ACTIVE LEARNING ANIMATIONS FOR THE THEORY OF COMPUTATION

by

Michael Thomas Grinder

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

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 2002

APPROVAL

of a dissertation submitted by

Michael Thomas Grinder

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

Rockford J. Ross	(Signature)	Date

Approved for the Department of Computer Science

Rockford J. Ross	(Signature)	Date

Approved for the College of Graduate Studies

Bruce McLeod	(Signature)	Date

STATEMENT OF PERMISSION TO USE

ACKNOWLEDGMENTS

Glory to God in all things.

Many people need to be thanked for their assistance in the process that brought me to this point:

- Rocky Ross, my dissertation advisor, for his advice, patience, and careful proofreading
- Jeff Adams, Fred Cady, Bob Cimikowski, Gary Harkin, and John Paxton, for serving on my dissertation committee
- Ann Defrance, Bob Cimikowski, and the TAs and students in CS 221 and 223 at Montana State University during Spring Semester 2002, for being my "guinea pigs"
- my parents, Edwin and Linda Grinder, and the rest of my family
- Lou Glassy, Chris Boroni, and Zuzana Gedeon, the other three "Horsemen"
- Marty Hamilton, for important advice about statistics
- Terri Dore and Jeannette Radcliffe

I do not have room to thank everyone else who helped me along the way. Please forgive my oversight.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES - CONTINUED

ABSTRACT

This dissertation presents the author's design, implementation, and evaluation of active learning animation software for teaching the theory of computation. The software builds on techniques used in traditional textbooks, in which concepts are illustrated with static diagrams. Developers of animation software have worked to make these traditional static diagrams come to life using motion, color, and sound (a process commonly referred to as animation), allowing students to manipulate and explore concepts in a fully interactive graphical environment. However, the mere vivification of static diagrams exploits only a small amount of the potential that modern personal computing environments provide. It is possible for animation software to make further use of this potential by providing learning activities that would be impractical or even impossible to duplicate using traditional methods.

To support this claim, the author developed software for simulating finite state automata (FSAs), the FSA Simulator. The FSA Simulator is designed for a variety of uses from in-class demonstrations to integration into a comprehensive "hypertextbook." Although many others have developed similar software, 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.

The effect of the FSA Simulator on students' learning was evaluated in preliminary experiments in undergraduate computer science laboratories at Montana State University. 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.

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 computation. The author 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.

ACTIVE LEARNING ANIMATIONS FOR THE THEORY OF COMPUTATION

Michael Thomas Grinder

Advisor: Rockford J. Ross, Ph.D.

Montana State University
2002

Abstract

This dissertation presents the author's design, implementation, and evaluation of active learning animation software for teaching the theory of computation. The software builds on techniques used in traditional textbooks, in which concepts are illustrated with static diagrams. Developers of animation software have worked to make these traditional static diagrams come to life using motion, color, and sound (a process commonly referred to as animation), allowing students to manipulate and explore concepts in a fully interactive graphical environment. However, the mere vivification of static diagrams exploits only a small amount of the potential that modern personal computing environments provide. It is possible for animation software to make further use of this potential by providing learning activities that would be impractical or even impossible to duplicate using traditional methods.

To support this claim, the author developed software for simulating finite state automata (FSAs), the FSA Simulator. The FSA Simulator is designed for a variety of uses from in-class demonstrations to integration into a comprehensive "hypertextbook." Although many others have developed similar software, 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.

The effect of the FSA Simulator on students' learning was evaluated in preliminary experiments in undergraduate computer science laboratories at Montana State University. 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.

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 computation. The author 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.

CHAPTER 1

INTRODUCTION

This dissertation presents the author's design, implementation, and evaluation of active learning animation software for teaching the theory of computation. The software builds on techniques used in traditional textbooks, in which concepts are illustrated with static diagrams. Developers of animation software have worked to make these traditional static diagrams come to life using motion, color, and sound (a process commonly referred to as *animation*), allowing students to manipulate and explore concepts in a fully interactive graphical environment. However, the mere vivification of static diagrams exploits only a small amount of the potential that modern personal computing environments provide. It is possible for animation software to make further use of this potential by providing learning activities that would be impractical or even impossible to duplicate using traditional methods.

To support this claim, the author developed software for simulating finite state automata (FSAs), the FSA Simulator. The FSA Simulator is designed for a variety of uses from in-class demonstrations to integration into a comprehensive hypertextbook [15, 35]. Although many others have developed similar software (for example, [74, 5, 67, 36, 70]), 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.

The effect of the FSA Simulator on students' learning was evaluated in preliminary experiments in undergraduate computer science laboratories at Montana State University. 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.

Enhancing Computer-Aided Instruction with Active Learning

As personal computers have improved and become ubiquitous over the years, their potential for enhancing education has increased dramatically. From their earliest days, personal computers have been integrated into educational settings in one form or another. In the beginning, they were most often used in elementary school classrooms to aid rote memorization tasks such as learning basic arithmetic and spelling. As their capabilities increased, personal computers equipped with CD-ROM drives began to be used as reference tools. Electronic books and educational multimedia presentations were provided for online use. Many science labs were also equipped with computerized measurement devices. In recent years, especially with the arrival of the Internet, computers have routinely been used by students for doing research and collaborating with others through e-mail and other forms of electronic communication.

So far, most applications of computer technology within education have largely been passive. For example, electronic encyclopedias 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. Passive learning programs tap into only a small portion of the promise for enhanced learning afforded by personal computers. It is well known that students learn better when they are engaged in active, rather than passive, learning [11, 52]. Thus, it is imperative that interactive learning software be developed in order to exploit the full potential of computer-enhanced learning.

Active Learning Software

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. The most elementary means of providing a student with active learning opportunities is to allow the student to submit differing inputs for the system to use during an animation of a concept. For example, a finite state automaton animator might allow a student to input a string for the automaton and then watch as the automaton processes it. While the software is animating the concept (e.g., a finite state automaton) based on the student's input, other opportunities for interaction are also provided: The student can often control the speed and appearance of the animation or even choose among several different views of the concept being animated.

More sophisticated active learning features include such things as allowing the learner to change the model being animated on the fly, providing facilities to a learner to allow construction of entirely new models for animation, and providing feedback to students to guide them toward successful completion of exercises. Examples of these features are provided as part of this dissertation.

Incorporating active learning into teaching and learning resources has numerous benefits. There is some evidence that active learning animation software, when used properly, can improve student learning (see page 35). Even if animation software does not directly improve learning, its use often appears to markedly increase students' enthusiasm for a topic, indirectly improving their performance in a course.

There is also evidence that active learning animation software can also benefit those who already understand a concept. The process of creating or watching a visualization of a subject may trigger new insights. For example, an animation of sorting algorithms inspired a worst-case analysis of Shell-sort [24] and visualization software for teaching logic has caused researchers to reconsider the nature of reasoning [6].

Despite these benefits and the availability of many quality educational software packages, active learning animation software is not widely used. One can speculate about the reasons for this. One is that animation software for education was historically written for a single, specific computing environment, which precluded its use on other systems. Currently, with the popularity of the cross-platform Java programming language and the dominance of the Windows operating system, this is not as much of a problem as it was in the past.

A second possible reason for the lack of use of animation software is that it often requires complex installation and configuration. Since most animation packages usually only deal with one particular topic, an instructor wanting to use animation software in a course, must find, install, and integrate each animation system of interest into the course. Most instructors do not have the time to do this. Thus, helpful software often remains unused.

To alleviate these problems, an integrated, cross-platform learning environment that incorporates animation software is needed. One effort to develop such an environment is underway in the Webworks Laboratory of the Computer Science

Department at Montana State University. The eventual objective of the Webworks Laboratory is the construction of a framework that supports the development of hypertextbooks for the World Wide Web. A *hypertextbook* combines hyper-linked text, images, audio, and video, with active learning Java applets (the animations) to provide a dynamic, web-based teaching and learning resource that greatly extends the capabilities of traditional textbooks. Since hypertextbooks are based on cross-platform web technology, they can be used on any platform that has a Java-enabled web browser. Hypertextbooks can be distributed either over the Internet through a web site or on some form of electronic media such as a CD-ROM, requiring little or no installation or configuration. Since the animation software is already integrated into the text, no extra effort by the instructor is required to use animations in a course. Thus, hypertextbooks address most of the issues discussed above that have limited the adoption of active learning educational software in computer science courses. The work presented in this dissertation represents an important step towards making hypertextbooks a reality.

Conventional Computer Science Education

Conventional instructional methods have many drawbacks when used to teach the numerous dynamic processes found in computer science. For example, topics such as algorithms, data structures, and models of computation require descriptions of constantly changing information. Presentation of these topics can be accomplished, in part, by an instructor at a whiteboard using diagrams and illustrations, but it is still difficult using such means to clearly convey these ideas to students.

Dedicated instructors often hone their lecturing skills in order to improve their presentation of complex, dynamic topics. A teaching style that actively engages students through dialog and that incorporates whiteboard diagrams of the subject is known to be effective [11], but it requires much effort and many years of experience to develop. Even then, the best lecturers do not get through to many students. Although students appear to comprehend the topic of a dynamic lecture as it is being presented, they must struggle to recapture this dynamic information later from their notes. As memories fade, the notes (which are a static representation of a dynamic event) often become a source of frustration and misunderstandings rather than a helpful study aid. Exacerbating the problems inherent in traditional teaching methods are large class sizes and heavy class loads that often prevent instructors from giving sufficient and timely feedback to their students on assignments and exams, further impeding the learning process.

Textbooks, being entirely static, have even more limitations in the presentation of dynamic concepts. Unlike teachers, books cannot be queried for additional information or alternative explanations. The clearest, most engaging texts often fail to successfully inform many students, even after repeated readings.

Clearly, conventional teaching methods are often not an efficient way to teach the inherently dynamic topics that are ubiquitous in computer science. Web-based, active learning resources offer powerful new ways of presenting information to augment traditional teaching and learning methods.

Unconventional Computer Science Education

Interactive animation software provides one possible solution to the limitations of traditional teaching and learning resources. Such 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. When using animation software, students are not required to rely solely on their memories and cryptic handwritten notes to review material taught in class.

Additionally, rather than being restricted to the limited set of examples provided by an instructor in a classroom, 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. Unfortunately, the learning opportunities provided by animation software are not usually a sufficient incentive to provoke most students to investigate beyond what they need to know for an assignment or the next exam. True active learning software needs to entice students into becoming actively involved with the topic being animated. 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.

The research discussed in this dissertation is an initial step toward providing such an “unconventional” active learning environment for the theory of computation. Much more research and development needs to be done to provide a complete set of resources for teaching this topic, but we are well on the way towards our goal.

CHAPTER 2

LITERATURE REVIEW

Introduction

Animation software for computer science education can be divided into three general categories: *program animators*, *algorithm animators*, and *concept animators* [16]. *Program animators* allow the user to step line by line through the source code of a computer program as it executes. The user is provided with a view of variables and the program stack and can watch how each line of the program modifies the variable and stack values. *Algorithm animators* provide dynamic, graphical representations of the execution of an algorithm operating on a particular data structure of interest. *Concept animators* illustrate higher-level concepts in computer science, such as the execution of a theoretical model of computation. The dividing lines between these categories are somewhat fuzzy, since some educational software packages fit within the definitions of more than one category, but the distinctions are useful for gaining a general overview of the field. Prominent examples of each type of animator will be discussed below.

Although much of the animation software discussed in this chapter is not directly related to teaching the theory of computation, it is still important to review it. The author's animation software for simulating finite state automata was influenced by many of the animation projects that preceded it. Also, in an integrated hypertextbook environment, the FSA Simulator will need to be integrated with other types of animation software to provide a comprehensive view of the theory of computation to its users.

Another important aspect of the study of animation software is evaluation. While many educators intuitively feel that active learning animation software helps their students learn, little empirical evaluation of animations has been done. Past efforts at empirical evaluation of animation software for computer science education will be discussed as well.

Evolution of Animators

All three types of animation software discussed in the previous section have passed through the same general evolutionary process. This process was fueled by improvements in computer hardware and by technology trends within the computing community. Many of these software systems were initially created in the late 1980s for specific computing platforms with very simple graphical or textual interfaces. As the speed and graphics capabilities of personal computers and low-end workstations increased in the early 1990s, animation software began utilizing more sophisticated graphical interfaces. However, most of these programs remained targeted at specific platforms, such as the Apple Macintosh, IBM PC, or a specific brand of Unix workstation, thus restricting their use to a small subset of the educational community that used various of these platforms in their curricula.

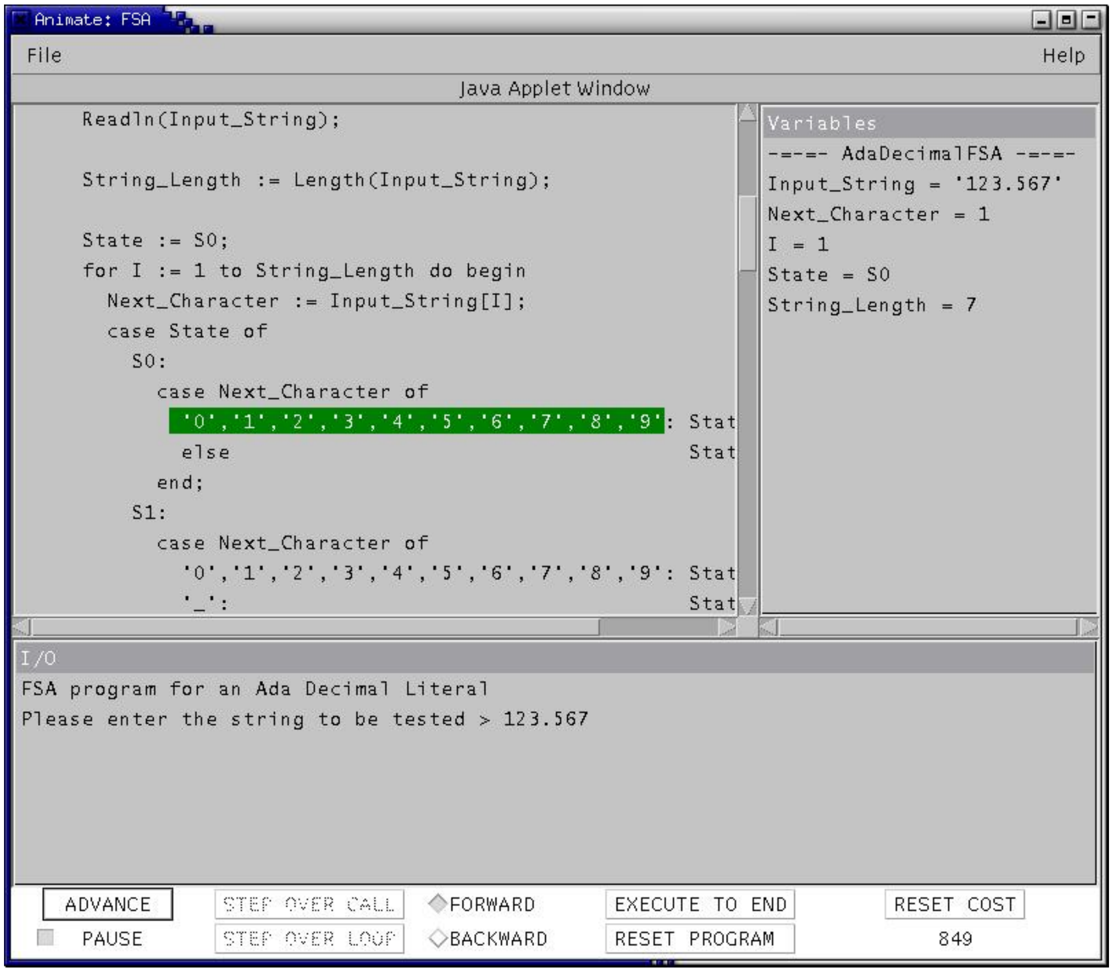
The release of the first version of the Java programming language in 1996 was a watershed event for the developers of educational animation software. The widespread adoption of Java by much of the computer industry made Java (and the Java virtual machine) an ideal platform for visualization software. The various animation software packages could be targeted to the Java virtual machine and then be run without recompilation on any computer with a Java virtual machine installed, no matter what operating system the computer was running. During this time, many of the visualization software packages for computer science education were ported to Java and many new projects were written from scratch in the language.

Program Animators

As stated previously, program animators allow the user to step through the source code of a computer program as it runs. They provide a view of the program stack and display the changing values of variables as the program executes. Although program animators are similar to debuggers used for software development, program animators are specially designed for educational purposes. Some of the unique features of program animators include the ability to reverse execution at any point, automatic display of variable values, and special stepping modes that encourage students to predict the behavior of a program.

Dynalab

A prominent example of a program animator is Dynalab, which was developed at Montana State University under the direction of Rockford J. Ross. The Dynalab project started with the design of a virtual machine, known as the E-Machine [64], that allows reverse execution. An emulator for the E-Machine was developed shortly thereafter [9]. Once the E-Machine emulator was available, development of C, Ada, and Pascal compilers began [32, 33, 65]. As the compilers were created, the software for animating source-level programs in the corresponding high-level languages (i.e. the program animators) were developed concurrently. One program animator was



```
    Readln(Input_String);

    String_Length := Length(Input_String);

    State := S0;
    for I := 1 to String_Length do begin
      Next_Character := Input_String[I];
      case State of
        S0:
          case Next_Character of
            '0','1','2','3','4','5','6','7','8','9': Stat
            else                                     Stat
          end;
        S1:
          case Next_Character of
            '0','1','2','3','4','5','6','7','8','9': Stat
            '_':                                     Stat
```

```
Variables
-=-=- AdaDecimalFSA -=-=-
Input_String = '123.567'
Next_Character = 1
I = 1
State = S0
String_Length = 7
```

```
I/O
FSA program for an Ada Decimal Literal
Please enter the string to be tested > 123.567
```

Figure 2.1: Java version of the Dynalab program animator

written for Unix [66] and another for Microsoft Windows [12]. Unfortunately, as is often the case with software developed by graduate students, the C and Ada compilers were never fully completed and have not been maintained.

The Pascal compiler was completed, however. Originally it covered only a subset of the Pascal language, but it has been continuously updated and maintained over the years and now supports nearly all of the Pascal standard [80].

After the advent of Java and web browsers, in order to escape the confinement of platform-dependence, the E-machine emulator and the program animator were ported from C to Java to take advantage of Java's cross-platform capabilities. At this point the Pascal compiler has not been ported to Java, so Pascal programs can be animated but not compiled from the Java version of Dynalab. A Java compiler (written in Java) for the E-machine is currently under development [13].

The Dynalab user interface is divided into four parts (see Figure 2.1). The *Source* pane in the upper left corner displays the source code of the program being animated. In the upper right corner, the *Variables* pane contains the current program stack with the values of the variables in each stack frame. Below these two panels is the *I/O* (Input/Output) pane which displays any data input by the user and any output from the program. Below the *I/O* pane is a panel containing widgets for controlling the program's execution and a counter which shows how many E-Machine instructions have been executed so far (for use in time complexity analysis of programs).

There are several possible modes of execution in Dynalab. The default mode is to run forward with pausing. In this mode, the animator stops executing at each block of code that will change the program stack or perform input or output. The piece of source code that is about to execute is highlighted in red to emphasize that the student should stop and think about what will happen next. When the student presses the *EXECUTE* button, the block of code is executed and the animator pauses again, this time with the code highlighted in green. When the student presses the *ADVANCE* button, the next piece of code that will be executed is highlighted in red. If pausing is disabled, the animator will skip the step with green highlighting and jump directly to the next block of code.

At any point during execution, the animator can be put in reverse. In *BACKWARD* mode, the highlighting color changes to black and an *UNEXECUTE* button appears, allowing the student to reverse the execution of the program an arbitrary amount. This reverse execution is especially useful when students become confused about how a particular segment of a program works and need to see it execute again, perhaps many times. In a traditional debugger, the program would need to be restarted from the beginning.

The Unix and Microsoft Windows versions of Dynalab are complete development environments which allow programs to be edited and compiled as well as executed. Since a Pascal compiler for the Java platform has not been developed yet, the Java version of the Dynalab animator can only execute precompiled programs. As noted earlier, a full version of the animator for animating Java programs, including a Java compiler for the E-Machine, is currently being developed.

FIELD

FIELD, the Friendly Integrated Environment for Learning and Development, is a programming environment designed for use in an educational setting [68]. Rather than being a stand-alone environment built from scratch, FIELD was built on and integrated with traditional Unix development tools such as *make*. Users can write programs in Pascal, Object-Oriented Pascal, C, and C++. At the heart of FIELD is a message server which passes information between the various tools that are used. Another important component of FIELD is the cross-referencing database which stores information gleaned from the source code and the compiler. A text editor interface allows students to write, compile, and step through the execution of the source code of their programs.

Besides the ability to watch programs step from line to line as they execute, other visualization tools are provided with FIELD. A data structure view allows students to see and modify their program's data structures. A call graph view of the program displays the call structure of programs and highlights nodes when they are active during program execution. FIELD also has components that provide a view of memory allocation and file I/O.

ZStep 95

ZStep 95 [45] is a program animator for a subset of the Common Lisp programming language. Like Dynalab, it allows students to write and step through the execution of a program in both forward and reverse modes. Its stepping capabilities are very flexible, allowing the user to easily select the granularity of the stepping. Stepping settings range from stopping at each individual expression to continuous execution of the program.

ZStep has some very innovative features for displaying expression and variable values in a running program. Rather than forcing users to "ping-pong" their attention between the source code display and a value display, ZStep provides a "value" window which aligns itself alongside each expression as it is executed. This window reveals the current value of the expression. Not only can the users see the current value, but they can also see a list of the values that the expression has taken on during the execution of the program. These historical values can also be filtered to show only those that meet specific conditions. Users can also browse values even more dynamically using the *Show Value Under Mouse* feature. When this feature is activated, the user can place the mouse pointer over any arbitrary expression in the program and the "value" window will appear beside it to display the current value of that expression.

Leonardo

Leonardo [79, 30] is a program animator with similar functionality to Dynalab. Using Leonardo, students can write and compile ANSI C programs using the standard C libraries. Like Dynalab, Leonardo allows students to step through the source code as the program is executing. Since programs are run on a special virtual machine, reverse execution is also allowed at any point.

