination state $p \in Q'$. For each of these states *q*, any singleton states $r \in Q'$ that have transitions that enter *q* must also be found. If *p* has any empty transitions leaving it, all states that are connected by empty transitions from *p* will be added to a set *P* along with *p*. This process continues until all such states added to *P* that also have empty transitions leaving them must have their destination states added to set *P*, until no new states can be added. Once all of the states connected to *p* through empty transitions that originate at *P* have been found, new transitions will be added. Each added transition will go from each state *r*, defined above, to state $P \in Q'$.

The symbol for the newly added transition will be the same as the transition from *r* to *q*. The old transition from *r* to *q* will be removed once the new transition is added. Once all such new transitions have been added, no empty transitions will remain[3]. At this point in the construction of *M*, *M* is still nondeterministic, it has no ε transitions, and *L(M)=L(N)*.

The next step is the most involved; it determines all of the remaining transitions that will be part of *M*. Any state $q \in Q'$ that enters multiple states on the same input symbol, *a*, in *N*, must now have a new transition added to it that travels from *q* to the state $R \in Q'$ that corresponds to the multiple states that *N* enters on $\delta(q, a)$. Then the original nondeterministic transitions from *q* must be removed. The newly added states in Q' that correspond to multiple or empty states in *Q* must have appropriate transitions added to them as well. In accomplishing this, the algorithm computes $\delta(A, a) = B$ for each state $A \in Q'$ such that $B = \bigcup_{q \in A} \delta(q, a)$ (remember that there are no more ε transitions in *M* at this point). All transitions so computed for each non-singleton state in *M* will be added to *M*, and the old transitions that leave *A* on input symbol *a* will be removed.

The last step will remove unnecessary states. Each state $A \in Q'$ that is not the start state and has no transitions entering it from other states will be removed, along with all transitions that leave *A*.

[3] In the current build of the FSA applet, cycles of empty transitions are not handled. Therefore, the NFA to DFA conversion modes do not handle this occurrence.

THE NFA TO DFA CONVERSION APPLET

This conversion applet is a direct extension of Grinder's and Cogliati's FSA animator applets. There are new modes that have been added so that all of the old functionality is still accessible while allowing the new conversion functions to work. In particular, one new mode has been added that places the animator into the conversion state. This new mode has four secondary modes that allow different ways to view and interact with the conversion process. The modes are listed below, along with what buttons are added and briefly what actions are performed when these buttons are clicked.

- The first mode is the detail mode. This places the *Detail Demo* button in the top control panel. This button allows the user to view the conversion process at a very slow pace with every step described in great detail.
- The second mode is the demo mode. This places the *Slow Demo* and *Fast Demo* buttons in the control panel. It is primarily intended for users who are familiar with the conversion algorithm, but who wish to review certain sections, and don't need all the detail that the detail mode provides.
- The third secondary mode is the active mode. This places a *Self-Exercise* button into the control panel. This mode allows the user to build the DFA that accepts the same language as the NFA given, using the same steps that the demo and detail modes use.
- The fourth mode is default mode. It places the Slow Demo, Fast Demo, and Self-Exercise modes into the control panel.

Detail Demo Conversion

The Detail Demo moves step by step through Sipser's NFA to DFA conversion algorithm. At each step, text is displayed that tells the user what has just occurred and what is going to happen next in the algorithm. Precise and detailed explanations about the steps in the algorithm are given.

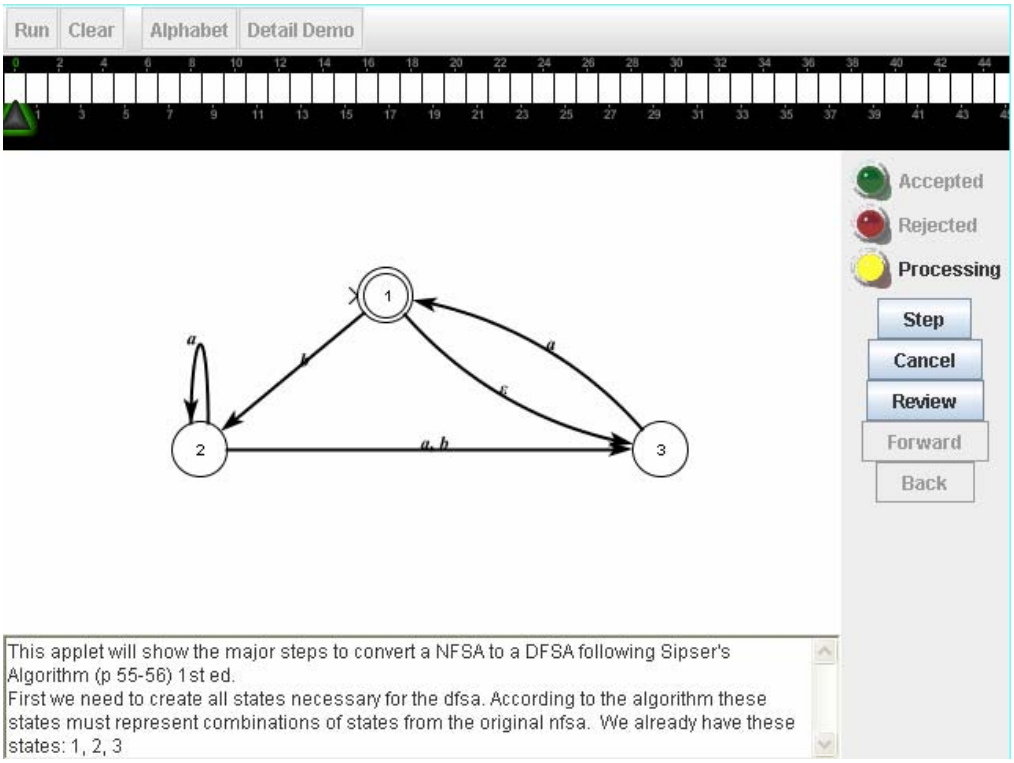

Figure 12. The start of the Detail Demo

The state of the applet right after the Detail Demo button has been clicked can be seen in Figure 12. All of the modes start by labeling each of the states with a number for ease in reference. Each time the Step button is pressed a new state will be created until all of the states exist. Figure 13 below shows the states that have been created after the Step button has been pressed twice. Pressing this button three more times will result in

all eight states (corresponding to the power set of states 1, 2, and 3) to be displayed, as shown in Figure 14.