Leonardo does not display the program stack or variable values when programs are executing, but it does have a graphics system that allows graphical views to be built on top of C programs. Special commands can be placed in the program's comments which direct the graphics system to take actions that illustrate what the program is doing. A preprocessor integrates these commands into the code of the program before compilation. Many animations for various algorithms and data structures have been developed for Leonardo using this system.

The current version of Leonardo only runs on recent Apple Macintosh computers, but a new version is being developed that will run on both Microsoft Windows and the Macintosh. In addition to becoming multi-platform, new features are also being added to Leonardo. The runtime environment is being redesigned to be more stable and support multi-threading. The graphics system is also being extended to allow interaction and to provide smooth transitions during animation.

Algorithm Animators

Algorithm animations are probably the most popular form of animation software used in computer science education. Data structures have been represented in textbooks and lectures as various kinds of diagrams for many years. In fact, it would be extremely difficult to explain most data structures and their associated algorithms (i.e., abstract data types) without resorting to diagrams of some sort. In this particular case, a picture is indeed worth (at least) a thousand words. Algorithm animations take these kinds of diagrams a step further by providing a dynamic view of the operation of an algorithm.

Although they are closely related to program animations, algorithm animations provide a layer of abstraction from the details of an algorithm's implementation. Rather than showing a literal view of how a computer would execute the algorithm, an algorithm animation usually provides just enough information to show the essence of the algorithm. It may also provide some extra "synthetic" information that would not be directly visible in a normal computer program [19].

A difficulty that occurs when static diagrams of algorithms are used in textbooks is that they must display changes to data structures as a series of snapshots. It is often difficult for students to perceive what changes occurred between snapshots and what actions caused the changes. Thus, it is logical to make use of computer graphics to make the progress of these changes explicit. Using animation software, changes in the structure of a diagram can be portrayed using smooth transitions from one state to the next. Students can see not only what changes resulted from the action, but also the exact nature of the changes themselves. Also, most algorithm animation software packages allow arbitrary data sets to be used. Students can see multiple examples of how the algorithm works. Instead of being restricted to the common cases of execution that can be fit into a textbook, students can view the algorithm performing in a virtually limitless variety of circumstances.

Since algorithm animations are relatively easy to produce, there are many software packages to choose from in this area. Rather than providing an exhaustive catalog of all algorithm animation software, only the most prominent or innovative examples will be discussed.

Sorting Out Sorting

The 1981 release of the 30-minute film, *Sorting Out Sorting* [4], is generally acknowledged to be the event that inspired much of the subsequent research into algorithm animation software. At that time, computer terminals capable of displaying color graphics were expensive and not usually available for classroom use. Thus, film and videotape were the only practical means for exposing students to algorithm animation software. *Sorting Out Sorting* provided a graphical representation of the actions of nine sorting algorithms along with voice narration explaining how each algorithm worked. At the end of the film, a "grand race" of all nine algorithms sorting the same data set was displayed. As the quicker algorithms completed sorting the data well ahead of the others, students received a powerful demonstration of the relative efficiency of each algorithm.

Although *Sorting Out Sorting* provided a more accessible view of sorting algorithms than a traditional lecture would have, it was still a passive and inflexible method of presenting the information. Instructors and students were not able to rerun the animations on different data sets or introduce animations of other sorting algorithms. Such activities require direct access to animation software.

Brown

Balsa Marc Brown's Balsa [19] framework for algorithm animation was one of the software packages that was inspired by *Sorting Out Sorting*. Balsa-II had a modular design that provided much functionality without restricting flexibility. Algorithms could be implemented in normal fashion in a high-level programming language and then annotated with special *animation events* at significant points in the source code. When a program was compiled for Balsa, a preprocessor converted these annotations into calls to special functions which broadcast messages to other parts of the system when significant events occurred. These event messages were delivered to *views* that provided a graphical representation of the algorithm's current state. Views could be written by the animation developer or prebuilt views could be used. In addition to the views, developers needed to specify *input generators* which provided data to the program. Input generators allowed the data that the algorithm processed to be customized to a wide variety of situations.

Balsa's modular design provided much flexibility for users as well. If multiple views were provided for an algorithm, all of the views could be displayed simultaneously or the user could choose to see only specific views. The user was also able to watch the same algorithm execute on multiple data sets simultaneously or multiple algorithms concurrently execute on the same data set. Using this framework, animations of many different algorithms were produced, including sorting, bin-packing, and graph traversal.

In addition to displaying interactive animations, Balsa was also able to record scripts of the actions in an animation session. This capability allowed animation sessions to be recorded for later playback or for broadcast to other computers running the Balsa-II software. This also allowed examples presented in a lecture to be saved for students to review or for an instructor to present as an animation to a group of students in a networked computer lab.

Zeus The Zeus system [20] was a further refinement of Balsa. Although Zeus had the same basic architecture as Balsa, it was designed to be more powerful and flexible. While Balsa was implemented in Pascal, Zeus was written in Modula-2, which provided many new features such as object-orientation and threading. Using Zeus, several innovations in algorithm animation were explored, such as the use of color, sound [21] and three-dimensional graphics [22] to convey information about an algorithm.

Collaborative Active Textbooks Collaborative Active Textbooks (CAT) [23] transferred the concepts pioneered in Balsa and Zeus to the world wide web. CAT had the same basic design as its predecessors; an implementation of an algorithm was annotated with event procedure calls that sent event messages to *views.* CAT was implemented in an interpreted, object-oriented language called Obliq. CAT's animations (oblets) could be embedded into web pages and displayed using specially modified web browsers. Since Obliq was designed to support distributed computing, oblets running in different browsers on different machines were able to easily communicate with each other. This allowed CAT's oblets to be used by many students simultaneously in classroom and lab situations. An instructor and students could view the same animation at the same time on different computers. Each user had control over some aspects of the animation's display, but all of the views were synchronized and only the instructor had control of the animation's execution.

The usefulness of CAT was limited by its implementation in Obliq, a little-known programming language that required a custom, in-house web browser. To work around this problem, JCAT, a port of CAT to the Java programming language, was created [26, 55]. The move to Java allowed JCAT applets to be run in all of the major web browsers available at that time. Later, JCAT was enhanced to allow the use of three-dimensional graphics in views [25].

Naps

Unlike the fully interactive approach taken by Brown, Naps' GAIGS (Graphical Algorithm Illustration through Graphical Software) was less interactive, but more flexible in some ways [56, 57, 62]. Algorithms to be animated by GAIGS could be implemented in any programming language. An algorithm's implementation would need to be modified to write out textual commands to a GAIGS "show file" as "interesting events" occurred during the algorithm's execution. The "show file" could then be loaded and interpreted by the GAIGS software to produce a series of static snapshots of the algorithm as it executed. Although this scheme did not allow students to modify the execution of the algorithm dynamically, it did allow the animation to be "rewound" if the student wished to review the steps of the animation.

The first version of GAIGS supported nine basic data structures including stacks, queues, and lists. To make the creation of animations easier, a library of abstract data types that automatically output GAIGS commands was written for the Pascal, Modula-2, and C++ programming languages.

Later versions of GAIGS increased user interaction by pausing execution of the algorithm after each snapshot was generated and displayed [63]. To provide a more complete learning environment, GAIGS was combined with web pages. Initially, the web-based implementation of GAIGS required that the GAIGS software be installed on the client machine [58], but, later, a Java applet (WebGAIGS) was created to display GAIGS "show files" that were generated on the server [59]. WebGAIGS was then enhanced to provide multiple snapshots in time-order sequence, to show multiple views of the same algorithm, and to display side-by-side comparisons of two different algorithms that solve the same problem [60].

Recently, GAIGS has been integrated into a client-server system, JHAVÉ (Java Hosted Algorithm Visualization Environment) [61]. JHAVÉ's server combines multiple animation systems that generate "show files" and sends these files to be displayed in a Java applet client. Combining multiple animation systems allows students to view both discrete snapshots using GAIGS and smooth-motion animations using Samba (discussed on page 22).

Stasko

Tango John Stasko's Tango [75] algorithm animation framework is based on the *path-transition paradigm*. An animation is made up of a number of graphical images, their locations, paths that they travel between locations, and transitions that they undergo (movement, color changes, etc.). Tango has a very different design from Balsa. Rather than being a self-contained system, Tango uses the functionality provided by the FIELD development environment mentioned in on page 13.

Construction of a Tango animation can be broken into three parts: "identifying fundamental operations in the algorithm, building animations of those operations, and mapping the algorithm operations to their graphical representations" [76]. When executing, a Tango animation consists of two Unix processes. The Tango process provides the graphics support needed to do the animation and another process provides an implementation of the algorithm to be animated. These two processes pass information back and forth to each other using FIELD's interprocess communication server. The algorithm can be implemented using direct calls to Tango procedures at points where interesting events occur or the implementation can be left untouched and the events can be included using FIELD's source annotation editor. Algorithm implementations that include the Tango procedure calls directly can be run stand-alone. If the Tango events are included using the annotation editor, the algorithm's implementation must be run in FIELD's debugger.

Polka Tango's animation paradigm was sufficient for simple animations, but, when used for constructing animations involving many concurrent transitions, it became unwieldy. To address this weakness, the Polka system was created [76]. Polka is very similar in concept to Tango, but an explicit animation clock was added to allow many simultaneous actions to be scheduled independently. In addition, Polka was implemented as an object-oriented tool set as opposed to Tango's procedural structure. In this restructuring, explicit support for multiple views was built into Polka, allowing multiple windows to display different representations of an algorithm.

Taking the concepts of Polka a step further, an enhanced version of Polka, Polka-RC, was developed to allow animation authors to specify precise times (in seconds and milliseconds). Using "real clock" times provides much simplicity to animation construction. For example, rather than needing to provide specific offsets for a trajectory, designers need only describe a path and the pace at which an object should traverse it.

Samba Samba is a simple animation language that was added on top of the Polka framework [77]. Like GAIGS (see page 20), any program in any programming language can be animated by emitting Samba commands on the program's standard output. Although Samba's command language does not provide access to all of the features of Polka, the subset of functionality that it exposes is sufficient for many sophisticated animations. It also makes creating an animation extremely simple. A student needs only to include print statements throughout a program to create an animation of it, rather than having to link in a library during compilation.

Concept Animators for the Theory of Computation

Although this area has not been researched nearly as heavily as algorithm animation, animation software for teaching the theory of computation has a fairly long history. Much of the effort has been expended developing simulators for the various types of automata that are commonly discussed in an introductory theory of computation course. The next most common category is applications for displaying parse trees and animating various parsing techniques. These types of animations are often developed to support compiler courses. Finally, there are a few programs that push the envelope and attempt to assist students in learning more abstract concepts such as the pumping lemma for regular languages.

Automata

Automata are the topic in the theory of computation that seems to be the easiest to animate. Although automata are really intangible mathematical abstractions, it is common practice to describe them as if they were actual physical machines. Turing machines are often described in terms of physical parts: a movable read/write head, an unbounded tape, and so forth. It is only natural that software has been written to visualize automata using their usual physical descriptions.

As with other types of educational animation software, there have been many different implementations of automaton animation software—far too many to be discussed exhaustively in this dissertation. Thus, only the most prominent examples in the literature will be discussed.

Hypercard Automaton Simulation As with algorithm animators, the early automaton simulators had somewhat primitive user interfaces. One of the early simulators, the Hypercard Automaton Simulation (HAS) [36], used a table-based representation of automata rather than the graphically more complex state diagram views. This software was available only for Apple Macintosh computers as it was written using Apple's Hypercard software. HAS could simulate finite state automata, pushdown automata, and Turing machines.

Turing's World Barwise and Etchemendy's *Turing's World* [5] is probably the most sophisticated automaton simulator that has been developed to date. Like HAS, Turing's World is an application that runs on Apple Macintosh computers, but Turing's World's has a far more elaborate representation of automata as state diagrams. The primary purpose of Turing's World is to simulate Turing machines. It is also possible to build finite state automata as restricted versions of a Turing machine. Pushdown automata are not supported directly.

Some of Turing World's unique features include the ability to create submachines, schematic machines, wild card transitions, process tree views of nondeterminism, and annotations. Submachines are small Turing machines for performing a specific task. They function much like functions or procedures in imperative programming languages. Using submachines, students can use Turing's World to create very complex machines without having to construct one large, unwieldy state diagram. Schematic machines are the complement to submachines. Schematic machines are essentially template Turing machines that contain “holes” into which submachines can be placed for a specific application. Wild card transitions provide a powerful way to create Turing machines that use a large alphabet. Rather than being forced to specify a transition for each symbol in a large range, a wildcard transition uses an ellipsis in place of one of the alphabet’s symbols to indicate that that transition should be taken if the current input symbol is not specified in other transitions from that state. Process trees display the execution of a nondeterministic automaton by displaying a tree of the states that are entered as the automaton executes. Automata built in Turing’s World can also include annotations, which are textual notes that explain the function of an automaton.

<u>TUMS</u> TUMS (TUring Machine Simulator) [53] was written to bring the functionality of Turing’s World to the Sun Microsystem’s OpenWindows environment on the SPARC architecture. In addition to providing most of the features found in Turing’s World, TUMS could also be used to simulate pushdown automata.

<u>NPDA, FLAP and JFLAP</u> Development of automaton simulation software led by Susan Rodger began with NPDA [29], a system for simulating nondeterministic pushdown automata (NPDAs). It was written using Unix’s X11 windowing environment and the Athena widget toolkit. Users can load and modify prebuilt NPDAs from a file or create one from scratch. The NPDA can then be run on an input string. It can handle up to sixteen different nondeterministic configurations. The individual configurations are displayed below the state diagram of the NPDA. At

every step, each configuration's state, stack, and input tape are updated. If a configuration is at a nondeterministic step, new configurations are created. Individual configurations can be killed or paused for convenience.

One weakness of NPDA's interface that was inherited by its successors is the inability to display multiple arrows for transitions that have the same endpoints. For example, if an automaton contains a transition that moves from state 1 to state 2 and another transition from state 2 to state 1, both transitions will be represented with a single arrow that has two labels. Each label contains an arrow head that indicates which direction the transition for that label travels. Although this method allows multiple transitions between the same endpoint states, it can be very difficult to understand at first.

NPDA later became FLAP (Formal Languages and Automata Package) [46, 47, 71]. Like NPDA, FLAP was developed for the X11 windowing environment. In addition to nondeterministic pushdown automata, FLAP also simulates finite state automata and Turing machines with one or two tapes.

As mentioned on page 9, after the release of the Java programming language, many educational animation projects were ported to this language. FLAP followed this course, becoming JFLAP (Java Formal Languages and Automata Package) [67, 8, 34, 38] (see Figure 2.2). JFLAP can be run as either a stand-alone application or as an applet in a web browser. JFLAP has all of the features of FLAP. In addition, many of the algorithms for processing automata have been animated. Students can watch a nondeterministic FSA be transformed into a deterministic FSA and a deterministic FSA transformed to contain a minimal number of states. FSAs can be converted to regular expressions and regular grammars. Regular expressions and regular grammars can also be convert to FSAs. The same is also true for PDAs and

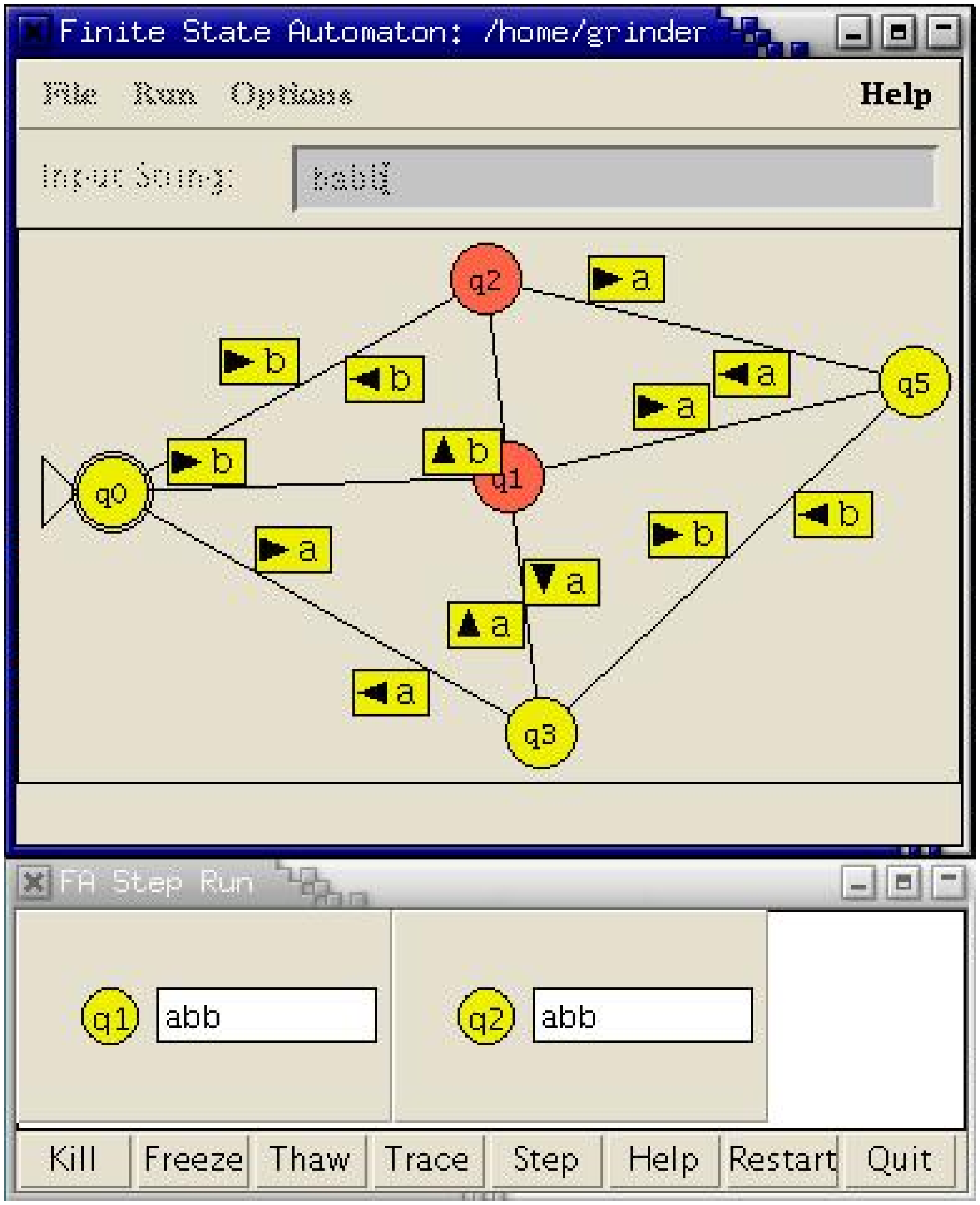


Figure 2.2: A Nondeterministic FSA in JFLAP

context-free grammars.

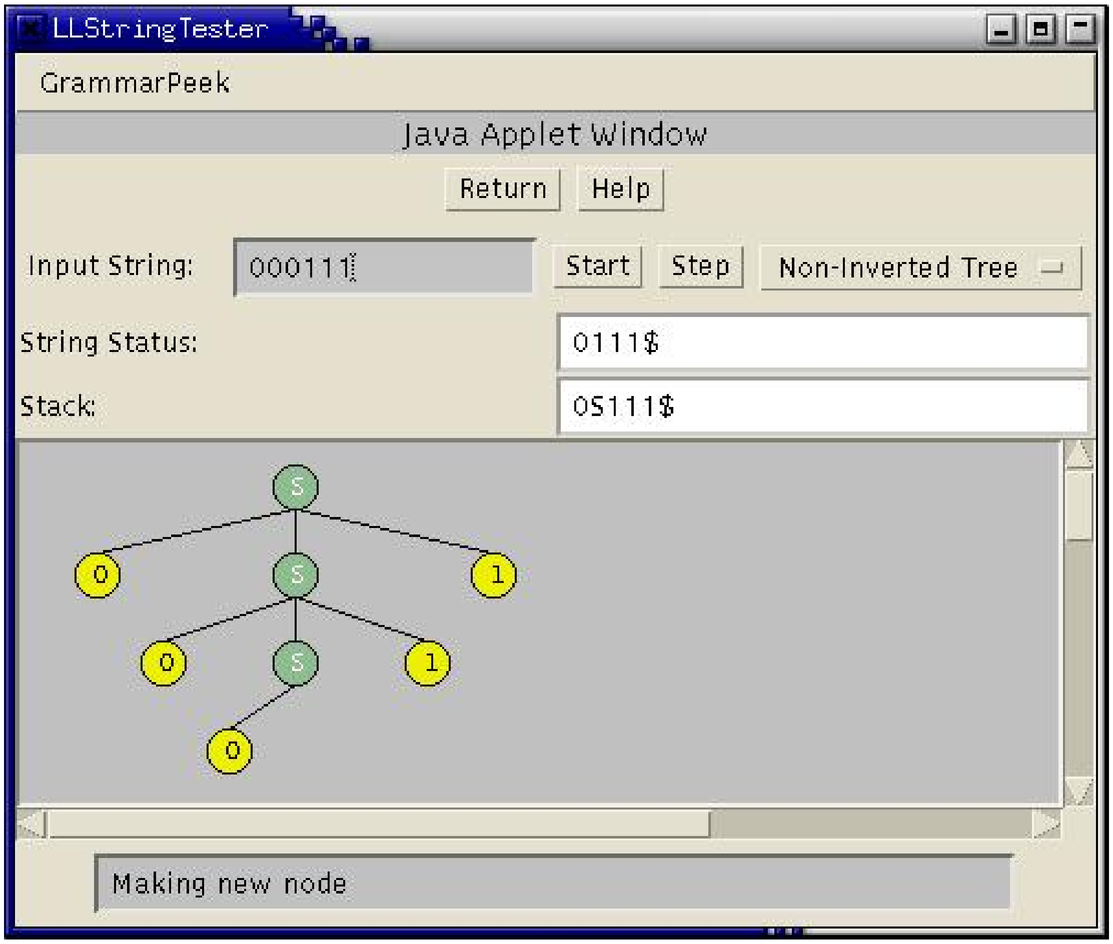
JCT The Java Computability Toolkit (JCT) [70] is animation software similar to JFLAP. JCT initially supported only the simulation of finite state automata and

Turing machines, but it was later extended to also simulate pushdown automata.

JCT was built using the “Swing” user interface toolkit that was introduced into version 1.2 and later of the Java Runtime Environment. Unlike Java’s original Abstract Windowing Toolkit (AWT), Swing draws its own widgets, which allows Java applets and applications to have a consistent look-and-feel across different computing platforms. JCT can be used as either an applet in a web browser or as a stand-alone application, although when using it as an applet, security restrictions in the Java virtual machine prevent it from loading or saving files as well as some other activities. The interface of JCT is quite complex with multiple windows and many different toolbars for constructing and simulating automata. The architecture includes a pluggable interface that allows new functionality, such as the pushdown automaton environment, to be added easily.

The finite state automaton environment allows finite state automata to be constructed as state diagrams and simulated on arbitrary input strings. In addition to standard simulation, the closure operations such as minimization, concatenation, intersection, and others can be performed on finite state automata. A unique feature of JCT’s finite automaton environment is its “circuit board diagram”, which represents an FSA as a grid with the states on the diagonal.

The Turing machine environment of JCT is perhaps even more sophisticated than that of *Turing’s World*. Like *Turing’s World*, JCT allows Turing machines to use other Turing machines (submachines) as subroutines. Any Turing machine created in JCT can be used as a submachine. Five submachines for common movements of the read/write head are included on one of JCT’s default toolbars. Tape information that is passed between submachines can be either locally or globally scoped, allowing complex Turing machines to be simplified. JCT also provides



Input String: 000111

String Status: 0111$

Stack: 0S111$

Making new node

Figure 2.3: JeLLRap's Parse Tree View

"variable transitions" that are used to represent bulky groups of transitions and local or global scoping of tape squares passed between submachines.

A drawback of JCT's sophistication is its complexity. When the software starts, the user is suddenly faced with a large array of buttons and toolbars, many of which do not have an obvious function. It would take quite a bit of time for a novice to become productive in the JCT environment.

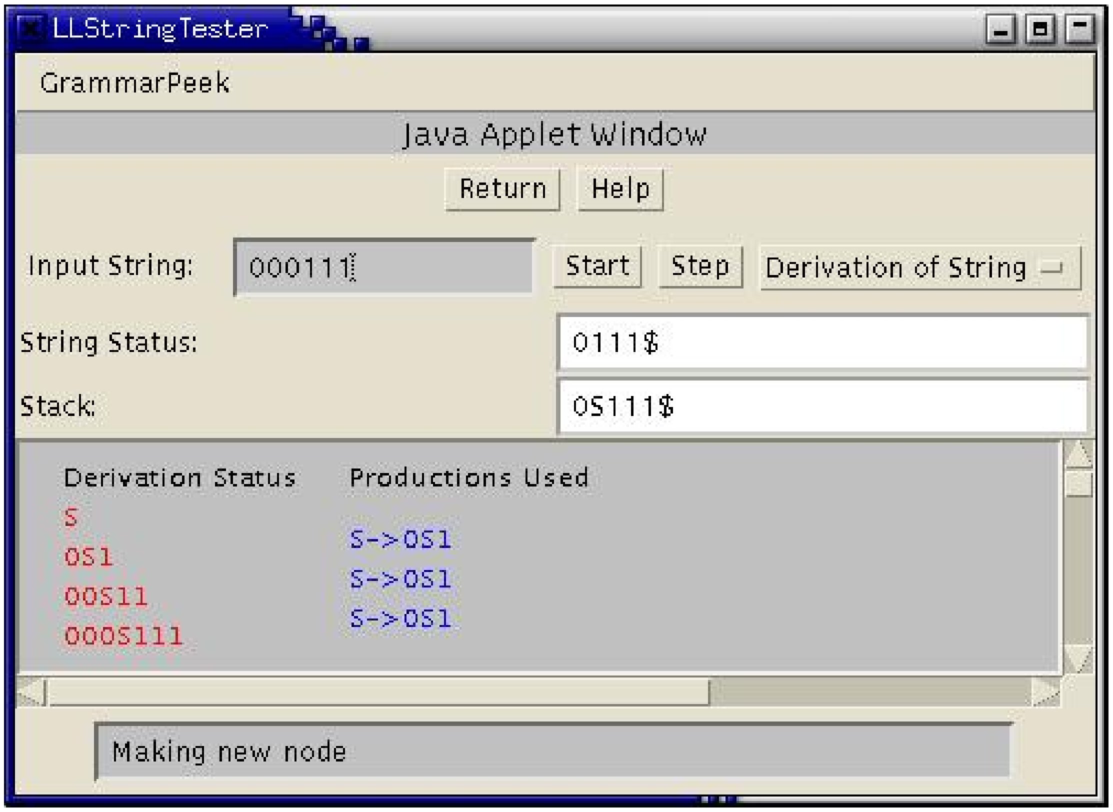
Context-Free Grammars

Susan Rodger LLparse and LRparse [10, 8] are software packages for animating the parsing of LL(1) and LR(1) context-free grammars. When running one of these packages, the user is first asked to enter a context-free grammar. Once the grammar has been created, the user must determine the first and follow sets for each nonterminal in the grammar and then create a parse table. When a parse table has been successfully built, the user reaches the actual parsing animation. The user can enter a string to parse and then observe each step of the parse including which portion of the string has been parsed at each step and what the current state of the parse stack is. The original LLparse and LRparse were developed for the X11 windowing environment on Unix. The two programs were later ported to Java and combined into a single package named jeLLRap [34]. In addition to LL(1) and LR(1) grammars, jeLLRap now also supports LR(2) grammars. At the parsing animation stage, in addition to the string and the stack, the user can also view the construction of a parse tree or a step by step string derivation (see Figures 2.3 and 2.4).

Pâté [8, 38] is a software package that is similar to jeLLRap, but works with arbitrary context-free grammars rather than being restricted to an LL or LR subset.

Webworks Laboratory Projects Members of the Webworks Laboratory at Montana State University have been working on two projects for animating concepts related to context-free grammars. The first project, the Parse Tree Applet, is an animation that allows the user to interactively use a grammar to generate a parse tree. The other grammar-related project is an applet that animates the generation of first and follow sets for LL(1) grammars.

The Parse Tree Applet, originally developed by Jessica Lambert [14] and later



Input String: 000111

String Status: 0111$

Stack: 0S111$

Derivation Status	Productions Used
S	
0S1	S->0S1
00S11	S->0S1
000S111	S->0S1

Making new node

Figure 2.4: JeLLRap's String Derivation View

enhanced by Teresa Lutey [35] allows students to load a context-free grammar and build a parse tree using the rules of the grammar. The user interface of the Parse Tree Applet (see Figure 2.5) contains two main elements. In the upper right corner of the applet, a window containing the rules of the context-free grammar is displayed. In the lower half of the applet is a window that displays a parse tree view of the rules that have been applied so far. Users can select from a set of predefined grammars, create a new grammar, or edit an existing grammar.

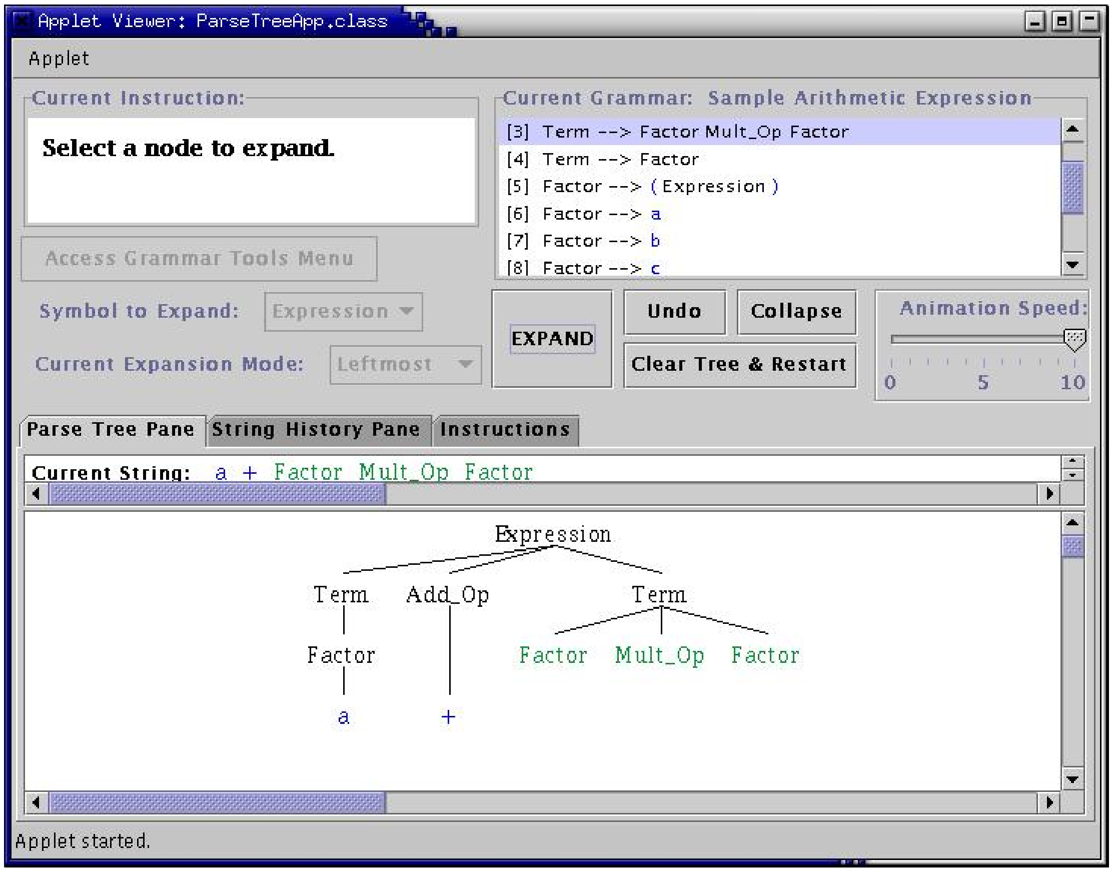
Before creation of the parse tree begins, the user can select a nonterminal symbol with which to start the tree and the expansion mode for the tree: leftmost, rightmost, or any node. The starting nonterminal will be placed as the root node of the parse tree. The user can then select the root node by clicking the mouse on it and then select a grammar rule from the grammar window to apply to that nonterminal. After a node and a grammar rule have been selected, clicking the button labeled *Expand* will add the symbols from the right-hand side of the selected rule as the selected node's children. The tree will be redrawn in smooth fashion to accommodate the new nodes below their parent.

Unexpanded nonterminal nodes are displayed in green. Nonterminal nodes that have been selected for expansion are displayed in red. Terminal nodes are, of course, leaf nodes, and are displayed in blue at the lowest level of the tree, so that the string that is being generated can always be read easily from left to right. At any point during expansion, the user can undo expansions and collapse entire subtrees back to an unexpanded state.

Pumping Lemmas

It is very difficult to produce active learning software for some topics in the theory of computation. Topics like automata and grammars have had graphical representations for many years. Adapting such graphical representations to active learning animation software has been a relatively simple task. In contrast, very few attempts have been made to produce active learning software for topics such as the pumping lemmas for regular and context-free languages. In fact, the PumpLemma software [8] developed by students under the direction of Susan Rodger is the only example in the literature of active learning software being created for such a difficult topic.

PumpLemma assists students in proving that specific languages are not regular. The user interface consists of a series of text boxes which the user must fill out in the correct order (see Figure 2.6). First the student must specify the language to



Select a node to expand.

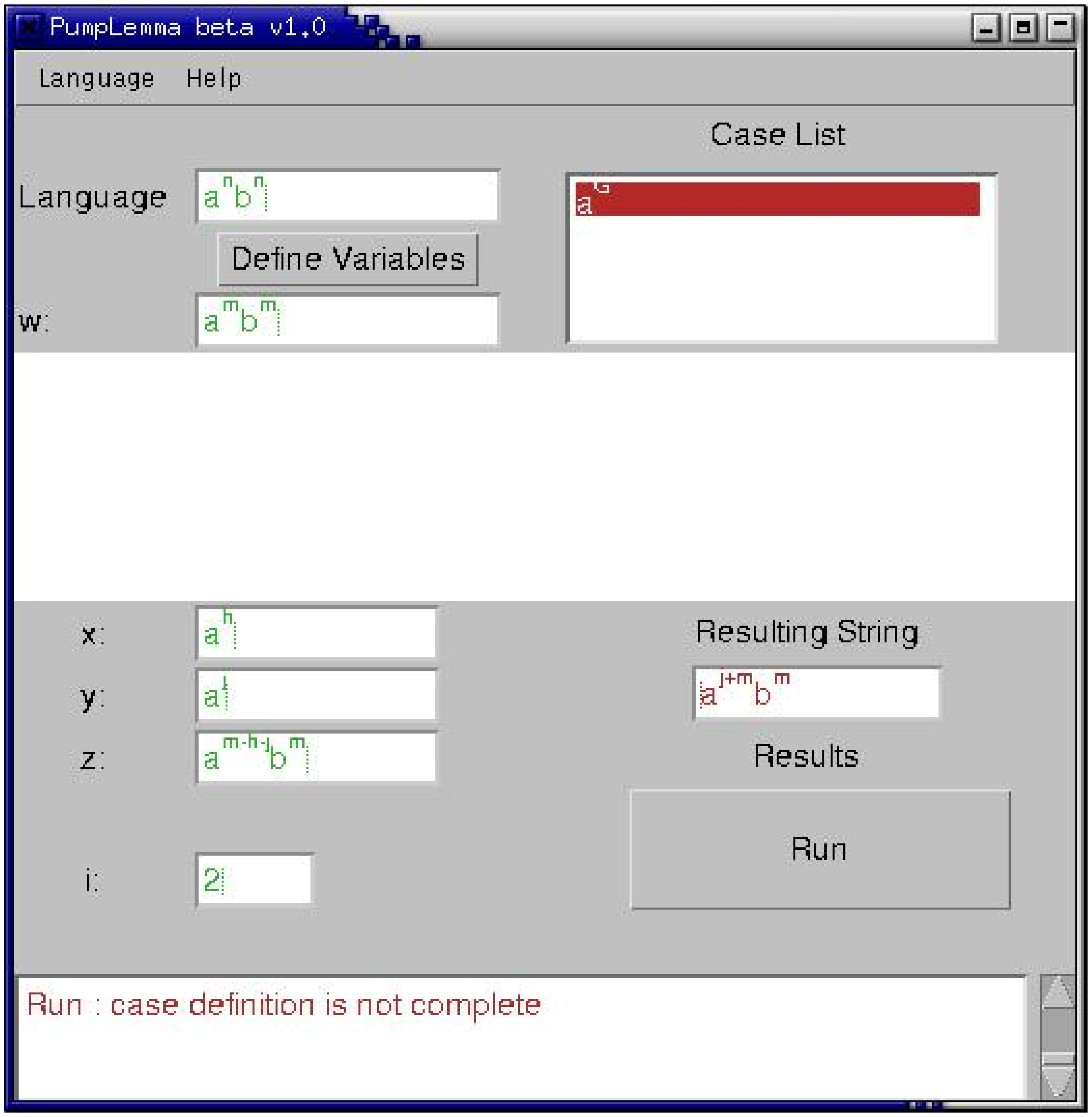
[3] Term --> Factor Mult_Op Factor
[4] Term --> Factor
[5] Factor --> (Expression)
[6] Factor --> a
[7] Factor --> b
[8] Factor --> c

Current String: a + Factor Mult_Op Factor

Applet started.

Figure 2.5: The Parse Tree Applet

be investigated. PumpLemma restricts the languages to relatively simple strings such as a^nb^n and $a^{\frac{n}{2}}b^n$. Once a language has been specified, the student can define the range of the variables used in the language description. For example, the n used in Figure 2.6 was set to be > -1. After that, the student needs to enter a string from the language that does not pump as specified in the pumping lemma. The pumping length of the language, m, is available to be used when entering the string. Once the string has been entered, PumpLemma will determine how many cases the student must consider to prove that the language is not regular. The cases are displayed in the list box on the upper right side of the interface. For each one of these cases, the student needs to divide the string into parts x, y, and z and then



Language $a^n b^n$

w: $a^m b^m$

Case List: a^G

x: a^h

y: a^j

z: $a^{m-h-j} b^m$

i: 2

Resulting String: $a^{j+m} b^m$

Run : case definition is not complete

Figure 2.6: PumpLemma in action

indicate how many times the y substring is to be repeated in the i text box. When those four text boxes have been filled in, the student can press the button labelled "Run" to check if the resulting string is not in the language. If so, the string will be displayed and the student can move on to the next case. When all cases have been shown to not pump for some i, PumpLemma will display a message indicating that the language has been proven to not be regular.

PumpLemma is an intriguing piece of educational software. Unfortunately, the publicly available PumpLemma package is a beta version that was released in 1997. The software is very buggy and the interface is awkward and confusing. Although there were plans to complete its development, all work on PumpLemma appears to have stopped.

Evaluation of Educational Animation Programs

Importance

Formal evaluation of the effects of interactive animation software on learning is extremely important. Although many intuitively feel that visualization software will improve students' comprehension of complex topics, initial expectations of how students will use a particular resource and what effect it will have on their learning often prove to be highly inaccurate. Students often tend to look for the most efficient way to complete an assignment without any consideration of how that method affects what they learn (or whether they learn at all). Even if a visualization is useful for learning, its use and environment may need to be "tuned" to get the most benefit for students.

Difficulties

Despite its importance, very little empirical evaluation of interactive animation software has been done. Educational evaluation is a complex process that is fraught with difficulties. To be done properly, it requires a large, carefully selected pool of subjects who are representative of the target population, meticulously designed materials for teaching and testing, and the ability to isolate groups of students from each other during the evaluation. Most educators in computer science do not have a background in educational research and rarely have the time or resources needed to learn about and deal with the complexities of formal evaluation methods [1, 73].

Past Efforts

Nearly all of the researchers who have developed animation software for computer science education have made some attempt at evaluating its effect on students. Much of this evaluation focuses on anecdotal evidence and quantification of the students' subjective response to using the animation software (for example [46]). Very little effort has been expended to generate empirical data on the effects of animation software on learning. The two most notable efforts at evaluating the effects of educational animation software on learning have been done by Richard Mayer and John Stasko.

Richard Mayer

Richard Mayer's research has focussed mostly on the use of illustrations and passive animations for explaining mechanical topics such as the operation of hydraulic brakes and bicycle tire pumps. Mayer's first study [48] examined how illustrations in scientific texts influenced readers' understanding and retention of the material. He found that labeled illustrations in a text increased "explanative recall" and "problem-solving transfer", but did not increase retention of non-explanative material or verbatim recall. In other words, the illustrations increased the readers' ability to explain how a mechanical system worked and to apply that knowledge to new situations, but did not increase their understanding of material that was not directly related to the general idea of the mechanical process. A second study with Joan Gallini [51] found very similar results when using a series of illustrations that depict labelled parts of a mechanical system and the steps that these parts go through when the system is functioning ("parts-and-steps" illustrations).

Mayer later turned from static illustrations to multimedia animations. In two studies with Richard Anderson, he examined how non-interactive computer animations of mechanical systems when coupled with verbal explanations of the system affected understanding. In the first study [49], they found that students who viewed an animation with a simultaneous verbal explanation performed better on a problem-solving test than students who received a verbal explanation separately from the animation. A similar, but larger, second study [50] found similar results. The group that viewed an animation concurrently with a verbal narration outperformed all other groups on a problem-solving test.

Mayer's results, while not directly related to interactive animation software for computer science education, provide evidence that animation software can be beneficial to learning and provide a foundation for evaluating other forms of educational animation software.

John Stasko In addition to the development of algorithm animation software at Georgia Tech's *Graphics, Visualization, and Usability Center*, John Stasko and his associates have also pioneered the empirical evaluation of visualization software for computer science education.

In an initial exploratory study [3], the group found that students were receptive to animations and wanted to see them used in the classroom. They also concluded that factors that should be considered in empirical studies should include the students' academic and technical background, spatial abilities, and prior experience with visual technologies.

The first empirical evaluation [43] used an animation of Kruskal's minimum spanning tree algorithm in a classroom and lab setting. All groups in the study attended a lecture about the algorithm. Some of the groups attended a lecture that used slides to illustrated Kruskal's algorithm while other groups viewed a lecture that used the animation. These groups were further divided into those who participated in a lab and those who did not. The lab groups were also subdivided into those who constructed their own graphs on which to run the algorithm (active lab) and the those who viewed animations based on prepared data files. Two tests were administered after the treatment, a multiple-choice/true-false test and a free response test. The lab groups performed better than the lecture-only groups on both tests. The active lab group performed better than the passive lab group on the free response test, but there was no significant difference between the lab groups on the multiple-choice/true-false test.

A second empirical evaluation [27] examined the effect of animations on both non-computer science and computer science students. Participants watched a video-taped lecture and read a text about an algorithm. Some of the students watched an animation and others did not. Half of the animation and non-animation groups were asked to predict the next step of the algorithm in a situation while the other half only viewed the material without making any predictions. All students took the same post-test after the treatment. There were no significant differences on the results of the post test for any of the groups.

After the discouraging results of the second evaluation, Stasko's group reevaluated their approach. Stasko [77] tried a different use of animations. Rather than presenting students with a prebuilt algorithm animation, he required the students to build their own animations for two algorithms with the Samba animation tool. In later exams, students answered questions about the two algorithms nearly perfectly.

Meanwhile observational studies were also performed to better determine how students use animations. Kehoe and Stasko [40] observed three graduate students complete a “homework” assignment about binomial heaps using a text with pseudocode, an animation, and static diagrams. They found that students used the animations in successful learning strategies, mostly using the animations to learn the steps of the algorithm. They also observed that the students needed better ways to make connections between the different representations of the algorithm. The students claimed that the animations helped them learn more quickly. A second observational study [41] was performed with twelve students. Half of the students used animations while the other half used only a text to complete a “homework” assignment on binomial heaps. After the students completed the assignment, they all took a post-test. The students who used the animation did significantly better on the post-test than those who did not. The authors observed that animations were more useful in open, interactive situations rather than in closed exam situations. They also noted that animations appear to make algorithms more accessible and less intimidating. Animations also appeared to increase student motivation. The researchers also concluded that animations were best used for learning procedural operations.