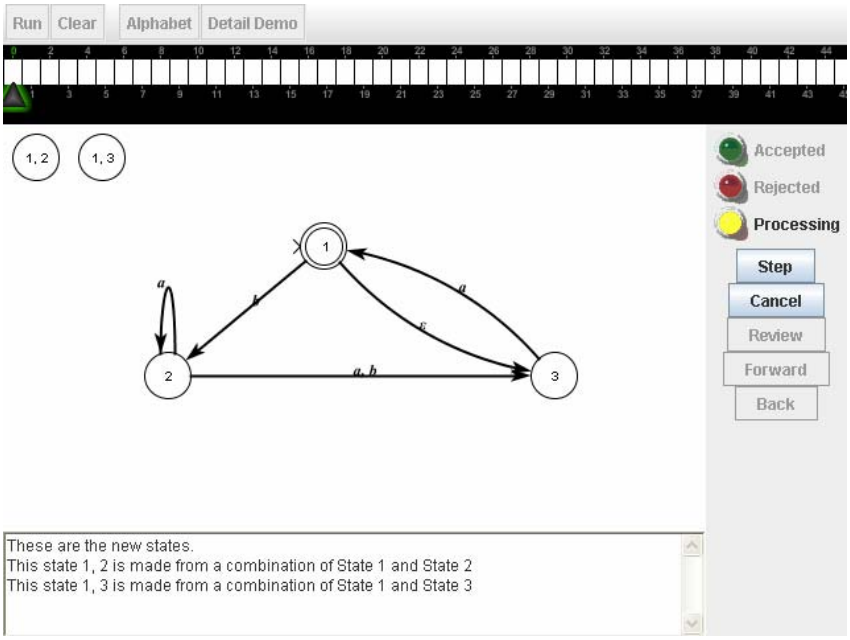

Figure 13. After two of five new states have been added

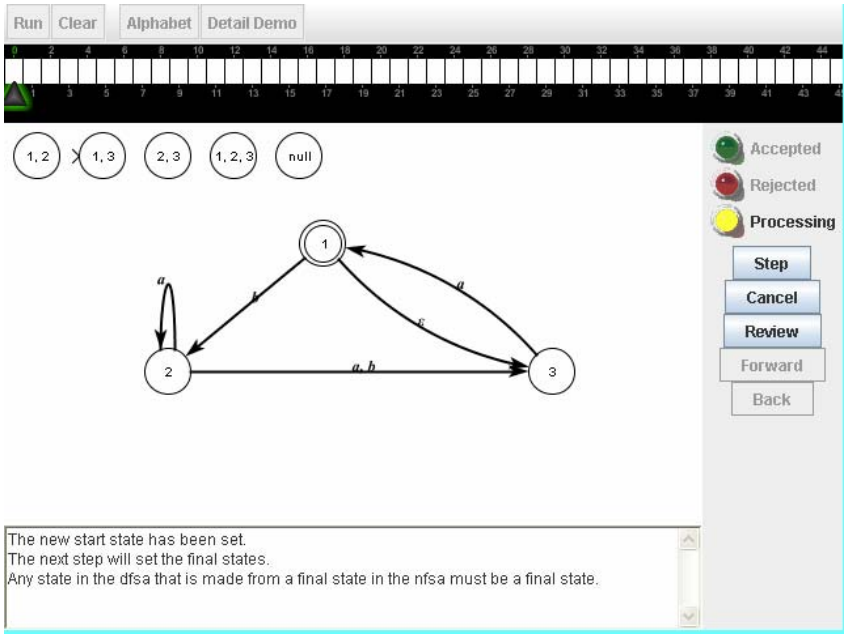

Figure 14. The start state has been set

The next step is to change the start state. In this example state (1) is the start state. Since there is an ε transition going to state (3) from state (1), state (1, 3) will be made the new start state. This can be seen in Figure 14.

Once the start state is set, the final states must be set. In this case, since state (1) is a final state in the NFA, states (1), (1, 2), (1, 3), and (1, 2, 3) are final states in the DFA. This can be seen in Figure 15.

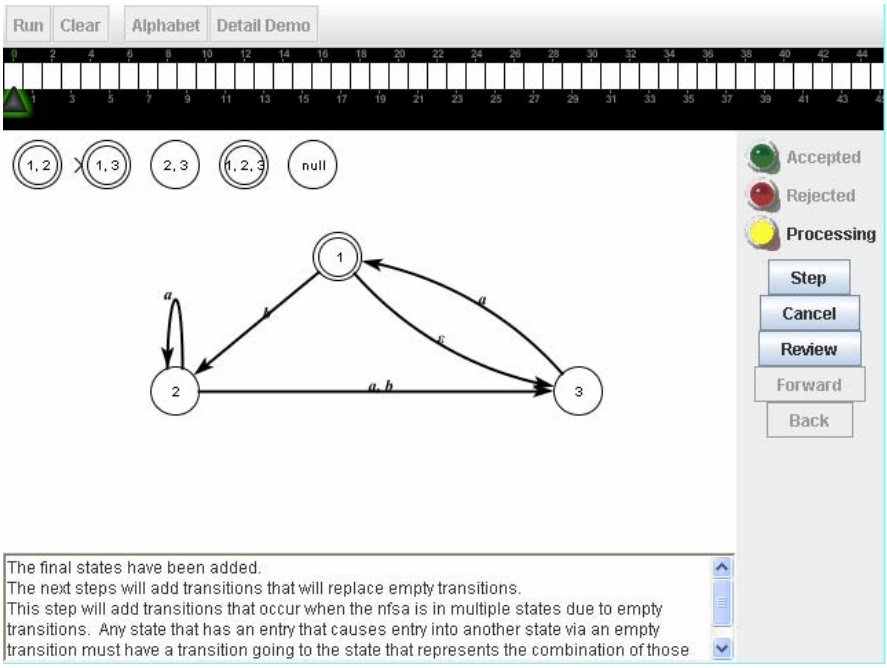

Figure 15. The final states have been set

The next step that the applet does is to remove ε transitions. In order to do this, new transitions must be added to replace the empty transitions. In our example there is only one empty transition: from state (1) to state (3). In order to remove this transition, every state that has a transition that enters state (1) must be looked at. State (3) is the only state that enters state (1), so a new transition must be made that starts in state (3) and ends at state (1, 3) with the transition symbol *a*. This is because every time state (1) is

entered, state (3) is entered as well via the ε transition, so in order to make this deterministic we redirect state (1)'s input to state (1, 3). At this point the transition from (3) to (1) will be removed. The empty transition can now also be removed. Figure 16 demonstrates this process.