Conclusions

Although much work has been done to develop interactive learning software for computer science education over the past 20 years, this field of research is still in its infancy. There is very little evidence of collaboration among the researchers in the field. Much work is duplicated by researchers at different institutions. One of the major investments that a research project must make is to develop the software infrastructure on which their later work will be based. In the current situation, each project must invest a significant amount of time setting up infrastructure. Should the primary investigator lose funding, retire, or just lose interest in a project, the software usually languishes and eventually disappears, bringing little benefit to researchers who follow, other than some journal publications.

The products of many animation projects often do not see use outside of the institution where they were developed. While it is often useful to have multiple "competing" research efforts, given the relatively small number of resources allocated to computer science education research, it may be better for the field as a whole if researchers were to pool their efforts more. For example, establishing a publicly available archive of source code from various projects would ensure that progress made in one area would be available to researchers and programmers to build upon when embarking on new projects.

Other obstacles to the use of interactive learning software are beginning to be eliminated. Now that most projects use the Java programming language, platform-dependence has become less of an issue. The time-consuming complexity of integrating these tools into a course is still a major hurdle that needs to be overcome for animation software to see widespread use in computer science education. To alleviate this problem, more work needs to be done to improve the ease of use of this software for both the instructors and students. There are many ways to do this including integrating the software into a single course resource such as a “hypertextbook” or modifying the software to be easy to install and adapt to a wide variety of educational situations.

Although evaluation of the effects of active learning animation software on students’ learning is recognized as important, much research still needs to be done. Research into the evaluation of animators would also benefit from collaboration between institutions. Such collaboration would allow resources to be pooled and more effective evaluation methods to be developed.

CHAPTER 3

THE FSA SIMULATOR

Introduction

The FSA Simulator is the main product of the research performed for this dissertation. It is similar in function to other automaton simulators, notably JFLAP and JCT, but its focus is exclusively finite state automata (FSAs). The FSA Simulator also has a number of unique features that set it apart. The most notable is its capacity for FSA construction exercises that allow a student to check answers and receive feedback. Feedback is given in such a way that the student is guided toward the correct answer without the answer being revealed.

The FSA Simulator is written in the Java programming language and can be executed as a stand-alone application or embedded as an applet in a web page 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 the browser 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.

User Interface Overview

The user interface of the FSA Simulator is fairly simple (see Figure 3.1). Most of its display is taken up by the state diagram panel in the lower left corner. An FSA is displayed in this panel as a state diagram, similar to diagrams found in

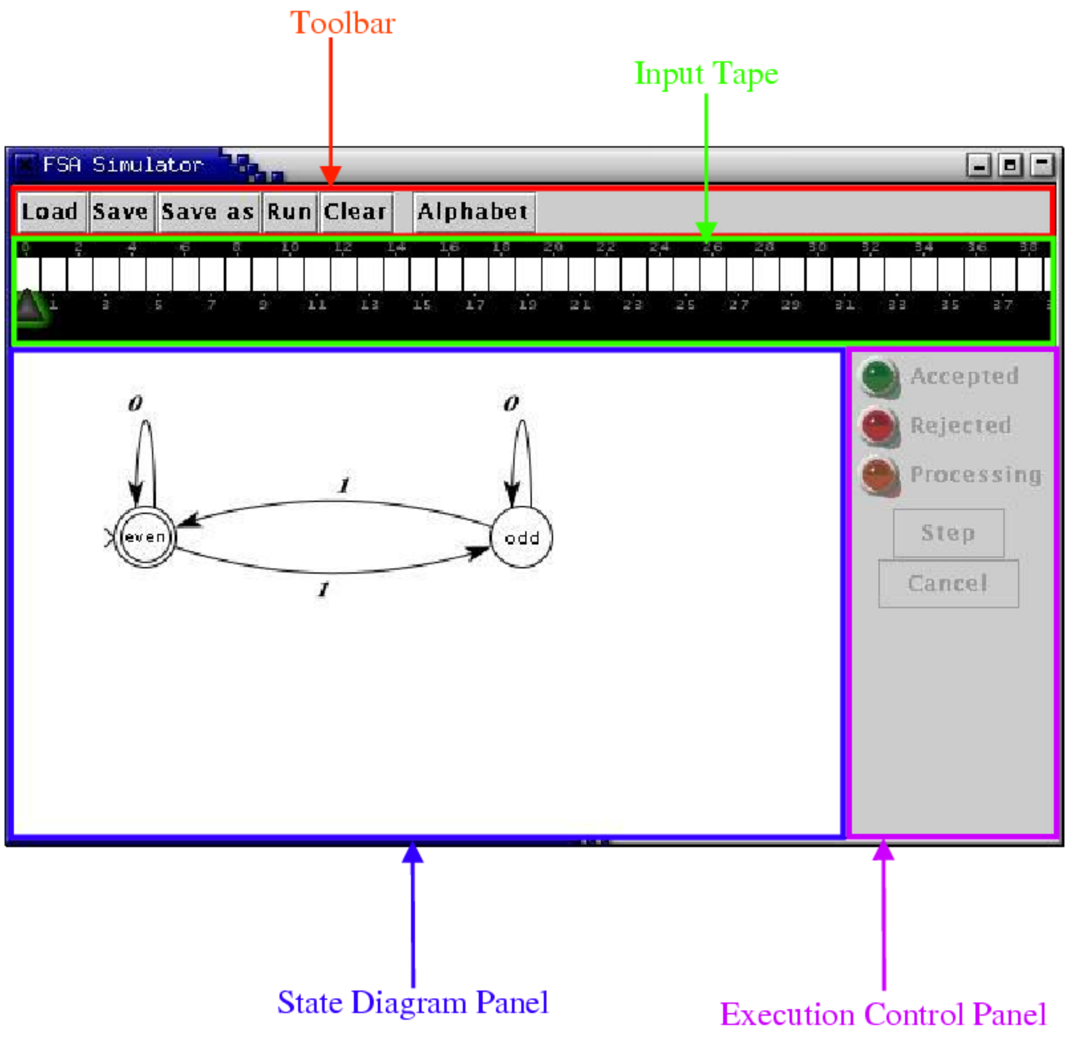


Figure 3.1: User Interface of the FSA Simulator

textbooks. The state diagram panel also serves as the main interface for creating and modifying FSAs.

Above the state diagram display is a panel that represents the FSA's input tape. In this panel, a user can enter and edit a string for the currently loaded FSA to process. A triangular tape marker, positioned below the tape squares, is used to indicate which symbol of the input string is currently being processed by the FSA.

Above the input tape panel, a toolbar with a set of buttons is 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, and for modifying the FSA's alphabet—labelled *Run*, *Clear*, 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—labelled *Load*, *Save*, and *Save As*—are also displayed. (Figure 3.1 is a snapshot of the stand-alone application version of the FSA Simulator.)

To the right of the state diagram panel is the execution control panel. If the currently displayed FSA is not processing a string, all of the items on this panel are disabled. However, in *Run* mode, the *Step* and *Cancel* buttons for controlling the execution of the FSA are enabled as well as a set of three "lights" that indicate the current execution state of the FSA while processing.

Basic Operation

The most basic form of active learning exercise that can be done with the FSA Simulator is to have it load a prebuilt FSA, say M, from a file and allow students to test input strings for membership in the language of M. The user can enter symbols for the input string in the tape panel by clicking on the tape panel with the left mouse button and then typing characters from the FSA's alphabet on the keyboard. The tape marker serves as a cursor when the tape is being edited, positioning itself under the tape square that is currently available for editing. The marker can be moved about the tape squares using standard editing keys, including the left arrow, right arrow, and backspace keys.

When the user is ready for the FSA to process the string displayed on the tape, the *Run* button located above the tape panel can be clicked. Doing this shifts the Simulator into *Run* mode. When entering *Run* mode, the tape marker moves to

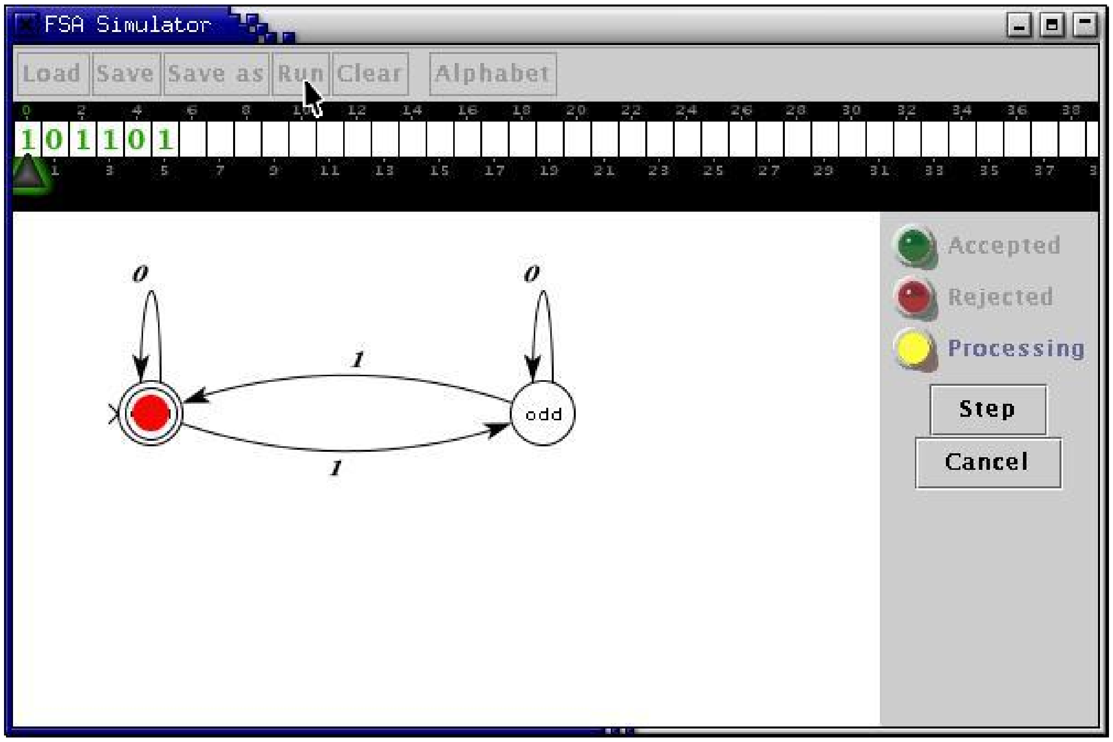


Figure 3.2: The FSA Simulator After Entering "Run" Mode

the first square on the tape, the *Step* and *Cancel* buttons in the execution control panel are enabled, and a solid red circle appears inside the start state (the state marked with the > symbol) in the state diagram panel (as shown in Figure 3.2). A solid red circle inside a state indicates the current state of the FSA.

In the execution control panel are three images that simulate light emitting diodes (LEDs) and two buttons. The LEDs are labelled *Accepted*, *Rejected*, and *Processing*. Upon entering *Run* mode, the yellow *Processing* LED lights up to indicate that the FSA is in the midst of processing a string. From this point on, while processing an input string, the Simulator repeatedly performs the following two actions:

1. It awaits a left mouse click on the *Step* button.

2. When it recognizes that the *Step* button has been clicked, the Simulator causes

the FSA to process the input symbol above the tape marker. Based on the current state and the current input symbol, it determines what the next state should be. The Simulator then moves the solid red circle smoothly along the transition arrow labeled with the current input symbol to that next state. At the same time, it advances the tape marker to the next square on the tape (see Figure 3.3).

When the tape marker reaches the end of the input string, the Simulator pauses one more time and waits for the user to click *Step*. This time when *Step* is clicked, the *Processing* LED will turn off, and, if the FSA is in an accept state (a state drawn with a double circle), the green *Accepted* LED will turn on; otherwise, the red *Rejected* LED will turn on (see Figure 3.4). If the FSA accepts the string, the state indicator circle, which must now be in an accept state, also changes color from

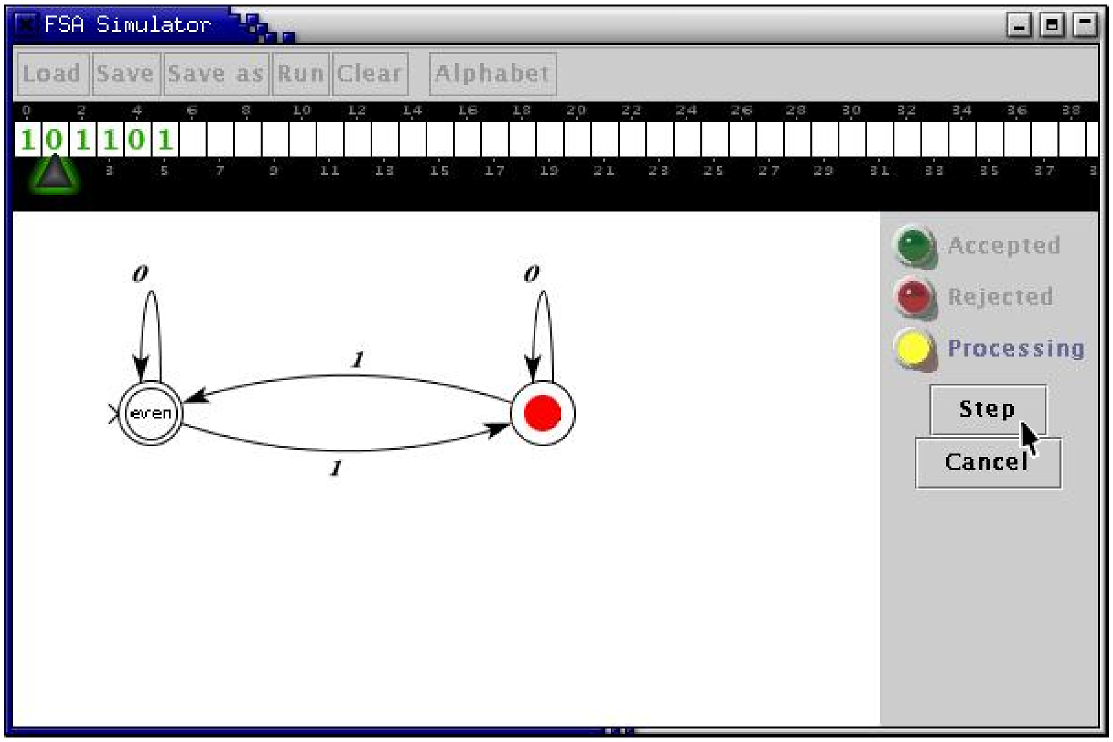


Figure 3.3: The FSA Simulator During Execution

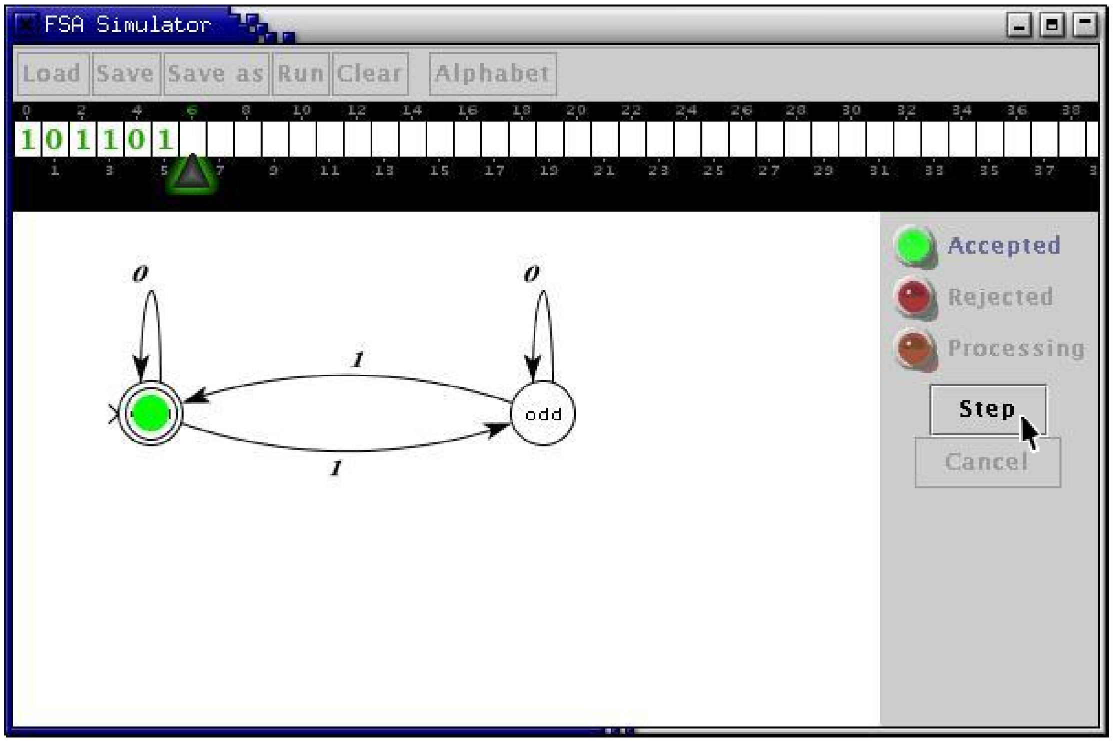


Figure 3.4: The FSA Simulator Accepting a String

red to green. Clicking the *Step* button once more causes the simulation to stop (*Run* mode is terminated).

FSA Construction and Modification

Beyond simply allowing the user to test strings with an FSA, the FSA Simulator also allows FSAs to be constructed and modified. This allows users to be more actively involved while learning about FSAs.

The state diagram panel serves as the main interface for creating and manipulating the structure of a finite state automaton. A user's interaction with the state diagram panel was designed to be as simple as possible. Rather than having multiple editing modes for creating and modifying states and transitions, all modifications of an FSA can be done through simple mouse actions. States are created

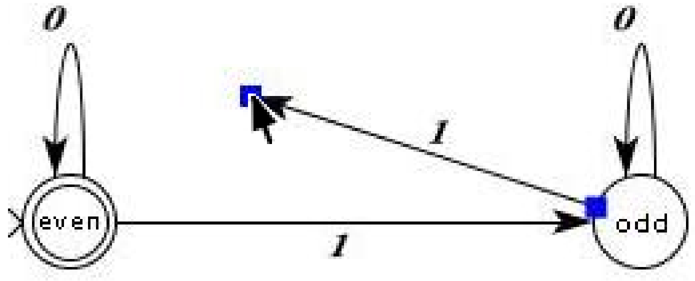


Figure 3.5: Moving a Transition in the FSA Simulator

by placing the mouse pointer over an unoccupied portion of the panel and clicking the left mouse button while holding down the Ctrl key on the keyboard. Similarly, a user creates transition arrows from one state to another by holding down the Ctrl key and pressing down on the left mouse button while the pointer is inside the first state of the transition; the user then moves the mouse pointer (i.e., drags the arrow) to the destination state and releases the left mouse button. As the pointer is dragged out of the first state, a transition arrow representing the new transition follows the mouse pointer until a destination state is reached, at which point the arrow tip of the transition attaches itself to that state (whereupon the left mouse button can be released). The position of a state can be changed by clicking the left mouse button inside of that state, and dragging the state by moving the mouse to the new position for the state, where the mouse button is released.

Endpoints of transition arrows can also be moved from state to state. Clicking on a transition arrow will cause square "handles" to appear on the arrow's endpoints. These handles can be dragged from one state to another, causing the transition to change one of its endpoints (see Figure 3.5). During the creation or movement of a transition arrow, the arrow will not attach to states which already have a transition connecting them to the state at the other end of the arrow. Multiple transitions

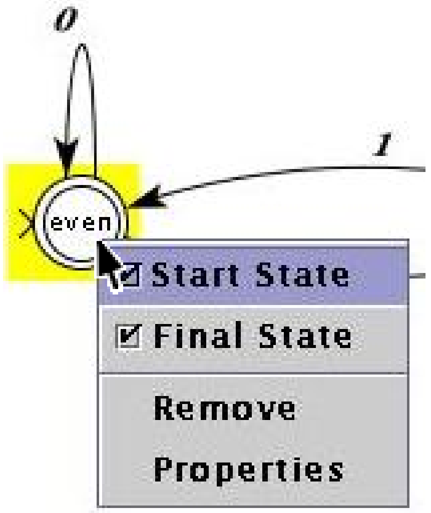


Figure 3.6: The State Popup Menu

from one state to another are denoted in the usual fashion by labelling the transition arrow with multiple symbols.

The properties of states and transitions can be changed using the popup menus that appear when the right mouse button is clicked over a state or transition. The state popup menu (see Figure 3.6) allows a state to be designated as the start state or one of the final states. There are also items for editing the label of a state and the state's tooltip description (see Figure 3.7) under the *Properties* menu item. (A tooltip description is a short piece of text that appears near the mouse pointer after it remains motionless above a state for certain amount of time.) States (and their associated transition arrows) can be deleted from the diagram by selecting the *Remove* menu item.

On the transition popup menu (see Figure 3.8), two items are available. Clicking the *Edit Symbols* item will cause a dialog box to appear that will allow the user to select symbols from the alphabet to be included in the label of the transition arrow. Just as in the state popup menu, clicking the *Remove* item, will delete the

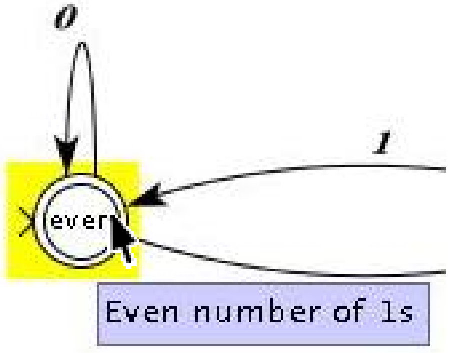


Figure 3.7: A State Tooltip Description

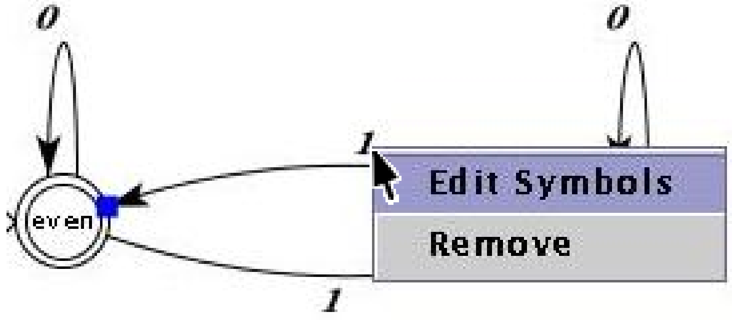


Figure 3.8: The Transition Popup Menu

transition arrow from the state diagram.

Among the buttons in the toolbar above the input tape panel are some that can be used when constructing or modifying an FSA. Pressing the *Clear* button will erase the current FSA from the state diagram panel. Clicking on the *Alphabet* button will cause a dialog box to pop up that allows the user to choose which symbols to include in the alphabet of the FSA (see Figure 3.9). The alphabet dialog currently allows only symbols typically found on US English keyboards to be selected, but this restriction could easily be eliminated.

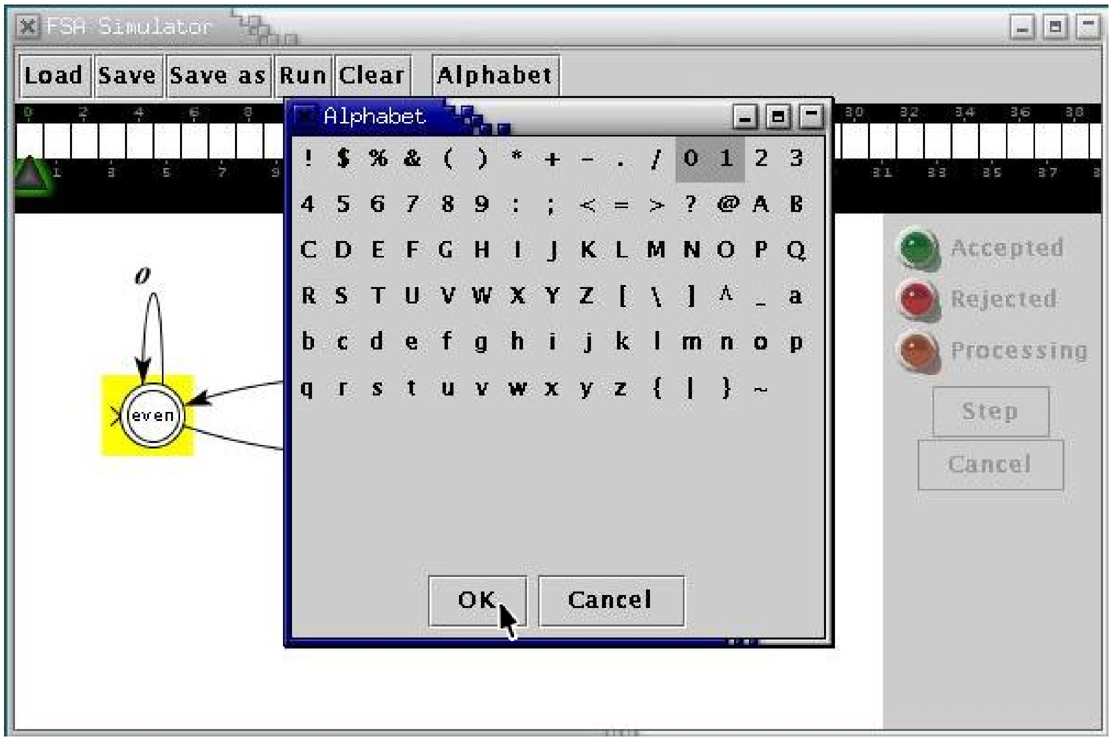

FSA Simulator
Load
Save
Save as
Run
Clear
Alphabet
Alphabet
! $ % & ( ) * + - . / 0 1 2 3
4 5 6 7 8 9 : ; < = > ? @ A B
C D E F G H I J K L M N O P Q
R S T U V W X Y Z [\ ] ^ _ a
b c d e f g h i j k l m n o p
q r s t u v w x y z { | } ~
OK
Cancel
0
even
Accepted
Rejected
Processing
Step
Cancel

Figure 3.9: The FSA Simulator's Alphabet Selection Dialog

FSA Construction and Verification Exercises

The FSA Simulator goes beyond just allowing new FSAs to be constructed and executed. Using features unique to the Simulator, FSA construction and verification exercises can be written in which students are required to construct an FSA to recognize a given language. The FSA Simulator will then guide the user toward a correct solution. This capability is one chief feature that sets the FSA Simulator apart from similar systems.

The creation of a construction and verification exercise is relatively simple. The author of the exercise, running the FSA Simulator as an application, first creates a correct FSA that recognizes the language described in the exercise and saves the FSA to a file. The author may also want to create a "starter" file that contains a partial FSA with which the students can begin constructing their solutions. These

FSA files must then be added into the “jar” archive in which the binary distribution of the Simulator is packaged. The exercise author must then create an HTML file containing a description of the language which the student needs to design an FSA to recognize. The author must also include the HTML tags needed to embed the FSA Simulator into the page as an applet. Between the HTML tags for the applet, the author needs to include the name of the FSA file containing the correct solution and (if desired) the name of a “starter” file to be displayed when the applet is started. Once these steps have been completed, the HTML file and the Simulator’s “jar” file need only be placed in a directory that is accessible to a web server and the exercise will be ready for use.

To complete an exercise, a student just needs to enter the Universal Resource Locator (URL) of the exercise’s HTML file into a Java-enabled web browser. The text of the exercise and the FSA Simulator applet will appear when the HTML file is loaded into the browser.

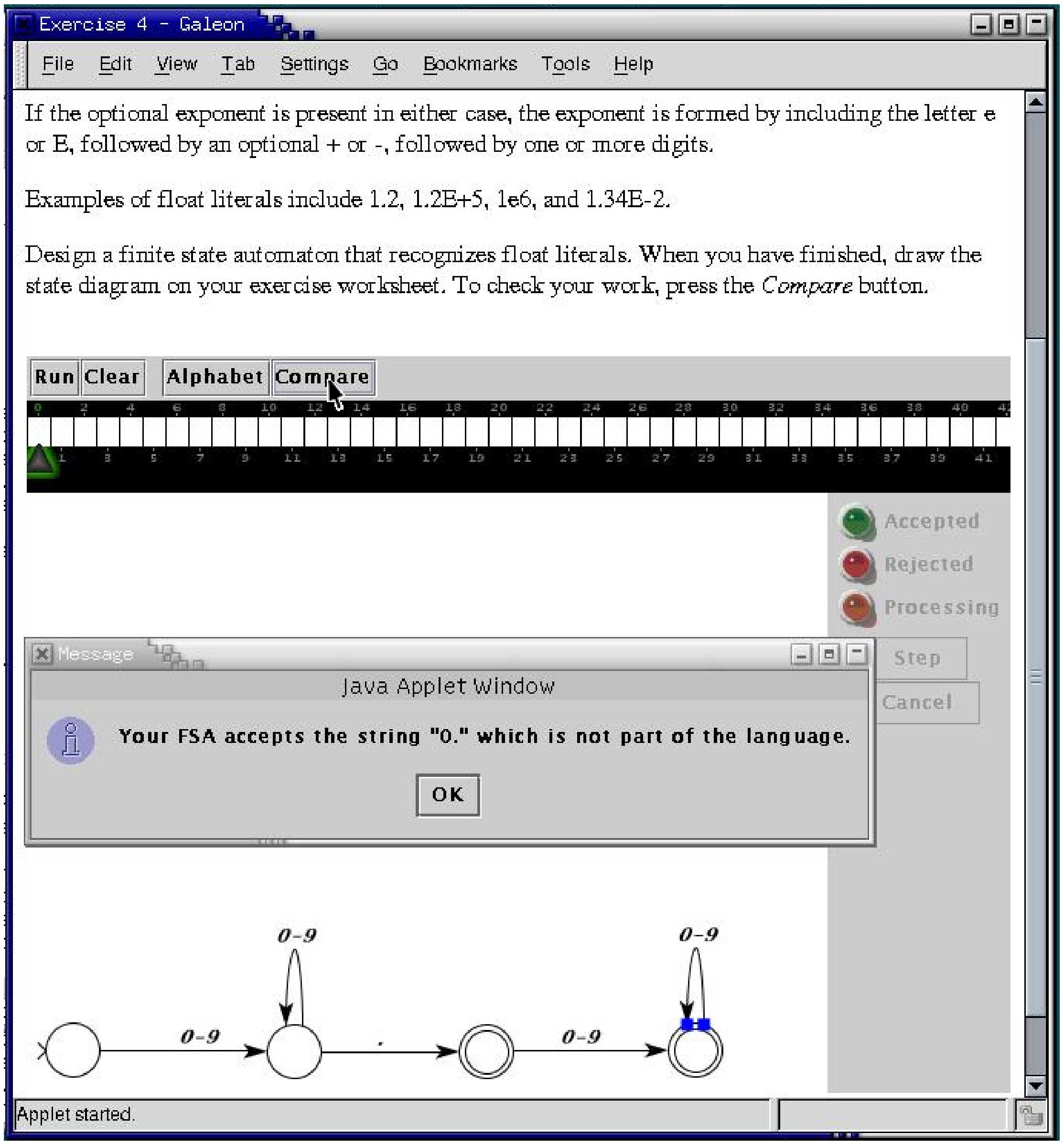
When the FSA Simulator applet is used in a construction exercise, its user interface remains the same as presented in previous sections describing the FSA Simulator with the exception that a *Compare* button is added to the toolbar. When the applet begins executing, it will load the correct FSA created and stored by the exercise author in the background. The “starter” file, if present, will be loaded and displayed in the state diagram panel.

At this point, the student attempts to construct an FSA that recognizes the language specified in the exercise using the FSA construction and modification tools described earlier. 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 3.10) or that it does not accept a string that is in the target language (see Figure 3.11). 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 3.12).

With the FSA comparison feature providing feedback, a high degree of active learning is involved, as students continue to work toward a correct solution with the guidance of the *Compare* feedback mechanism. It has been observed in laboratory exercises that without the feedback provided by the *Compare* feature, students often construct incomplete solutions. Since they do not have any way (short of a formal proof) to determine whether their solutions are correct or not, they often build something that seems to work and move on to the next exercise. Allowing students to see where their answers fall short gives them the motivation necessary to design a complete solution.

Nondeterminism

The FSA Simulator seamlessly supports the construction and execution of nondeterministic FSAs. Nondeterminism can be added by creating multiple outgoing transition arrows from the same state, labeled with the same symbol. Empty-string transitions can also be created by selecting the symbol ϵ from the alphabet for the



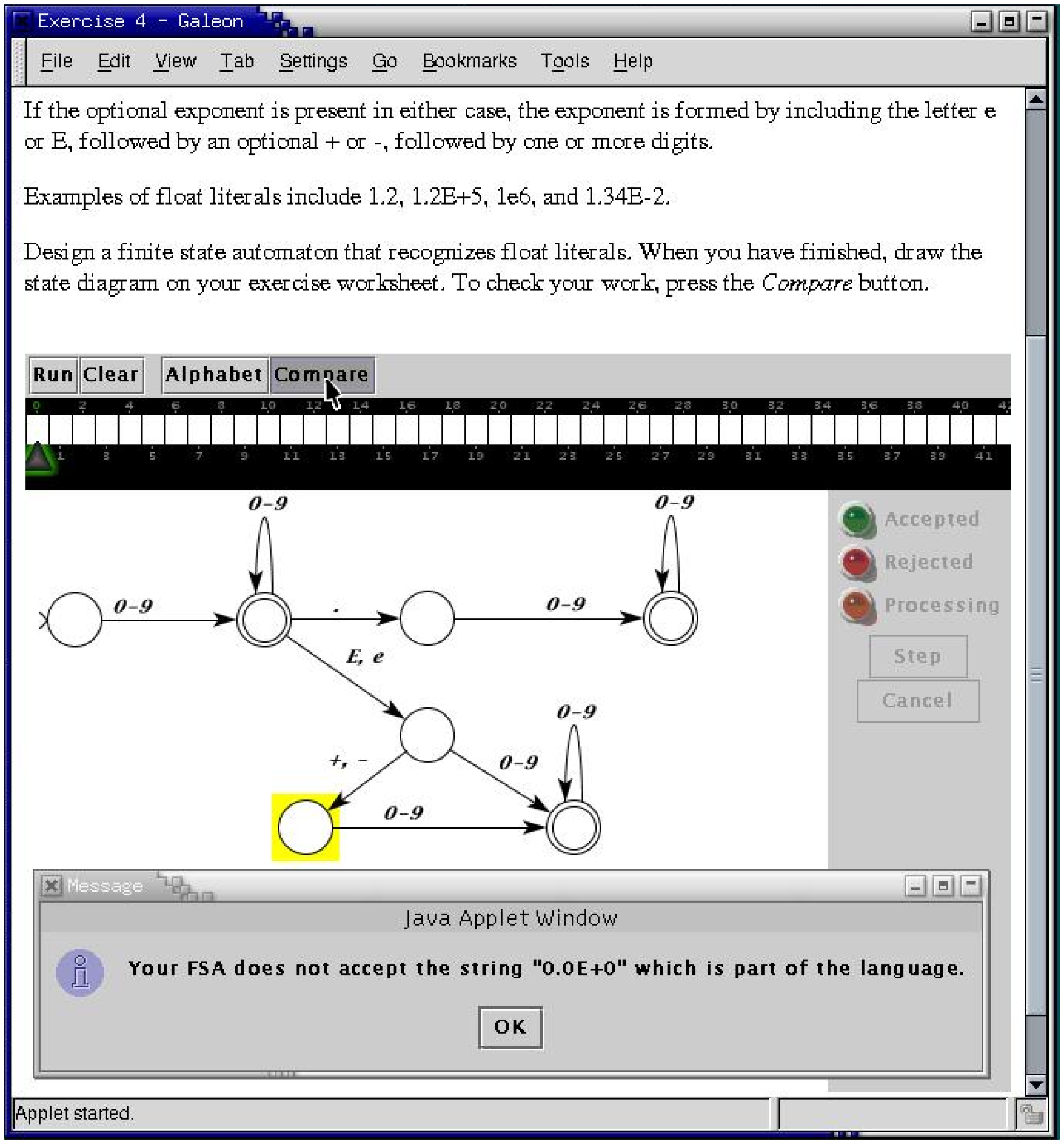
If the optional exponent is present in either case, the exponent is formed by including the letter e or E, followed by an optional + or -, followed by one or more digits.

Examples of float literals include 1.2, 1.2E+5, 1e6, and 1.34E-2.

Design a finite state automaton that recognizes float literals. When you have finished, draw the state diagram on your exercise worksheet. To check your work, press the *Compare* button.

Your FSA accepts the string "0." which is not part of the language.

Figure 3.10: FSA comparison message when the student's FSA accepts a string not in the target language



Exercise 4 - Galeon

If the optional exponent is present in either case, the exponent is formed by including the letter e or E, followed by an optional + or -, followed by one or more digits.

Examples of float literals include 1.2, 1.2E+5, 1e6, and 1.34E-2.

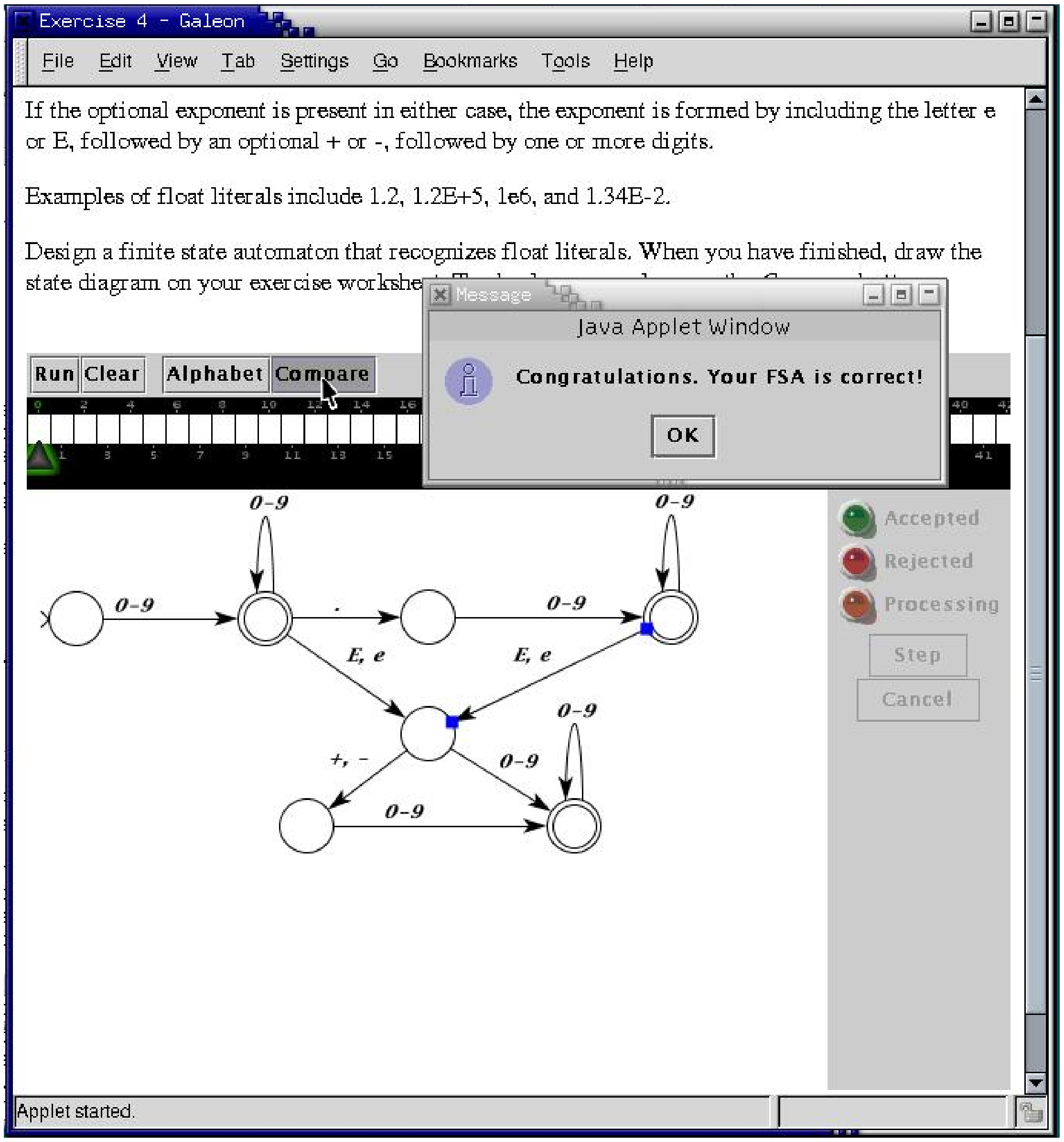
Design a finite state automaton that recognizes float literals. When you have finished, draw the state diagram on your exercise worksheet. To check your work, press the *Compare* button.

Java Applet Window

Your FSA does not accept the string "0.0E+0" which is part of the language.

Applet started.

Figure 3.11: FSA comparison message when the student's FSA does not accept some string in the target language



If the optional exponent is present in either case, the exponent is formed by including the letter e or E, followed by an optional + or -, followed by one or more digits.

Examples of float literals include 1.2, 1.2E+5, 1e6, and 1.34E-2.

Design a finite state automaton that recognizes float literals. When you have finished, draw the state diagram on your exercise workshe

Congratulations. Your FSA is correct!

Figure 3.12: FSA comparison message when the student's FSA correctly recognizes the target language

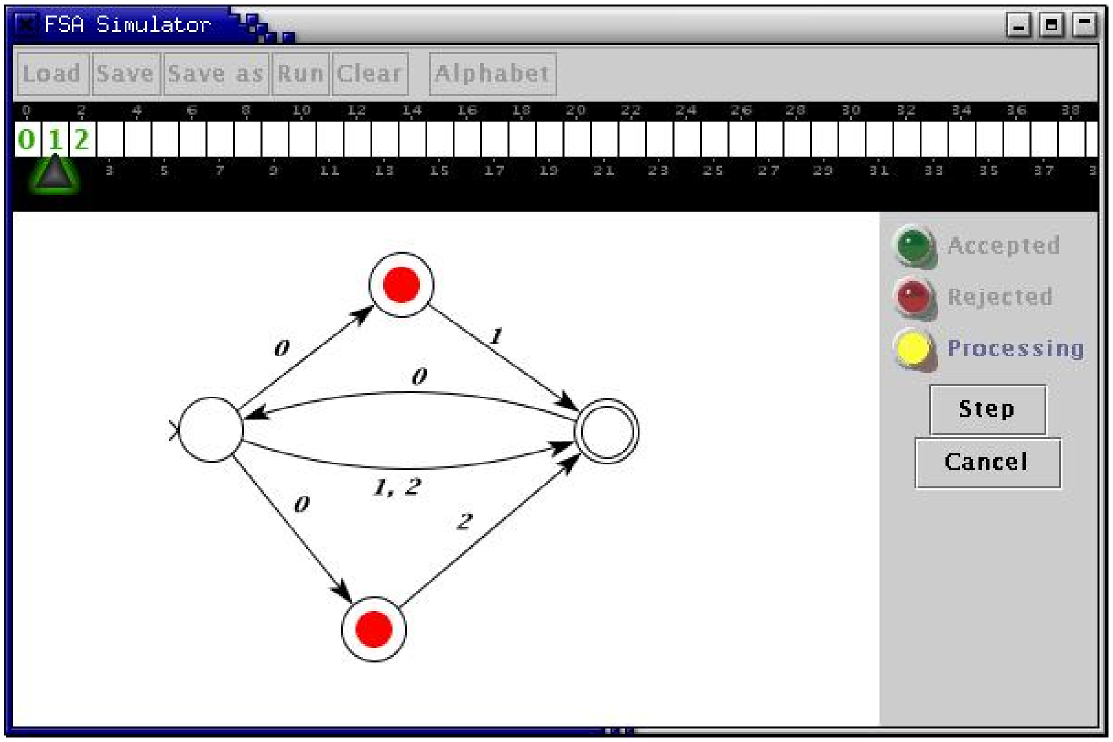


Figure 3.13: FSA Simulator after a nondeterministic transition

label of a transition arrow. When a nondeterministic FSA is processing a string, if multiple transitions exiting the current state are labeled with the current input symbol, the red circle will split into multiple copies, each of which will travel simultaneously over one of the transition arrows labeled with the current input symbol to the respective next states (see Figure 3.13). If the FSA enters a state with an outgoing empty-string transition, new red circles will split off of the red circle in that state and travel to all other states that are reachable using empty-string transitions.