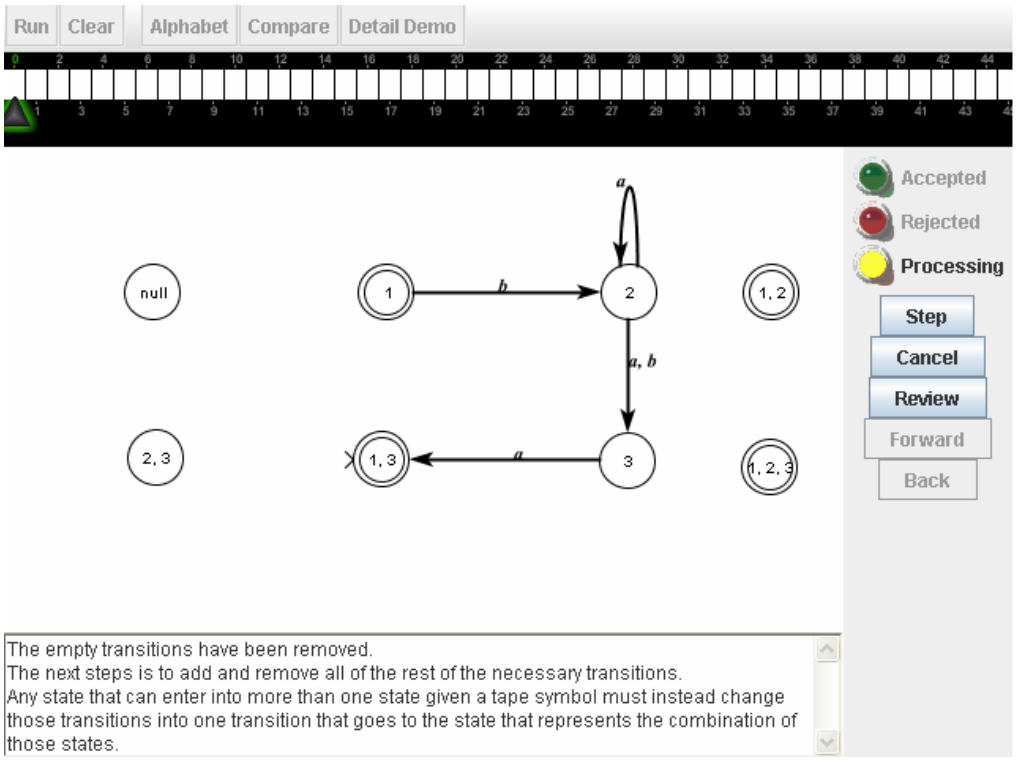

Figure 16. Empty transitions have been removed and transitions that replace them have been added

In this part of the algorithm, each state is selected and there will be a detailed description of what has just occurred for each step. Any transition that is added or removed will have its beginning and ending states reported. Also, if a transition was modified by having its symbol set changed, this will be reported. For an example of this refer to Figure 17.

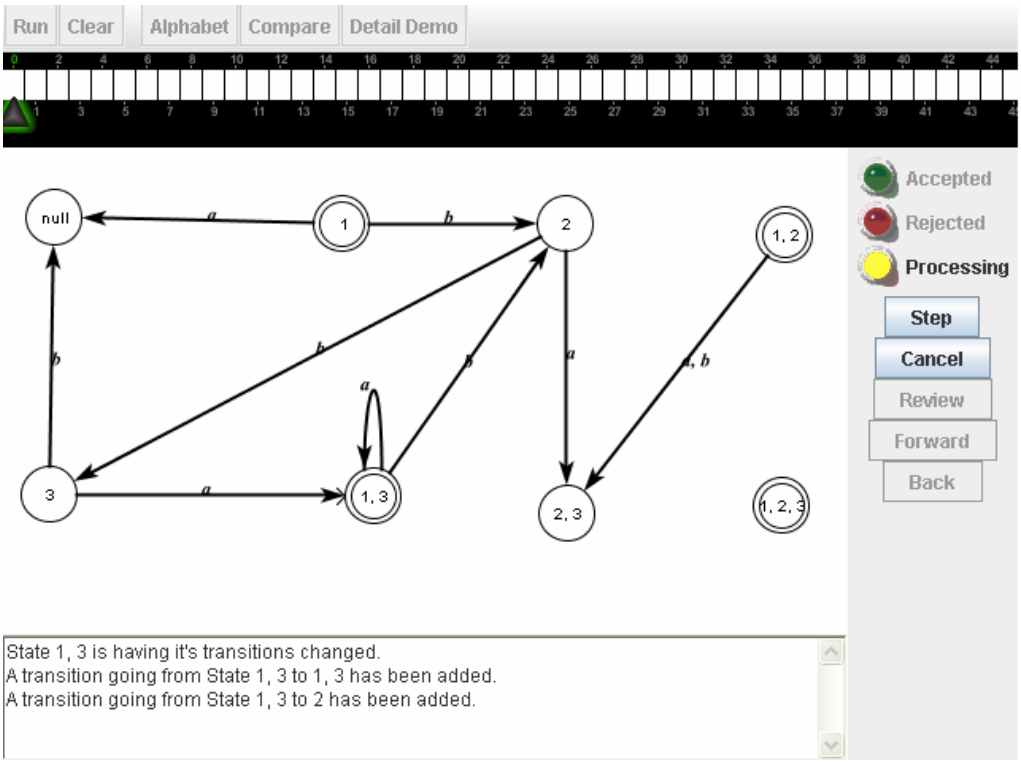

Figure 17. Some of the new transitions have been added, and all transitions that were replaced by new ones have been deleted

The final step of the *Detail Demo* is to remove unnecessary states. Any state except the start state that does not have a transition that enters it from another state is removed as it can never be entered. Once this is completed the green Accepted light is activated, and the animation is finished and enters the standard FSA mode, where the tape can be edited and a user can watch the FSA process the input. This allows students to see that this DFA is in fact equivalent to the given NFA.