Applications

Nearly every computer science or computer engineering curriculum requires that students learn some elements of the theory of computation. Regular languages and finite state automata are key concepts that these students must learn. The FSA Simulator can be used in many different ways to support the teaching and learning of these concepts. In this section, some of the ways that the Simulator can be—and has been—used will be discussed. For a discussion of possible future applications of the FSA Simulator see Chapter 6.

Classroom Demonstrations

An important use of the FSA Simulator is to aid instructors during lectures about FSAs. While it is possible to illustrate a lecture about FSAs using diagrams drawn on a whiteboard or overhead transparencies, the FSA Simulator can be used to make this task much easier and more interesting for the students. When demonstrating how a string is processed by an FSA on a whiteboard, the instructor must keep track of the details by constantly marking and erasing marks from a hand-drawn diagram. The FSA Simulator automatically displays the current state of the FSA and illustrates state transitions with a smooth animation.

Not only does use of the Simulator make lectures run better, it may also improve students' comprehension. Mayer and Anderson [49] found that animation of a concept combined with simultaneous narration improved student learning over other methods of teaching.

Supplementing Textbooks

The FSA Simulator can also be used as a supplement to an existing textbook. FSA construction exercises from the textbook can easily be adapted for the web by use of the applet version of the Simulator. Instructors can also provide additional exercises to supplement those from the textbook, giving students even more opportunities to improve their knowledge of regular languages and finite state automata. Through use of the comparison feature, students will be able to receive immediate feedback while doing these exercises. Such immediate feedback is impractical using traditional methods.

Grading Homework

One way to gauge the usefulness of a piece of software is to observe whether it is used for purposes that were not envisioned when it was originally designed. One such unanticipated application of the FSA Simulator that has emerged is in evaluating homework assignments and quizzes that are completed by hand (i.e., without access to the FSA Simulator). Homework assignments for course modules on regular languages, finite state automata, and regular expressions can be very difficult to grade. For example, asking students to construct an FSA that recognizes a nontrivial regular language can result in many different complex, but correct, solutions. It can be quite difficult for a human to discern whether or not a complex FSA really does recognize the language specified by a problem.

The FSA Simulator's ability to compare FSAs can come to the rescue in these situations. The comparison process can reveal if the students' solutions are indeed correct. If they are not, the comparison process will reveal a string that demonstrates places where the students' solutions diverge.

Hypertextbooks

Although many animation systems, including the FSA Simulator, are relatively simple to install and configure for classroom use, many instructors do not have time to locate, learn, set up, and integrate the different pieces of animation software that could be used in a theory of computation course. As mentioned in Chapter 1, hypertextbooks are one solution to this problem. All of the software and other materials used in a hypertextbook can be packaged onto a single CD-ROM that has no installation or configuration requirements other than a modern web browser and a recent version of the Java plug-in.

The members of the Webworks Laboratory are building an integrated hypertextbook for the theory of computation. The hypertextbook, entitled *Snapshots of the Theory of Computation*, will eventually contain material about all of the topics covered in a standard undergraduate theory course, from finite state automata to NP-completeness. Rather than being structured like a traditional textbook, *Snapshots* is a set of hyperlinked modules. Each module has material at three levels of difficulty. The difficulty levels are noted using the international trail marking system for ski areas (see Figure 3.14). Green circles indicate the easiest route through the hypertextbook. At this level, most of the explanations are at an intuitive level and incorporate liberal use of animation applets, such as the FSA Simulator. Blue squares denote more difficult intermediate material which still makes use of animation applets, but at a reduced level. Black diamonds mark advanced mathematical treatment of the material with much less use of animation applets (assuming that students visiting the material at this level do not need much in the way of visual, intuitive explanations).

Throughout *Snapshots*, Java applets are embedded within the text to animate

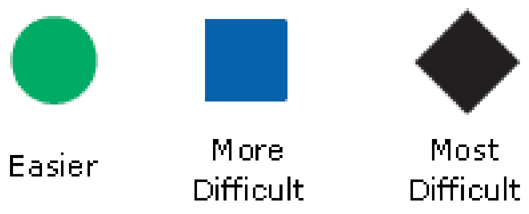
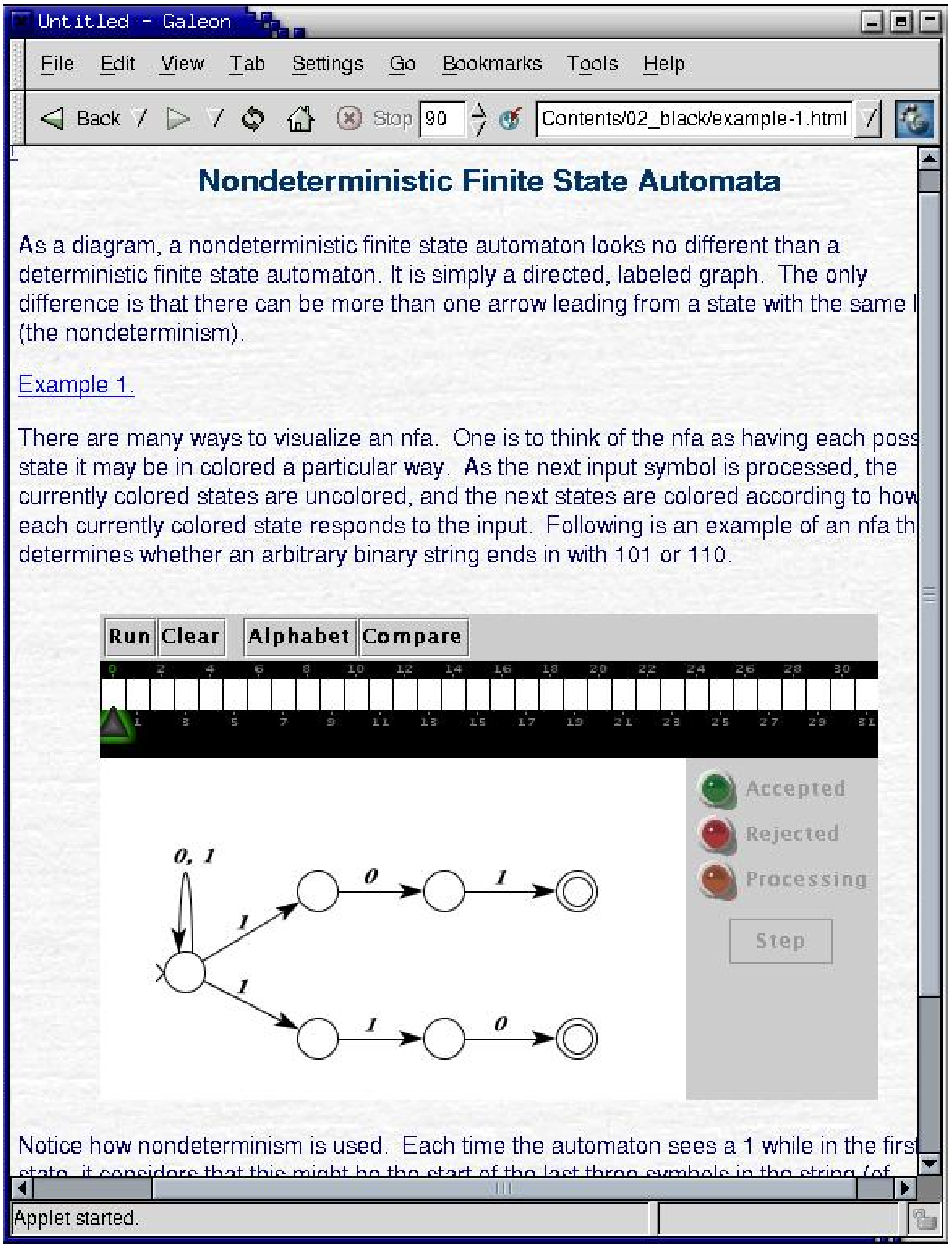


Figure 3.14: Ski Trail Marking System

concepts as they are discussed (see Figure 3.15). Sometimes the applets are simply used to demonstrate a concept. At other times, they provide exercises to give students an opportunity to apply what they have learned.

Conclusion

The FSA Simulator represents a new breed of active learning software for the theory of computation. It can be integrated into courses in many ways and provides active learning features that are not available in other FSA animators. Thus, the FSA Simulator provides an important base for the further development of active learning software for teaching the theory of computation. Information and expertise acquired during the development of the FSA Simulator will allow animations of other models of computation, such as pushdown automata and Turing machines, to be developed relatively quickly.



Nondeterministic Finite State Automata

As a diagram, a nondeterministic finite state automaton looks no different than a deterministic finite state automaton. It is simply a directed, labeled graph. The only difference is that there can be more than one arrow leading from a state with the same l (the nondeterminism).

Example 1.

There are many ways to visualize an nfa. One is to think of the nfa as having each poss state it may be in colored a particular way. As the next input symbol is processed, the currently colored states are uncolored, and the next states are colored according to how each currently colored state responds to the input. Following is an example of an nfa th determines whether an arbitrary binary string ends in with 101 or 110.

Notice how nondeterminism is used. Each time the automaton sees a 1 while in the firs

Figure 3.15: The FSA Simulator Applet Embedded in *Snapshots*

CHAPTER 4

FSA SIMULATOR INTERNALS

Development History

The FSA Simulator began as a “proof of concept” project for the WebLab Project (now the Webworks Laboratory) at Montana State University. The original scope of the project was merely to demonstrate the feasibility of using Java applets to animate concepts in the theory of computation for teaching and active learning. Creation of the first version of the Simulator also served as an opportunity for the author to sharpen his Java programming skills. When this initial version of the Simulator (see Figure 4.1) was completed, its potential was recognized and plans were made to redesign the software and add new features.

The second version of the FSA Simulator adopted many of the features found in animation packages described in the literature review while incorporating aspects found nowhere else. The architecture of the FSA Simulator was completely redesigned to be more flexible and powerful. Everything from the data file format to the graphical interface was enhanced and improved.

Even at this stage, the purpose of the FSA Simulator was limited to being a standard automaton simulation that allowed state machines to be created, modified, and executed, but nothing to support more advanced teaching and learning. Inspiration suddenly struck as the author prepared for comprehensive exams. It became evident that by applying the closure properties of regular languages, it would be possible to algorithmically compare the language of one FSA to the language of another FSA. That is, algorithms could be developed to determine whether two FSAs recognized the same language, and, if not, determine where the two languages differed. Thus, the FSA comparison feature (see page 51) was born. The FSA Simulator was transformed from a routine automaton simulator into a unique tool that could actively guide students through exercises toward a correct solution.

The creation of the FSA comparison feature opened the door for many new uses of the FSA Simulator (see Chapter 6). The remainder of this chapter will give a detailed description of the FSA Simulator's development and of the architecture that makes these exciting new capabilities possible.

Version 1

The first version of the FSA Simulator had many problems. Since development began shortly after the release of the first version of the Java programming language, the graphical interface was built using the relatively primitive graphics applications programming interface (API) available in Java 1.0's Abstract Windowing Toolkit (AWT). The limitations of the AWT resulted in poor-quality graphics for the user interface. The edges of figures in the state diagrams were ragged and the animation was slow and jerky. The slow execution speed of the first generation of Java virtual machines also contributed to the Simulator's performance problems.

The initial version of the FSA Simulator contained many deficiencies in its internal architecture. The code for displaying an FSA's state diagram was closely intertwined with the Simulator's internal model of the FSA. This made modification of the software difficult. Changes and bug fixes would often cause new problems. Addition of new features and alternative views of the FSA would have been a long, difficult process within the original design structure.

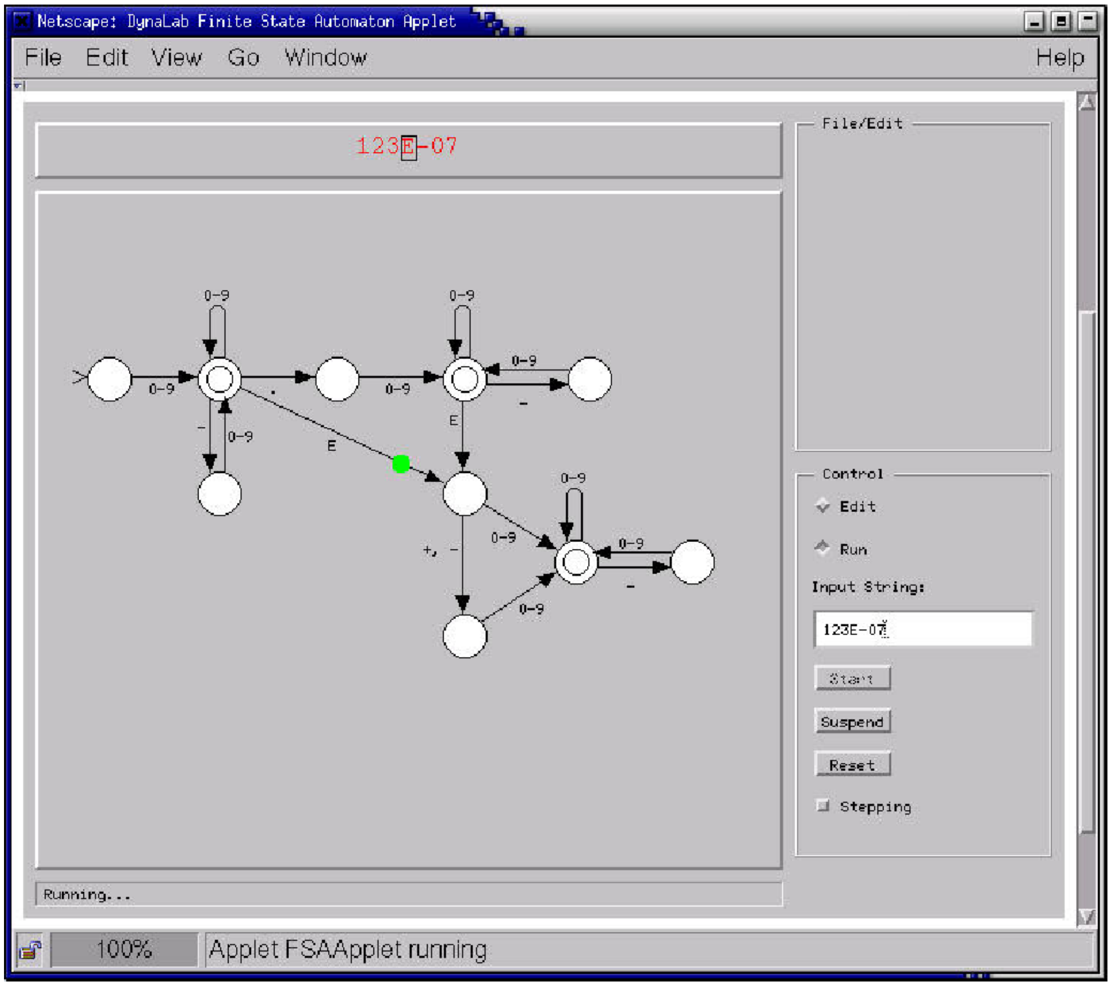


Figure 4.1: Version 1 of the FSA Simulator

The file format for saved FSAs had similar problems. The code for saving an FSA to a file was also mixed into the code for the internal FSA model. The files were saved as binary data that was specific to the Java virtual machine's representation of basic data structures. This limited the usefulness of saved FSAs. The files could only be used by Java programs and any changes to the way in which the Java virtual machine represented primitive data types would have prevented the files from being loaded by future versions of the FSA Simulator.

Further, although some support existed for nondeterminism in the first version of the FSA Simulator, it was quite primitive and unreliable. Attempts to execute a nondeterministic FSA often resulted in erratic behavior and could even crash the application.

Version 2

In the spirit of Fred Brooks' rule, "plan to throw one away" [18], the second version of the FSA Simulator was completely redesigned and rewritten in order to overcome the design flaws of the first version [15]. The major goals of the rewrite were to decouple the internal model of the FSA from its graphical and file format representations and to take advantage of new features available in Java 2. These changes were certain to make the FSA Simulator more robust and allow it to be integrated into a variety of teaching and learning resources more easily.

Separating the internal representation of the automaton from its graphical representation was a particularly important goal of the redesign. Besides making the source code easier to write and maintain, this change would also allow alternative graphical representations of an automaton (for example, as a table rather than a graph) to be developed. This objective was accomplished using a Model-View-Controller (MVC) [42] design. In the MVC paradigm, the internal objects that represent a structure (the model) are completely separate from the the objects that present a representation of them (the view). All communication between the model and the view is accomplished through a well-defined interface. In the case of the FSA Simulator, the internal model of the automaton sends out event messages as modifications occur. The graphical representation objects register to listen for these events and update their views as necessary. The graphical representation classes also act as controllers, converting mouse and keyboard events into modifications of the model.

This fundamental change resulted in a layered structure for the FSA Simulator (see Figure 4.2). At the center of these layers are the objects that make up the internal model of an FSA. All of the details of the FSA's structure are stored in these objects. Any changes to the FSA must be done through these objects. Other objects that need to be notified when some part of the FSA changes must register themselves as "listeners" to receive notification when changes occur. Thus, adding a new graphical representation is much simplified because modification of the internal data model is not required.

In its default configuration, the FSA Simulator consists of three main layers of objects. At the top is the GUI layer, which renders a graphical representation of the FSA. The next layer below it is the internal model. Under the internal model layer is a set of objects that maintains a Java Document Object Model (JDOM) [39] tree to be used if an FSA is loaded from or saved to a file. Between these three layers are relatively thin layers of adapter classes that translate events from one layer into actions in the next. Thus, GUI events such as mouse clicks and key presses are passed to objects which convert the events into method calls that modify the internal model. Changes to the internal model produce events that are passed back up to a translator object that converts them into commands that modify the rendering of the graphical representation. Internal model modifications are also passed down to the JDOM layer which causes the tree structure to be modified to reflect the modified structure of the FSA. When the user directs the program to save the current FSA to a file, the JDOM tree is able to write out the structure of the FSA to an XML file with a single method call.

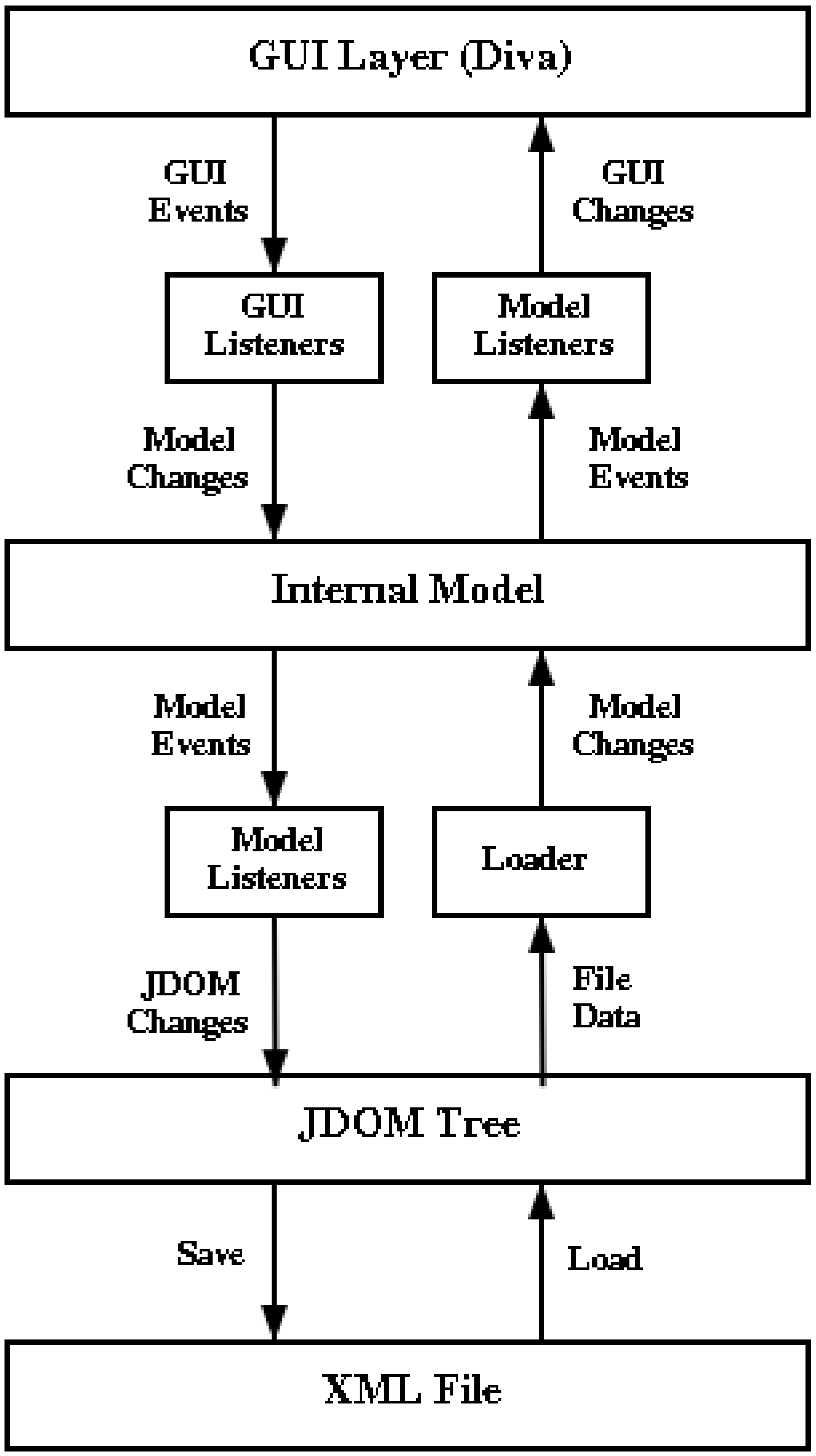


Figure 4.2: The Architecture of Version 2.

Internals

Graphical Representation

After evaluation of several different graphical frameworks, the Diva toolkit [31], developed at the University of California-Berkeley, was chosen for building the default graphical representation of the FSA Simulator. Diva takes advantage of the Java2D API that was added to Java 2 [2]. Java2D offers advanced graphics features such as antialiasing (which makes the state diagram appear sharper and smoother), automatic double buffering (which contributes to smoother animation), and timers (for scheduling animation events). The Diva toolkit also provides important graph display classes necessary for constructing the state diagrams used to display FSAs.

Although it is similar to other FSA simulators, the FSA Simulator's graphical interface has some features that distinguish it from other automaton simulators. Much attention was devoted to the user interface of the FSA Simulator to ensure that student learning is enhanced. The most obvious difference is in the animation of the execution of an automaton. Rather than having the graphical representation of an FSA switch instantaneously from one state to the next during a state transition, as described earlier, a solid, colored circle identifying the current state of the machine moves smoothly along the transition arrow from the current state to the next state, clearly showing which transition is followed during a state change. This feature was implemented based on observations that students are better able to follow state changes in any type of animation model when smooth motion is used [76]. In addition, the state diagrams that represent the FSAs were designed to look as much like traditional diagrams found in textbooks as possible. Further, all construction and modification of FSA state diagrams were designed to be done

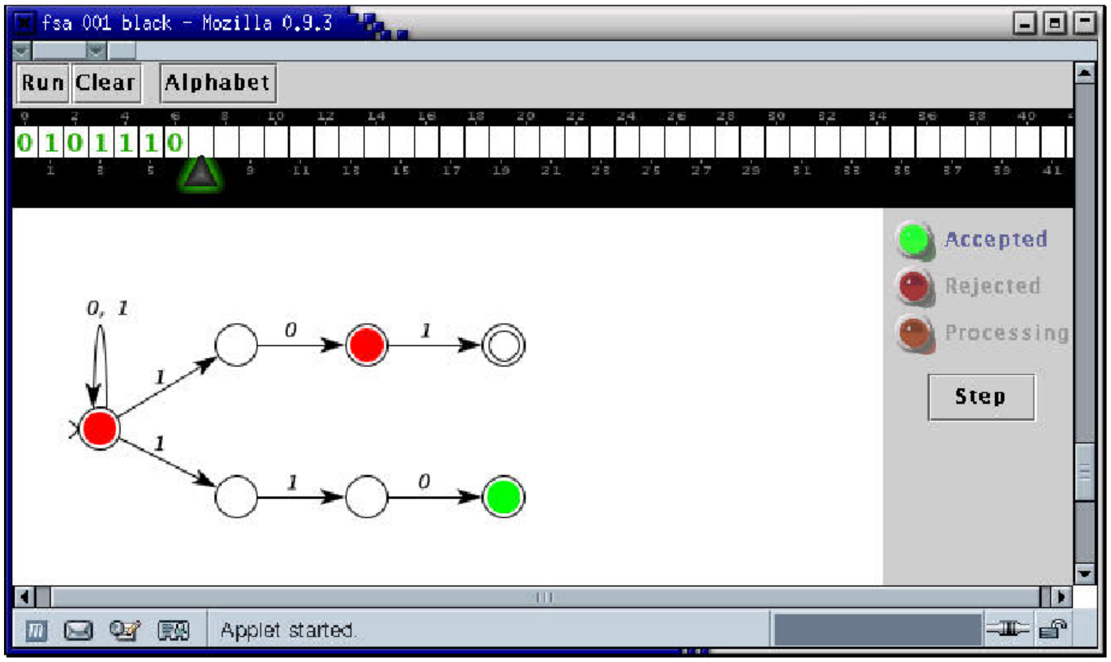


Figure 4.3: A nondeterministic FSA in Version 2 of the FSA Simulator

without requiring switching between different editing modes. Transitions with endpoints that are connected to the same pair of states were programmed to curve away from each other to prevent confusion. Also, both endpoints of the transition arrows were designed to allow them to be moved from one state to another.

Nondeterminism

Support for nondeterminism was greatly improved in the new version of the FSA Simulator. FSAs can be constructed with empty-string transitions as well as traditional nondeterministic transitions. Nondeterministic execution is represented by having multiple solid circles of the same color trace execution paths simultaneously on nondeterministic branches (see Figure 4.3). If any one of the currently active execution paths comes to a dead end, the colored circle representing it stops, turns gray, and then vanishes when the next step begins.

Nondeterminism is often a topic that students have difficulty understanding. It is also difficult to illustrate nondeterminism in a classroom using traditional static diagrams. The Simulator's method of animating nondeterminism can be especially helpful for explaining nondeterminism, particularly when explaining the standard proof by construction that there is an equivalent deterministic FSA for every nondeterministic FSA. Using the FSA Simulator, an instructor can more easily illustrate how a combination of states in the nondeterministic FSA can be represented with a single state in the new deterministic FSA.

File Format

One significant (and unique as far as the author knows) improvement in the FSA animator is the file format used for storing FSAs. To overcome the problems inherent in the original binary file format, a new format using Extensible Markup Language (XML) [81] was developed. XML is a standard developed by the World Wide Web Consortium (W3C) for representing structured data in plain text files. The structure of an XML file looks similar to the HTML files that are used for web pages. The data is structured as a series of nested opening and closing "tags". Data can be included as named attributes embedded within the opening tag, or as text between the opening and closing tag.

There are many advantages to using XML for data files. Since XML files have a standard format, there are XML parsers for nearly every programming language in use today. XML files can be transferred from platform to platform and from application to application without modification. Since they are plain text, XML files can be created, edited, and read by humans using only a text editor, yet computer applications can easily manipulate XML data as well.

The creation of an XML file format for the FSA Simulator separated the structure of the FSA data file from the FSA Simulator's implementation details. Developers of other applications who are familiar with the structure of the FSA Simulator's XML data files will be able to make use of the data without ever having to see the FSA Simulator's source code. There is no part of the file format that is specific to the features of the FSA Simulator. Any data that are outside of the essential structure of an FSA, for example, the positions of the circles that represent states in the state diagram panel, are encoded in generic "property" tags that can be used for application-specific data. Applications that are interested only in the FSA can safely ignore "property" tags without losing any vital information about the FSA. See Appendix A for the FSA file definition and an example FSA file.

Using the new MVC architecture and the JDOM XML library [39], a new file framework was created that updates information as the structure of the FSA is changed. At any point during the execution of the program, the structure of the current FSA is ready to be written to disk as an XML file.

With the new architecture in place, adding new output formats is also much simplified and does not require modification of other parts of the system. It is now possible to add code that can save FSA descriptions to destinations other than files on a local disk. For example, FSA descriptions could be saved to XML-enabled databases or to servers on a network using the various web services protocols which are now being developed.

Alphabets

In order to reinforce the concept of an alphabet, users must now specify a specific alphabet for every automaton built. All transitions and input strings are restricted to symbols in the alphabet.

FSA Comparison

An interesting and unique property of FSAs is that it is possible to detect whether two of them recognize the same language. This is possible because regular languages are closed under complementation and intersection [44]. An algorithm exists which, when given an FSA as input, will construct a new FSA that recognizes the complement of the language of the original FSA. There is also an algorithm for constructing an FSA that recognizes the intersection of the languages recognized by a given pair of FSAs [37].

To determine if two FSAs, FSA_1 and FSA_2, recognize the same language, one can first construct two new FSAs that recognize the complements of the languages of FSA_1 and FSA_2, $\overline{FSA}_1$ and $\overline{FSA}_2$ respectively. Applying the intersection algorithm to FSA_1 and $\overline{FSA}_2$ will produce $FSA_{1\cap\overline{2}}$ which recognizes the set of strings that are in the language of FSA_1, but not in the language of FSA_2. Going the other direction, applying the intersection algorithm to FSA_2 and $\overline{FSA}_1$ will produce $FSA_{\overline{1}\cap 2}$ which recognizes the set of strings that are in the language of FSA_2, but not in the language of FSA_1. Notice that if $FSA_{1\cap\overline{2}}$ accepts any strings, these strings are also accepted by FSA_1 but not FSA_2 (therefore, FSA_1 and FSA_2 recognize different languages). A similar statement can be made about $FSA_{\overline{1}\cap 2}$. To find whether either $FSA_{1\cap\overline{2}}$ or $FSA_{\overline{1}\cap 2}$ accepts any strings, it is simply a matter of doing a depth-first search along the transitions from the start state in both automata, looking for a final state. If a final state is found in one of these searches, that automaton will accept some string. If neither $FSA_{1\cap\overline{2}}$ nor $FSA_{\overline{1}\cap 2}$ accept any strings, then the two original automata, FSA_1 and FSA_2, recognize the same language.

The second version of the FSA Simulator makes use of these algorithms to provide exercises that give feedback to students. A target FSA, FSA_T, constructed by the author of the exercise, is loaded into the Simulator in the background when the exercise applet is started. The student is instructed to build an FSA that recognizes a given language (the same language that FSA_T recognizes). When the student thinks the task has been accomplished, the *Compare* button can be clicked, which causes the Simulator to compare the student's FSA, FSA_S, to FSA_T.

At this point, using the algorithms described above, the Simulator first constructs $\overline{FSA_T}$ and $\overline{FSA_S}$, which recognize the complementary languages of FSA_T and FSA_S respectively. Then, to check if FSA_S accepts any strings that the FSA_T does not, $FSA_{S \cap \overline{T}}$ is constructed. A depth-first search is then performed on $FSA_{S \cap \overline{T}}$ to determine if it accepts any strings. As the search is performed, a string is built from symbols associated with each transition that is traversed. If a final state is found, the string that was constructed during the search will be one of the strings in the language of $FSA_{S \cap \overline{T}}$. In this case, a dialog box pops up telling the student that the constructed FSA (FSA_S) accepts a string that is not in the language given at the beginning of the exercise and displays the example string that was constructed.

If FSA_S does not accept any strings that FSA_T does not accept, the Simulator then checks for strings that are not accepted by FSA_S but are accepted by FSA_T by creating $FSA_{\overline{S} \cap T}$ and searching for a final state as described for $FSA_{S \cap \overline{T}}$. If a string is found that $FSA_{\overline{S} \cap T}$ accepts, a dialog box pops up telling the student that FSA_S does not accept a string that is in the language described in the exercise and displays the example string that was constructed during the search.

If it is found that no strings are accepted by either $FSA_{S \cap \overline{T}}$ or $FSA_{\overline{S} \cap T}$, a dialog box informs the student that the constructed FSA is correct.

CHAPTER 5

EVALUATION

Goals

Evaluation is an important issue to consider when developing interactive visualization tools for education. It is pointless to use a tool that does not have a positive effect on students' learning. The question usually asked when discussing the effectiveness of a new teaching and learning resource is, "Does it work?" This question is too general to be useful. There are many ways in which visualization software can provide benefits in a learning environment. More useful specific questions include:

1. Does the resource, when used by itself as an aid in teaching and learning the specific topic for which it was created, work *better than* traditional methods?

   As mentioned on page 35, a few studies have found that students using visualizations have not performed better than those who did not have access to the visualizations [27, 69]. However, other studies have shown that students do learn better if they are actively involved in creating the visualization (rather than passively watching a visualization) [77]. More work needs to be done to determine whether active learning visualizations, in which students participate in setting up and directing the visualization, encourage learning more than activities that do not involve visualization systems.

2. Does the resource, when used by itself as an aid for teaching and learning, work *at least as well as* traditional methods?

This question is important, but often overlooked in the literature. As distance and self-study learning become more prevalent, resources will need to be created to assist students in environments where traditional teaching methods can not be used.

3. Do students learn *better* than with traditional methods when visualizations are integrated seamlessly into a comprehensive, computer-based learning environment that is the primary resource for a course?

   This question has not yet been addressed in the literature as no such integrated resources currently exist. Although many visualization systems are available as stand-alone systems, some of them very good, very few instructors or students actually use them. This phenomenon may be caused by a number of factors including [15]:

   - Downloading, installing, and maintaining visualization software often requires too much time from instructors, who are often overworked due to a lack of faculty and large enrollments.
   - Many visualization packages are dependent on a specific platform. This will automatically reduce the number of potential users.
   - Visualization software is often designed as a stand-alone tool to teach a specific concept. In order to provide a complete set of visualizations for a course, instructors must pull bits and pieces together from multiple sources. As mentioned above, many instructors do not have time to devote to such tasks.

   Many of these difficulties would be eliminated if a comprehensive, integrated teaching and learning resource such as a “hypertextbook” [15] were available.

The work presented here has been designed to be included in a "hypertextbook" on the theory of computation. When that occurs, formal evaluations can be done to answer this question.

4. Do students learn *at least as well* as with traditional methods when given a comprehensive, computer-based learning resource that seamlessly integrates active learning visualizations?

   Once again, this question is important for situations such as distance learning and self-study when faculty with the proper expertise are not available to teach a traditional course.

5. Do students *perceive* that they are receiving a better learning experience when active learning visualizations are used to illustrate important concepts? Are students more *excited*, more *motivated*, and more *eager* to learn?

   These questions may initially seem worthless if students are not actually improving their understanding of a topic. However, increases in students' motivation and enjoyment of computer science are also benefits that may indirectly lead to improved learning. If active learning visualizations make difficult topics more fun and interesting, students may be less intimidated and willing to continue to try to understand the topic despite its difficulty. Also, if such systems do inspire students to study computer science, they may also encourage more underrepresented groups (e.g., women) to enter the field.

Each of the above questions also contain many dimensions that should be explored. For example, how does the length and frequency of use of visualization software affect learning? How does use of visualization software affect short-term and long-term retention? How readily do students turn to active learning visualizations rather than traditional static resources, such as textbooks, when they are studying outside of the classroom? The evaluation of active learning visualizations is a rich but virtually unexplored area of research that is important for the future of computer science education.

Preliminary Evaluations

The primary objective of this dissertation was to determine the feasibility of creating cross-platform active learning software as part of a comprehensive resource for teaching the theory of computation via the World Wide Web. This objective has been accomplished. A secondary objective was to begin an evaluation of the effects of this software on student learning, as discussed in the previous section. Since evaluation was a secondary objective, the author has not been able to devote as much time to it as is necessary to produce many definite conclusions. Nevertheless, two preliminary evaluations have been completed that will be used as a basis for further research by the author.

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

Subjects

The first experiment was performed in a first-year computer science course, CS 221, at Montana State University. This course is mainly taken by computer science and computer engineering students during the second semester of their first year, after having taken an introductory programming class [54].

Before the students began the lab assignment, they were asked to fill out a form to provide some demographic information about themselves, so that the test and control groups could be compared for significant differences. The form asked for each student's age, sex, major, year in college, approximate grade point average, and standardized test scores. Students were also asked whether they were familiar with terms related to the subject of the lab, such as *finite state automaton* and *regular expression*, to determine whether they had had any previous exposure to the topic. To protect the students' privacy, the information provided in these forms was associated only with a number assigned to the student, not with a name.

Fifty-two students participated in the lab. The average age of the students was 20. 75% were freshmen, 15% sophomores, 8% juniors, and 2% seniors. 63% of the students were computer science majors, 35% computer engineering, and 2% (1 student) architecture. Males made up 98% of the class, which is much higher than national averages [28]. Most of the students were white, although a few non-white students participated. Overall the students tended to be white males near the traditional age of first-year college students.

Design

There were three sections of the lab, each lasting two hours. The students were divided into test and control groups by assigning each section to be in either the test or control group. The first and third sections, a total of 28 students, made up the test group. The second section, with 24 students, was the control group. All of the female students were in the test group.

In the test group, 71% of the students were computer science majors, 25% were computer engineering majors, and one student (4%) was studying architecture. The control group was almost evenly divided between computer science (54%) and computer engineering (46%) majors.

The average age in the test group was 20.29 with a standard deviation of 4.13. The average age in the control group was 19.75 with a standard deviation of 2.71. The difference was determined to not be statistically significant using a t-test and Student's Distribution ($\alpha = 0.05$) [17].

Although some of the students reported ACT or SAT scores, many students were not able to report their scores, so that data could not be used to compare the two groups.

The students' reported grade point averages were very similar. They were asked to choose a GPA range that best described their GPA. The ranges from lowest to highest were "less than 2.0," "2.0-2.5," "2.5-3.0," "3.0-3.5," and "3.5-4.0". To compare the GPAs of the two groups, each range was assigned a number: "less than 2.0" was assigned 1, "2.0-2.5" was assigned 2, and so forth. The average of these values for the test group was 3.67 (representing GPAs in the upper part of the range 2.5-3.0) with a standard deviation of 1.04. The average for the control group was 3.74 (also representing GPAs in the upper part of the range 2.5-3.0) with a standard deviation of 0.81. The difference was not statistically significant ($\alpha = 0.05$).

Six of the students in the test group and three in the control group reported having been exposed to regular language topics previously.

Overall, the demographics of the two groups appear to be equivalent. A more controlled way of choosing test and control groups would be preferable, but was not practical.

Treatments

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 computation hypertextbook [15]. 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. Exercise 1 asked basic questions about a finite state automaton (FSA) represented as a state diagram. The students were asked to identify the start state, final states, and alphabet of the FSA. They were also asked to list the sequence of states the FSA would enter while processing a particular string, and to determine whether the FSA would accept or reject 4 different strings. Exercise 2 asked the students to construct a state diagram of an FSA from its formal description. In Exercise 3, the students were asked to construct an FSA that recognized identifiers for a programming language. Finally, in Exercise 4, the students were asked to construct an FSA for recognizing floating point literals.

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. Problem 1 of the test was similar to Exercise 1, but two additional questions were added: The students were also asked to identify a string that would be accepted and a string that would be rejected by the FSA. For Problem 2, the students were asked to write a formal description of the FSA depicted in Problem 1. This was a bad idea, since the students did not have access to a description of the formal definition. The answers to Problem 2 were not graded. Problems 3-5 asked the students to draw state diagrams for FSAs which recognized specific languages of binary strings. For Problem 3, the language was strings that start with 1 and end with 0. The language for Problem 4 was strings containing at least three 1s. Problem 5's language was strings that do not contain the substring 110.

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. If students were not done with the tutorial within 80 minutes, they were asked to stop where they were and take the test during the last 30 minutes.

Observations

The students in the test group spent much more time on the lab than the students in the control group. It was observed that some of the test group's students encountered difficulties while learning the FSA Simulator's user interface. Students in the test group spent more time working on the exercises than their classmates in the control group. This was attributed to the test group's need to learn how to use the applet, some bugs in the applet that forced some students to restart in the middle of an exercise, and their ability to receive feedback about their solutions, which encouraged them to keep working on a solution until it was correct. Therefore, many of the students in the test group did not have sufficient time to work through all of the exercises. Nearly a third of them (8 of 28) were not able to begin the last exercise. The control group, on the other hand, completed the exercises (although not always correctly) and the test much more quickly. All of the students in the control group completed the lab in less than 90 minutes.

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 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.

Some of the students in the test group tended to build FSAs with a large number of states when using the comparison feature. Rather than reconsidering the design of their FSAs, some students would keep adding more states to handle strings in the target language that their FSA did not accept. (It may help to discourage such behavior by allowing the exercise author to specify a maximum number of states that can be created or a maximum number of times that the comparison button can be used.)

Question	**Group**	**Average Score**	**Standard Deviation**	**One-tailed t-value** ($\alpha = 0.05$)
Exercise 1	Test	5.1	0.994	-1.123
	Control	5.5	1.504	
Problem 1	Test	7.9	1.571	-1.449
	Control	8.5	1.641	

Table 5.1: Experiment 1 Results for Exercise 1 and Problem 1

Question	**Group**	**Number Correct**	**Percent Correct**	**Fisher Test p-value** ($\alpha = 0.05$) $\star$**-significant result**
Exercise 2	Test	16 of 28	57%	0.1281
	Control	9 of 24	38%	
Exercise 3	Test	16 of 28	57%	$3.425 \times 10^{-5}\star$
	Control	1 of 24	4%	
Exercise 4	Test	10 of 28	36%	$8.295 \times 10^{-4}\star$
	Control	0 of 24	0%	
Problem 3	Test	8 of 28	29%	0.5111
	Control	6 of 24	25%	
Problem 4	Test	15 of 28	54%	0.6258
	Control	13 of 24	54%	
Problem 5	Test	1 of 28	4%	0.4413
	Control	2 of 24	8%	

Table 5.2: Experiment 1 Results for Exercises 2-4 and Problems 3-5

Measures

Each group's scores on the exercises and problems are summarized in Tables 5.1 and 5.2 and in Figures 5.1, 5.2, 5.3, and 5.4. Although the students in the control group tended to do better on Exercise 1, the test group was much more successful than the control group at completing Exercises 2-4, with a significant difference on Exercises 3 and 4.