Demo Conversion

The demo conversion adds a Slow Demo button and a Fast Demo button. The slow demo is identical to the detail demo except for the level of detail of the text

description that is displayed. The difference between them is that the slow demo does not describe how the conversion is going to be accomplished. Instead the applet tells the user what step will be performed next without telling the user what is actually occurring. It is assumed that the slow demo will be used by students who understand the algorithm, but are not yet completely comfortable with the individual steps. The difference between the two can be seen by comparing Figure 15 with Figure 18.

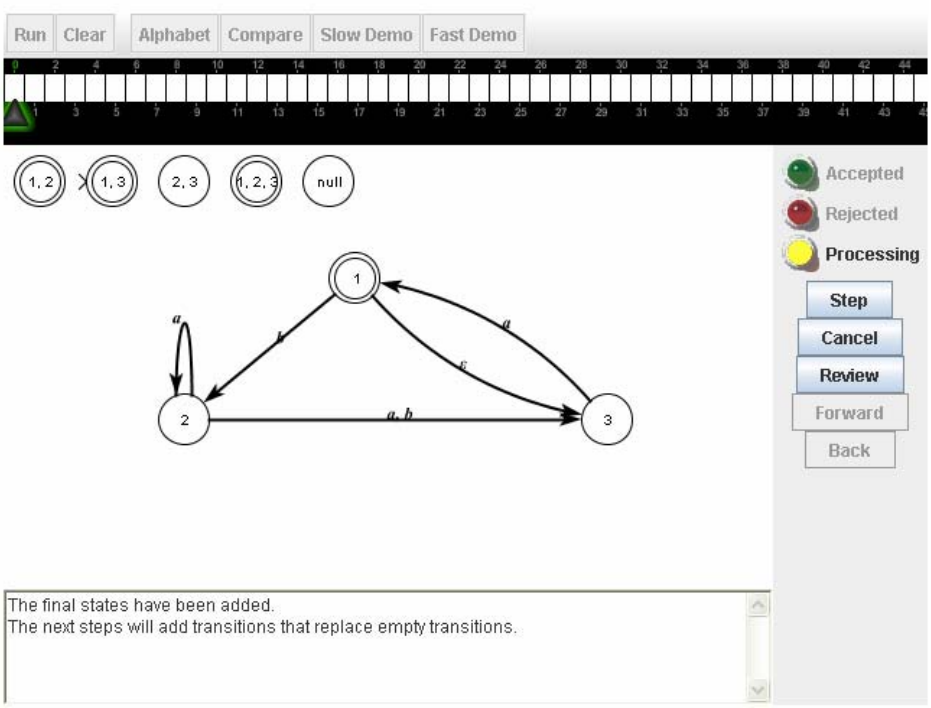

Figure 18. Slow Demo after the final state has been set

The fast demo takes a different approach than the other two demos. Rather than show each individual step of the algorithm, it only shows the major steps. The first step will therefore create all the extra states that are needed at once rather than show each individual state being created. The next step sets the new start state. After this the final states are set. It will then add any transitions that replace empty transitions. The fast

demo does not create each individual transition but instead creates all of them in one step. Likewise all empty transitions are removed in the next step. After this step all new transitions are created, transitions that need to be modified are modified, and transitions that have been replaced are deleted. The final step removes states with no entry point. This demo is designed for students who are very comfortable with the algorithm, but wish to review it.

Active Conversion

The active conversion mode allows the user to test what they have learned about the conversion. This mode of the applet guides the user through the process of converting a given NFA into an equivalent DFA. Each major step of the algorithm is preceded by detailed text describing to the user what is expected from them. After they have completed the step, the user can press the step button. At this point their answer is checked against the solution that was included in the background by the instructor when the exercise was created. If the user is wrong on any point, the machine is reset to the last point the user correctly finished and the user is told what mistakes were made.

Figure 19 shows the machine right after the Self-Exercise button has been pressed. The first step asks the user to create the new states that will exist in the DFA. There are instructions in the text bar on how to create the new states. The user must create new states and label them correctly to continue. An early version of this applet's full implementation in a web page for the hypertextbook includes more detailed instructions on how to create these states. In this example, in order to make the $2^{|Q|}$

states, the states (1, 2), (1, 3), (2, 3), (1, 2, 3), and (null) must be added to this FSA. Once the user has finished adding the new states they can check their answer against the solution by pressing the step button. Figure 20 shows what happens when the user includes invalid new states. If the user incorrectly included states that do not exist in the solution, the user is told which states are incorrect. If the user did not create a correct state however, the user is only told how many states are in the solution that they do not have in their answer. This makes it so that the user has some direction on how to complete this section without being given the entire answer. Incorrect transitions are handled in a similar manner.

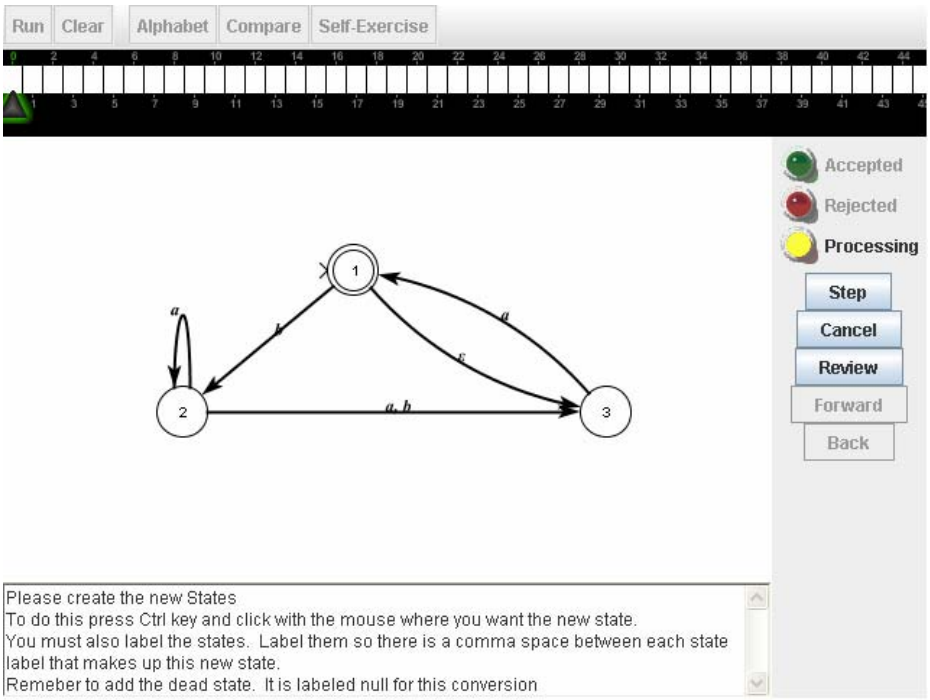

Figure 19. FSA right after the Self-Exercise button has been clicked

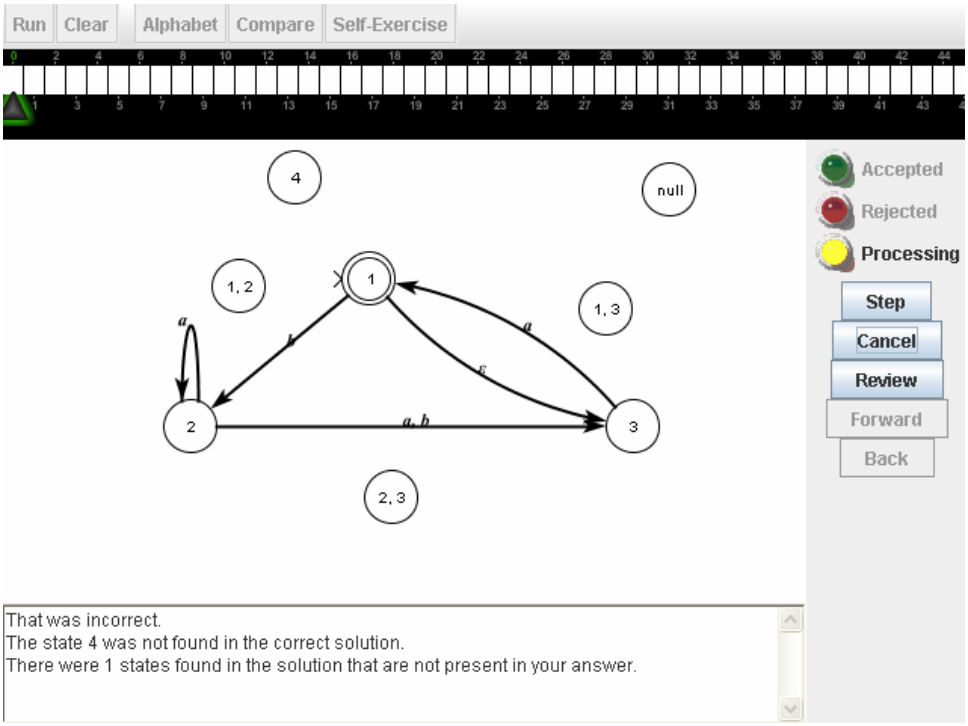

Figure 20. An example of an incorrect answer

Once the correct states have been added, the applet will ask the user to select a new start state. During the self-exercise mode it is assumed that the user is familiar with the algorithm, so the steps that must be performed are given, but no instructions for how the user is to accomplish a step are given. Figure 21 shows the applet asking the user to select the new start state. The new start state that will be selected is state (1, 3), as the current start state (1) has an empty transition going to state (3). If this state is not set to the start state, the applet will tell the user that the selected state is not the start state and ask them to set the correct start state.

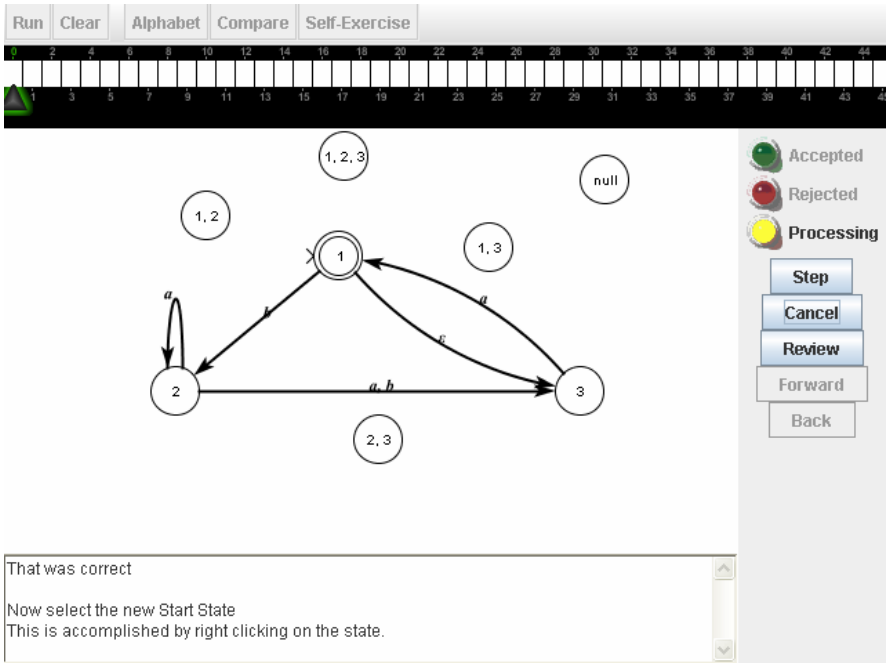

Figure 21. Applet asking the user to supply the new start state

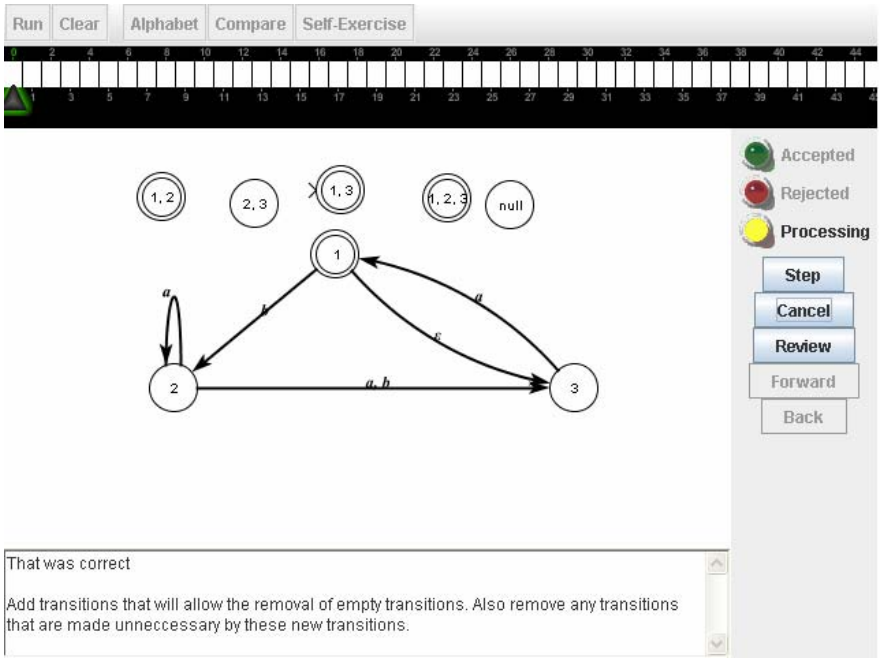

Figure 22. FSA after the user has set the new final states

The next step that is given to the user is to select the new final states. Figure 22 gives an example of this. In this example the user must select all states that have a (1) in their state label.

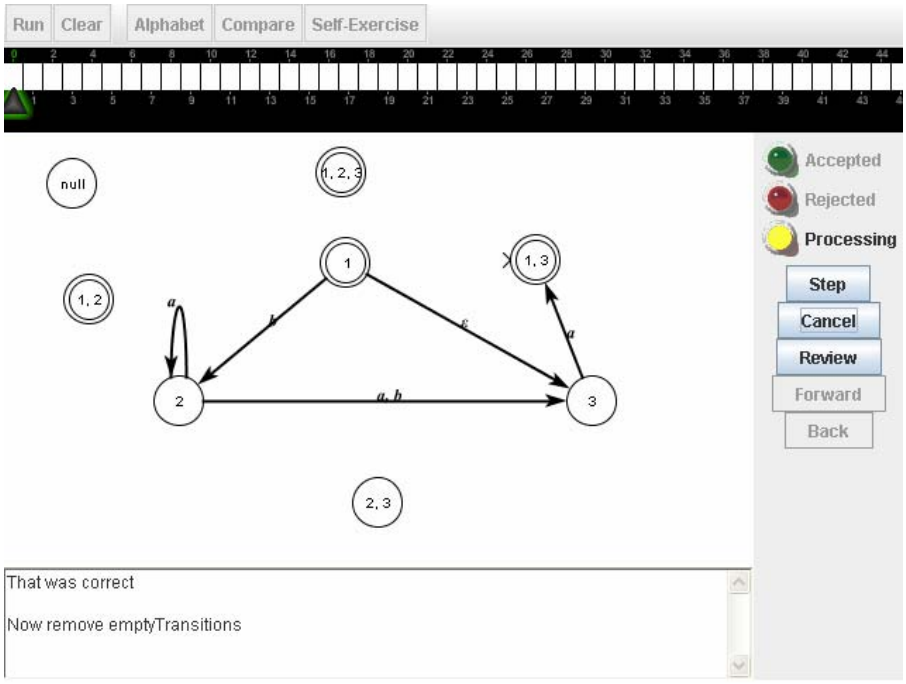

Figure 23. Transitions have been changed to allow the removal of empty transitions

After the final states have been selected, the user will be asked to add transitions that will replace the empty (ε) transitions and to remove the empty transitions. In this example there is only one empty transition, from state (1) to state (3). Since every time state (1) is entered, state (3) is also entered, the user must look at all of state (1)'s entry points. Therefore a new transition must be added from state (3) to state (1, 3) with a symbol *a* since the only entering transition to state (1) is the transition from state (3) to state (1) with a symbol *a*. The transition from state (3) to state (1) will also need to be removed. This can be seen in Figure 23. After this the user will be asked to remove the

empty transitions, in this example the transition from state (1) to state (3). This is shown in Figure 24.

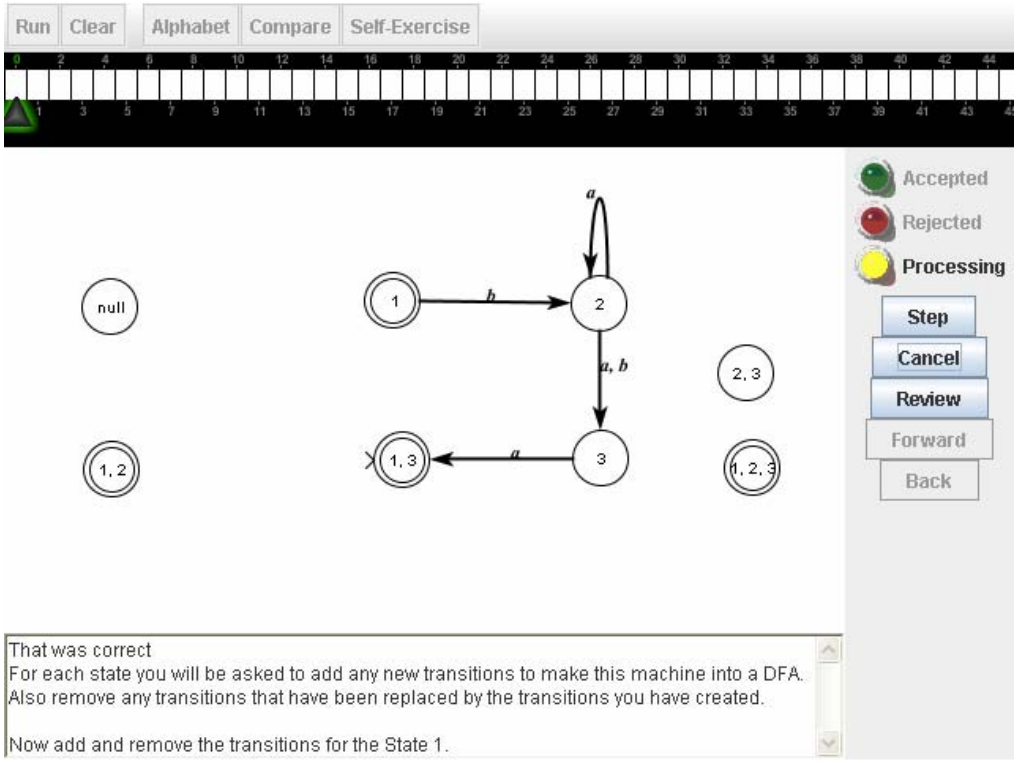

Figure 24. The empty transitions have been removed

Once this is done, the user will be asked to modify the transitions for the DFA. The applet will select a state, and the user will then be asked to create, delete, and modify all of the transitions leaving that state. As can be seen in Figure 25, state (1, 2) was selected and the correct transitions were added, and now the user is asked to complete the transitions for next selected state: (1, 3). By following the construction that is described by Sipser [7], the user needed to create two transitions. The first was from state (1, 2) to (2, 3) with symbol *a*; this was done because state (1) on *a* goes to null and state (2) goes to (2, 3) and the union of those two states is (2, 3). The second transition was from state

(1, 2) to (2, 3) with symbol b; this was because state (1) goes to state (2) and state (2) goes to state (3) on b.

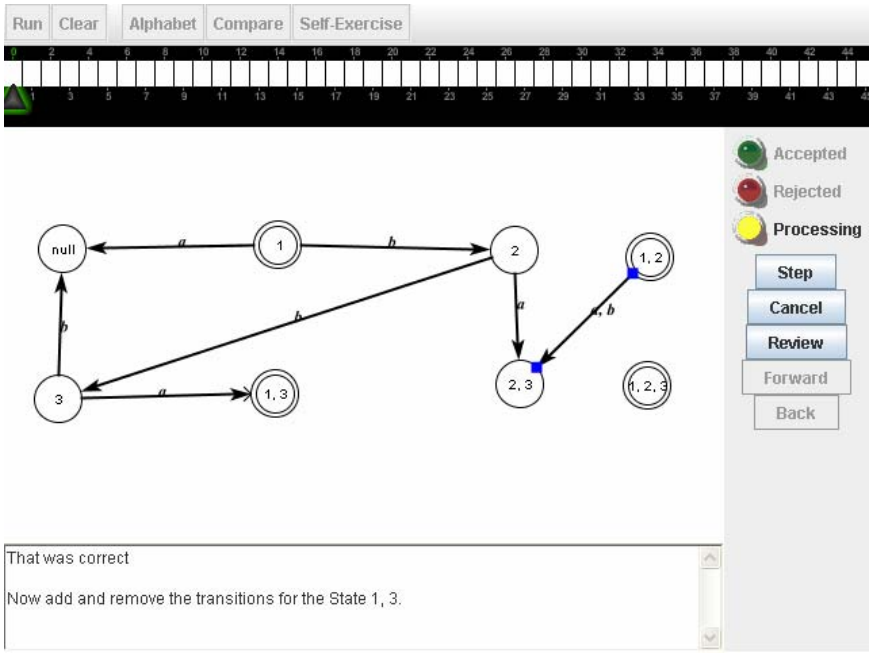

Figure 25. New transitions have been added to state (1, 2)

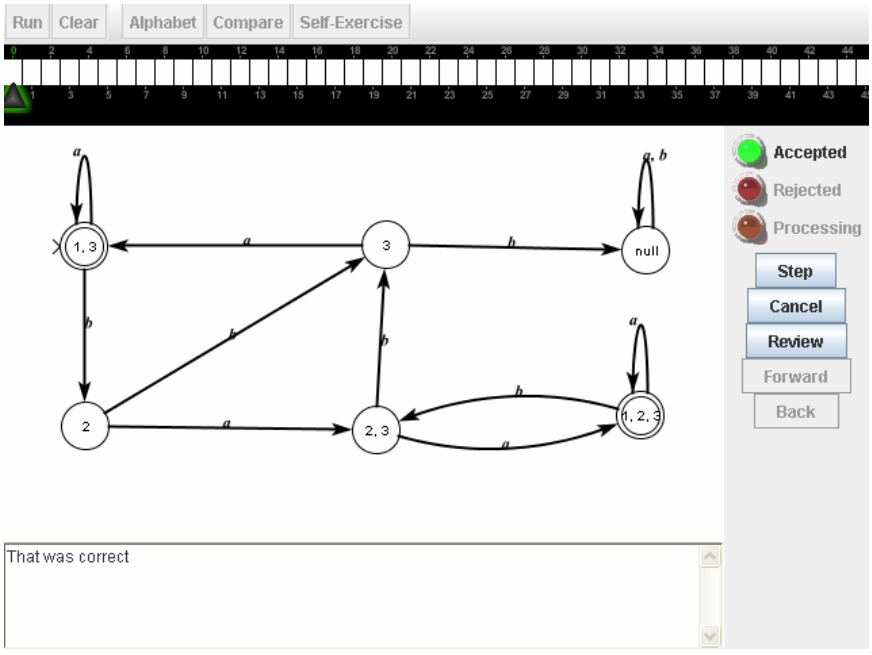

Figure 26. The FSA after the unnecessary steps have been removed.

The last step in the Self-Exercise is to remove unnecessary states. The applet asks the user to remove any states (other than the start state) that have no entry points. In this example states (1) and (1, 2) will be removed as can be seen in Figure 26. The green accepted light is lit at this point, indicating that the conversion has been completed successfully.

Default Mode

The default mode allows the user to click the Slow Demo, Fast Demo, or Self-Exercise buttons. It is there so that students can watch the demo and then work the exercise themselves. This is to help them to gain a better understanding of the algorithm and how it is specifically used in this applet.

Conclusions

This applet has three demo modes and one exercise mode in order to fully allow the student to understand the process of converting an NFA into a DFA. It is the belief of the author that these modes, in combination with all of the features of this applet, will help students to better understand this conversion.

Many students feel that the ability to use their own test data in visualization algorithms would be helpful [1]. In the software developed for this thesis, the user may input an NFSA of the user's own making.

Also, in all of these modes, there is a review button available that the user may press that allows them to review the process. This allows them to step back through the

algorithm and recheck what has happened and then step forward until they reach the point where they originally pushed the review button. At this point the algorithm continues as normal.

The text box at the bottom provides constant feedback, alerting the user to what has happened and what is going to happen next. Also in the self-exercise mode, if the user provides an incorrect answer, they are told what it is that they have added that is wrong and what they need to add to the DFA that is needed for that step to be correct.

EVALUATION

Much debate has been given to the effectiveness of algorithm animation. Some studies have shown that the animations do not help students learn any better than they do without them. However, in *Rethinking the Evaluation Algorithm Animations as Learning Aids: an Observational Study* by Colleen Kehoe, John Stasko, and Ashley Taylor, an evaluation of how to improve the use of algorithm animations was made. Their experiments led to three hypotheses about animations, which were shown to be accurate [1].

> Hypothesis 1: The pedagogical value of algorithm animations will be more apparent in open, interactive learning situations (such as a homework exercise) than in closed exam-style situations.
>
> Hypothesis 2: Even if animations do not contribute to the fundamental understanding of an algorithm, they do enhance pedagogy by making an algorithm more accessible and less intimidating, thus enhancing motivation. In that regard, they facilitate learning.
>
> Hypothesis 3: Algorithm animation can best facilitate learning of the procedural operations of algorithms [1].

In building this applet, these hypotheses were kept in mind. The foremost thought in creating the theory hypertextbook is to make an educational resource that will improve the capacity for students to learn the fundamentals of computer science theory. In pursuing this goal, there were choices made to help implement this. In addition to having demonstrations that could be shown in the classroom, the applets are easily placed into web pages and can be used by students out of the classroom, independent of hardware and software used. Only an Internet connection or a CD-ROM would be required as the

hypertextbook will be released on CD. So both demonstrations and exercises can be assigned as homework, facilitating students to work outside of the classroom on their own time.

The Webworks project has adopted a standard, simple interface for the automaton applets that will allow students to approach each new applet with a familiarity that will ease use and make the use of this software as painless and un-intimidating as possible. This is especially important as there will be quite a few algorithms that require animation in the project and it would be confusing and self-defeating for there not to be a standard interface, as no one would wish to relearn a new interface every time a new automaton related algorithm was presented to them. This allows each algorithm to be accessible, and it is our hope that this will help to facilitate a better understanding in the students.

One of the main goals of the Webworks project is to create applets that animate various computer science related algorithms in a step by step manner. This means that our project has more chance of succeeding as a pedagogical tool since our purpose is not to try to animate some parts of the science that are less responsive academically to animation, such as computational complexity, coding an algorithm, and viewing an algorithm's relationship to other algorithms [1].

In order to test the effectiveness of this applet, web pages were created with examples and exercises for students to use. A questionnaire was also developed that asks the users pointed questions about the use of the applet and the perceived effect of the applet on their learning and understanding. There was also some thought towards

creating a control group of students that didn't use the applet and a test group that did use the applet and comparing the two groups' grades at the end of the unit. Unfortunately, during the time this thesis was being completed, the course in which the applet would be used was not taught. Thus, the testing of the efficacy of this applet remains to be done.

FUTURE WORK

This thesis is not the end of the work that is needed for the Webworks project. There are many more theory concepts in the works to be animated in addition to enhancements to previously constructed algorithm applets.

There is a push to enforce a single GUI format for all of the applets, which is currently lacking in many of the older applets. The goal is to establish the format set out by the FSA animator in all applets, past, present, and future. Also at this point, except for the NFA to DFA conversion, the review button and text box are not utilized by the conversions. Hopefully in the future all applets in the Webworks project will have this functionality added.

The applet for this thesis also has some work that could be added to increase its effectiveness. Currently when one chooses to review (via the *Review* button) the NFA to DFA conversion, one cannot do this at any point due to resource conflicts introduced by the way the applet was threaded. It would be nice to find a way to work around these limitations and allow the user to review the conversion at any point they wish. The ability for the FSA to handle empty transition cycles is another addition that the author would like to see added to this project. One last thing that should be added is a mode that allows the user to build a DFA from a given NFA without going through the algorithm step by step. The user would build a DFA and press the compare button and then be told whether their machine is correct or not.

In addition to fixes to current parts of the Webworks project there are quite a few other theoretical concepts to be animated. The next addition to the project will be a

minimal conversion DFA applet. This applet will animate the algorithm that converts a DFA into an equivalent minimal DFA. Also in the works are applets to convert regular expressions into NFAs and DFAs into regular expressions. Once these are finished, applets that animate Turing Machines and many of the algorithms that accompany Turing Machines will be added. In all likelihood, any dynamic concept or algorithm that is part of the theory of computing will eventually have an applet animating it, so that students may better understand it.

REFERENCES CITED

1 Colleen Kehoe, John Stasko and Ashley Taylor. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *Int. J. Human-Computer Studies* (2001) 54:265-284. http://www.idealibrary.com

2 Joshua J. Cogliati, Frances W. Goosey, Michael T. Grinder, Bradley A. Pascoe, and Rockford J. Ross. Realizing the Promise of Visualization in the Theory of Computing. submitted to Journal of Educational Resources in computing, July 2004.

3 Rockford J. Ross. Hypertextbooks: Animated, active learning, comprehensive teaching and learning resources for the web. Software Visualization, pages 269-283, 2002.

4 Joshua J. Cogliati. Visualizing the Pumping Lemma for Regular Languages. MS Thesis, Montana State University, Computer Science Department, July 2004.

5 Michael Thomas Grinder. Active Learning Animations for the Theory of Computation. PhD disertation, Montana State University, Computer Science Department, December 2002.

6 Susan Roger. *Java Formal Language Automata Package(JFLAP)*. February 2006. http://www.jflap.org

7 Michael Sipser, Introduction to the Theory of Computation, chapter 1 pages 31-83. PWS Publishing Company, 1997.

8 Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13:259-290, 2002.

9 Susan Rodger and Thomas W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, 2006.

10 Peter Linz. *An Introduction to Automata and Formal Languates, 4ed*. Jones and Bartlett Publishers, 2006.