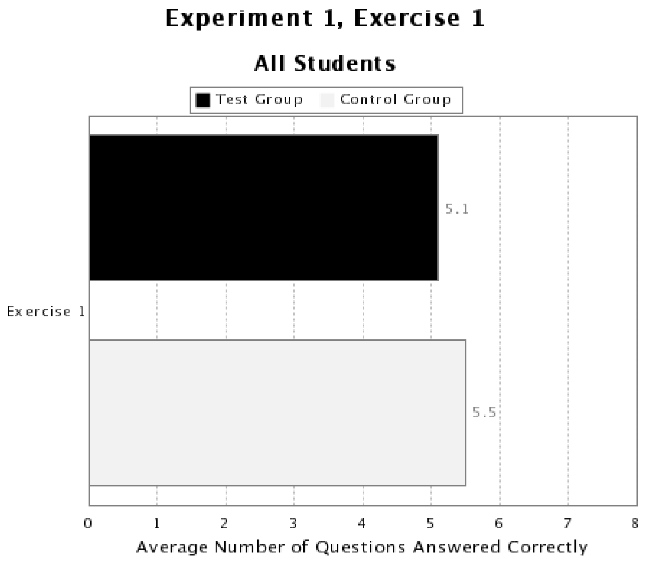


Figure 5.1: Experiment 1 Results for Exercise 1

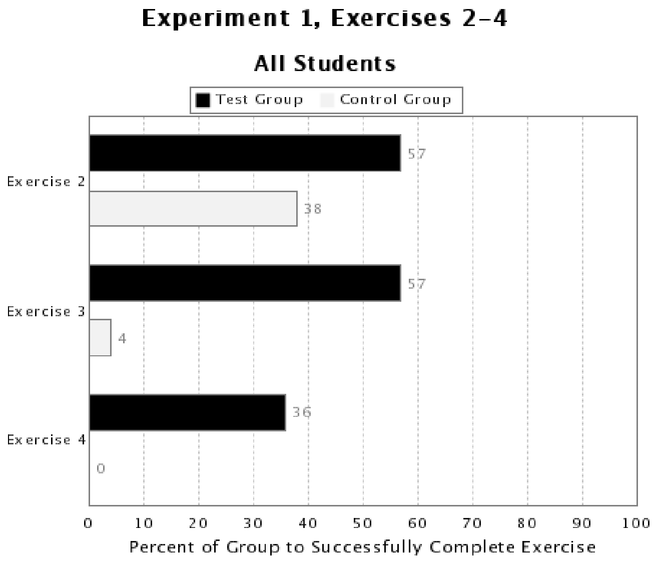


Figure 5.2: Experiment 1 Results for Exercises 2-4

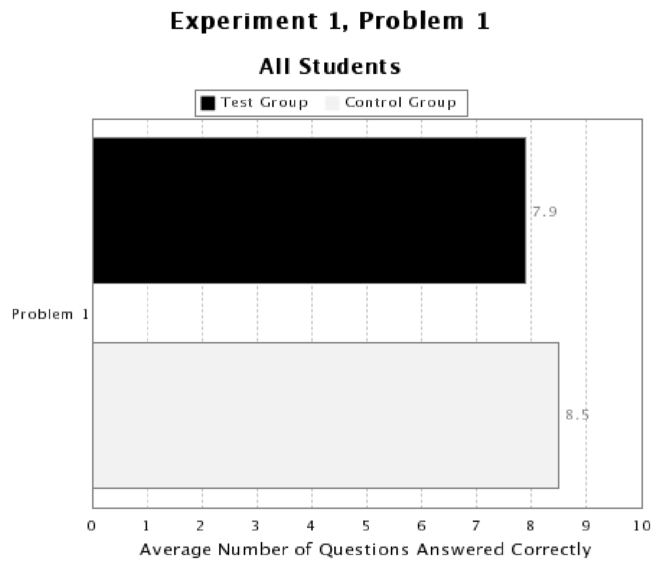


Figure 5.3: Experiment 1 Results for Problem 1

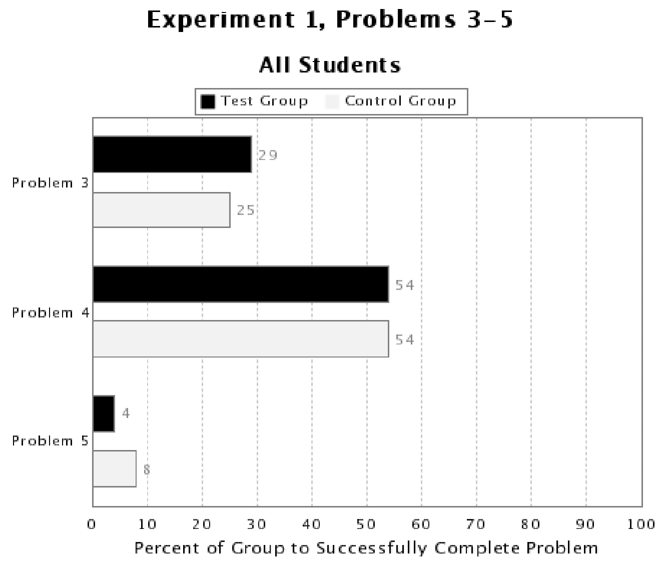


Figure 5.4: Experiment 1 Results for Problems 3-5

Analysis

Exercise Results The original goal of the comparison feature was to guide students to the successful completion of exercises (see page 51). The data from this experiment clearly demonstrates that the comparison feature 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. In contrast, the percentage of successful students in the test group started out high in Exercise 2, remained the same for Exercise 3, and dropped a relatively small amount (compared to the control group) on Exercise 4. The drop on Exercise 4 may be explained, in part, by time limitations. As mentioned earlier, students who used the FSA Simulator applet spent more time working on the exercises than those who did not. Many were not able to complete all of the exercises within the 110-minute limit. The test group's success rate on Exercise 4 might have been higher if the students could have worked on the exercises for a longer period of time.

Although the success rates for the test group were higher than the control group for Exercises 2-4, they were lower than original expectations. It may be that more than 57% of the students were able to complete the exercises, but some of the students may have made mistakes when transferring the state diagram from the computer screen to their worksheets. In the future, evaluations should use a completely electronic system for the test group to prevent such problems.

Test Results The test results of Experiment 1 were not as encouraging as the exercise results. The control group continued to have an edge on the identification portions of Problem 1 and the results for the two groups on Problems 3-5 were essentially the same. The advantages that the test group possessed for the exercises did not carry over when the extra support was removed. On the other hand, use of the FSA Simulator applet did not appear to be a disadvantage either.

Experiment 2

Subjects

The experiment lab was performed in a second-year computer science course, CS 223, at Montana State University. This course is usually taken by computer science majors during the second semester of their second year. Some computer engineering students also take this class as a professional elective [54].

Forty-four students participated in the lab. The average age of the students was 22 (standard deviation 3.5). 45% were sophomores, 27% juniors, 25% seniors and one student (2%) was pursuing a second degree. 86% of the students were computer science majors, 9% computer engineering, and 5% (2 students) were from other majors. Females made up 14% of the class. Most of the students were white, although a few non-white students participated.

Design

There were two sections in this lab, each lasting two hours. The students were divided into test and control groups by assigning each section to be either test or control. The first section, 17 students, made up the control group. The second section, with 27 students, was the test group. As in the first experiment, all of the female students were in the test group.

In the test group, 85% of the students were computer science majors, 7% were computer engineering majors, and two students (7%) were from other majors. The control group was 88% computer science and 12% computer engineering.

The average age in the test group was 21.8 with a standard deviation of 2.3. The average age in the control group was 22.6 with a standard deviation of 4.9. The difference was not statistically significant ($\alpha = 0.05$). As in the first experiment, not all students reported standardized test scores, so they were not used.

The students' reported GPA range was evaluated as described for Experiment 1. The average value for the test group was 3.78 (representing a GPA in the upper part of the range 2.5-3.0) with a standard deviation of 1.05. The average for the control group was 4.11 (representing a GPA in the lower part of the range 3.0-3.5) with a standard deviation of 0.99. The difference was not statistically significant ($\alpha = 0.05$).

Ten of the students in this lab (37% of the test group and 59% of the control group) reported having been exposed to regular languages topics previously. Many of the students were taking or had taken the undergraduate theory of computation course, CS 350. The difference in proportion is not significant at the $\alpha = 0.05$ level, but it appears that the control group may have an advantage over the test group.

Treatments

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 (see page 84) 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.

Measures

Each group's scores on the exercises and problems are summarized in Tables 5.3 and 5.4 and in Figures 5.5, 5.6, 5.7, and 5.8. 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.

Analysis

Exercises The results for the exercises in Experiment 2 were similar to those for Experiment 1. As expected, given their greater experience with the theory of computation and computer science in general, the students in both groups had higher scores than the students in Experiment 1. The test group had a slight, but not significant, edge over the control group on Exercise 1, a larger but still not significant advantage in Exercise 2 and significant advantages for Exercises 3-5. The control group's performance dropped at a steady rate as the difficulty of the exercises increased, while the test group's scores dropped somewhat between 2 and 3, but not drastically. The test group's scores actually increased slightly from 3 to 4. Although not quite as dramatic as in Experiment 1, the test group clearly had an advantage over the control group for the exercises.

Question	Group	Average Score	Standard Deviation	One-tailed t-value ($\alpha = 0.05$)
Exercise 1	Test	6.7	0.961	0.785
	Control	6.4	1.176	
Problem 1	Test	8.8	0.934	1.431
	Control	8.4	0.966	

Table 5.3: Experiment 2 Results for Exercise 1 and Problem 1

Question	Group	Number Correct	Percent Correct	Fisher Test p-value ($\alpha = 0.05$) $\star$-significant result
Exercise 2	Test	20 of 27	74%	0.1331
	Control	9 of 17	53%	
Exercise 3	Test	17 of 27	63%	0.0692
	Control	6 of 17	35%	
Exercise 4	Test	18 of 27	66%	0.0017$\star$
	Control	3 of 17	18%	
Problem 3	Test	16 of 27	59%	0.75
	Control	11 of 17	65%	
Problem 4	Test	20 of 27	74%	0.0683
	Control	8 of 17	47%	
Problem 5	Test	2 of 27	7%	0.371
	Control	0 of 17	0%	

Table 5.4: Experiment 2 Results for Exercises 2-4 and Problems 3-5

Test As in the exercises, the students in Experiment 2 scored higher on most of the problems on the test, with the exception of Problem 5, which was the most difficult. On Problems 1, 3, and 5, there were insignificant differences between the two groups. However, the test group did much better than the control group on Problem 4, although the difference was not quite large enough to be considered statistically significant. This suggests 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.

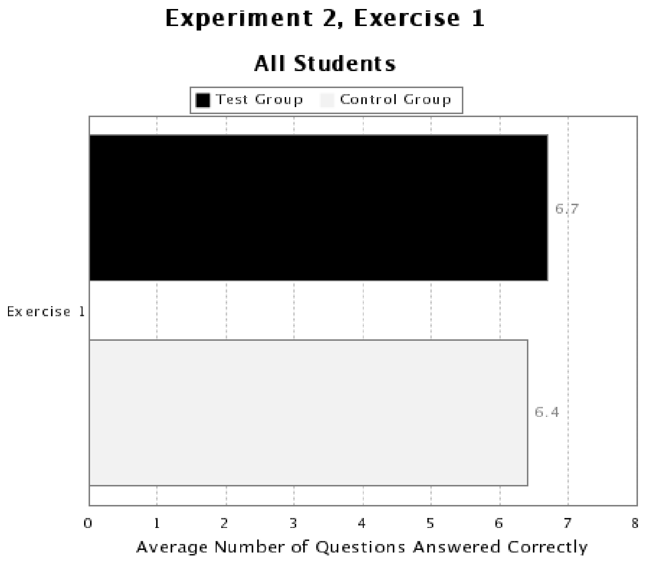


Figure 5.5: Experiment 2 Results for Exercise 1

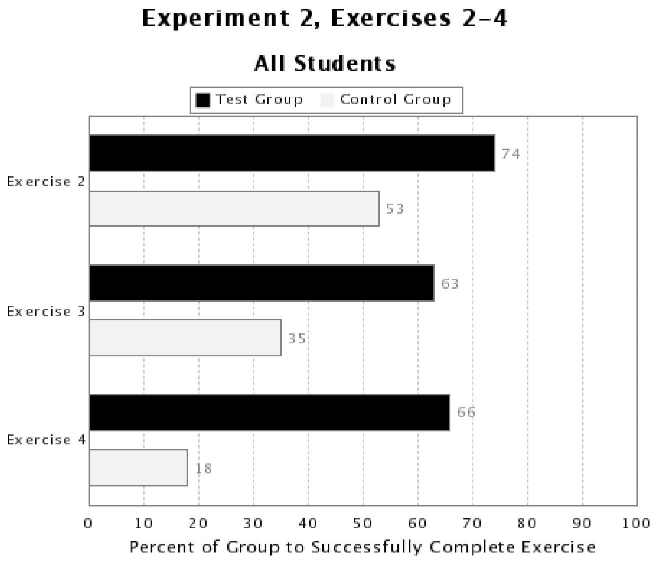


Figure 5.6: Experiment 2 Results for Exercises 2-4

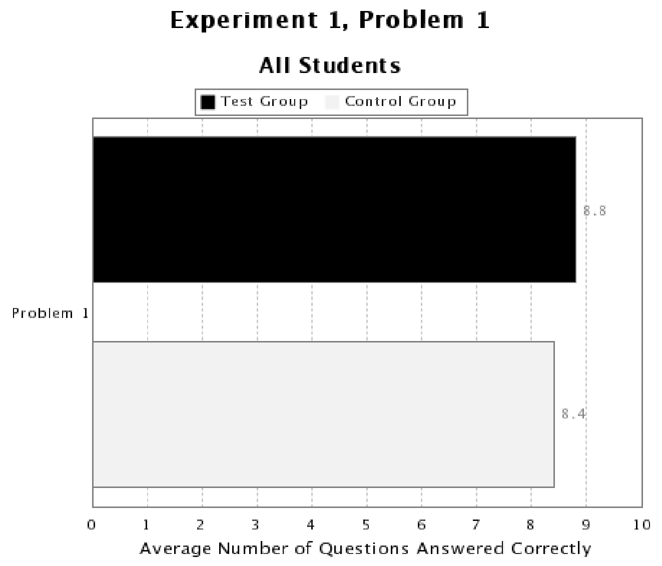


Figure 5.7: Experiment 2 Results for Problem 1

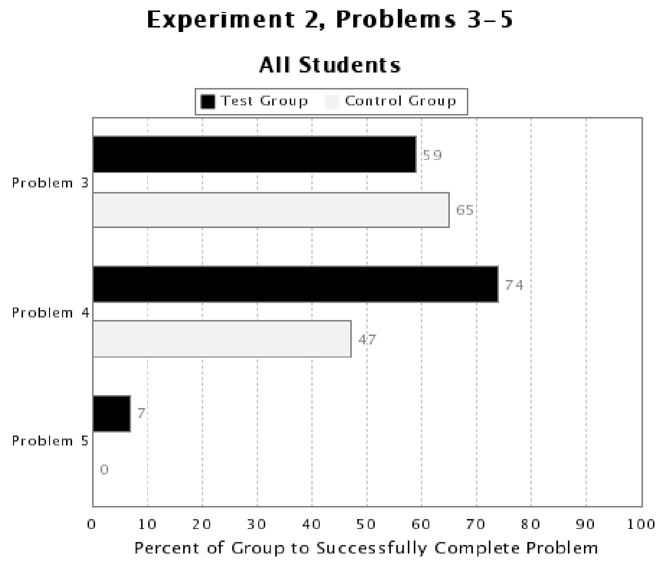


Figure 5.8: Experiment 2 Results for Problems 3-5

CHAPTER 6

FUTURE WORK

Introduction

In the near term, the goal of this research is to provide active learning animation applets for supporting a hypertextbook module on regular languages, finite state automata, regular expressions, and regular grammars, the topics usually found in the first chapter of many textbooks on the theory of computation. (See Chapter 1 for a discussion of hypertextbooks.) The long-term goal of this research is to provide all of the active learning animation applets necessary to create a complete hypertextbook for an undergraduate theory of computation course. By extension, the knowledge and experience gained as these goals are accomplished can then be applied to developing similar hypertextbooks for other topics and disciplines. This chapter will sketch out a road map of future work that will need to be completed in order to accomplish these goals.

Regular Languages Module

The FSA Simulator as it currently stands is a powerful tool for teaching about finite state automata, but it is not sufficient support for teaching the other concepts normally covered along with finite state automata, most notably regular languages, regular expressions and regular grammars (which we refer to here as the "regular languages module"). In order to have a comprehensive resource for teaching and learning this topic, more work needs to be done.

FSA Simulator Enhancements

Many enhancements can be made to the FSA Simulator to improve its support for a regular languages module in a hypertextbook. The flexibility of the Simulator's architecture will allow many new features to be added to the software without requiring significant changes to the existing source code. The current implementation only scratches the surface of its many possible uses.

Alternate Views

One of the important features of many of the algorithm animation packages [19, 75] is the ability to show multiple graphical views of the execution of an algorithm. Something similar could be done for the FSA Simulator. A common representation of FSAs used in textbooks is as a state transition table, which is used to determine which state to change to given the current state and a particular input symbol. A table view of the FSA could easily be included in addition to the default state diagram view.

A “source code” view could also be added to the Simulator by integrating a modified version of the Dynalab program animator [66] that would automatically generate source code for a particular FSA. Both of these views would be separate software modules that would register to receive events from the internal model of the FSA.

Another type of alternate view is a tree view of the execution of a nondeterministic FSA. As the FSA begins execution, the tree view would start as a single node at the root of the tree representing the start state. As the first input symbol is processed, successor states would become child nodes of the root. At each step,

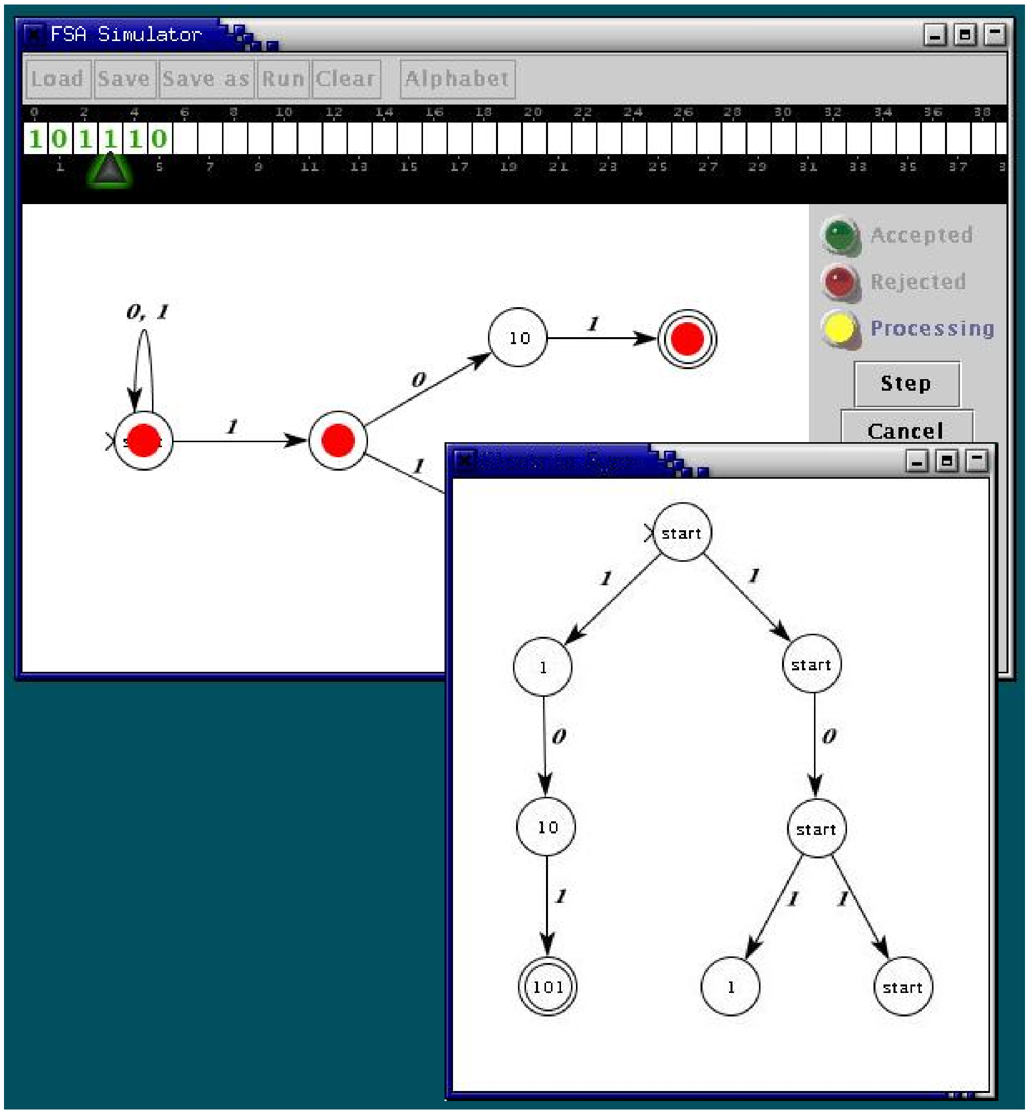


Figure 6.1: Mockup of a tree view

i, of the computation, the combination of the current input symbol with the states represented as leaf nodes at level i of the tree is used to determine the next states, which now become the leaf nodes at level $i+1$. Nondeterministic branches in the computation are represented as branches in the tree view. If there are no next states possible for a state and input symbol pair at level i, that branch ends at level i. See Figure 6.1 for a mockup of a tree view.

FSA Manipulation Algorithms Like JFLAP and JCT, additions to the FSA Simulator should be created to animate the various algorithms that exist for manipulating FSAs. For example, animations should be included to demonstrate the conversion of nondeterministic FSAs to deterministic FSAs, minimization of deterministic FSAs, conversion of regular expressions and regular grammars to FSAs (and vice versa), and so on.

Theorems

The most challenging task for the regular languages module of the hypertextbook would be to devise active learning animations and exercises for teaching the proofs and applications of theorems such as the pumping lemma for regular languages. Ways of illustrating the proof of the pumping lemma itself and its use in showing that languages are not regular will be difficult. As mentioned in Chapter 2, Susan Rodger's PumpLemma software was an interesting– but incomplete–attempt at developing such theorem animation software. More effort needs to be devoted to this area.

Regular Expressions and Regular Grammars

Since algorithms exist to convert regular expressions and regular grammars to FSAs, the FSA comparison features of the Simulator could also be integrated into grammar and regular expression animators to create exercises with instant feedback as well. A prototype applet for regular expression construction exercises has been developed. A project to integrate the FSA comparison code into the Parse Tree Applet (see page 30) to produce regular grammar exercises is also planned.

Automated Grading

Taking the FSA Simulator's ability to assist when grading homework a step further (see page 59), homework assignments and some testing could be completely automated using a client-server architecture. The client program, which could be an applet in a browser or a stand-alone application, would only need the FSA, regular expression, or grammar building program and information about which problem was being solved and how to contact the server. When a student wanted to check an answer or submit an assignment, the student could press a button and a remote procedure call would transmit information about the student's solution to the server. The server would process the information and store the results for later viewing by the instructor. The server could also send a response to the student depending on what kind of assignment is being done. If this is a learning exercise, an indication of whether the submitted solution was correct or an acknowledgement that an assignment has been submitted would be returned to the client.

If set up properly with a large library of problems to assign, grading of homework assignments could be almost entirely automated. Since potential solutions would be evaluated on the server, there would be no danger that students could cheat by extracting the solution from a downloaded jar file or by intercepting network packets going to or from the server.

Practical Programming

In addition to providing an understanding of the mathematical model it animates, the FSA Simulator could also be modified to be used as a practical programming tool. Software packages already exist to generate Java source code from state diagrams [78]. The FSA Simulator (with some modifications) could be used as a front-end to one of these packages to allow students to create useful software applications with their "theoretical" knowledge.

Other Hypertextbook Modules

Once a regular languages module is completed, work can begin on modules for other topics in the theory of computation. The development of the FSA Simulator will allow some of this work to be done with relative ease, but new challenges will also be encountered.

Other Models of Computation

Simulators for other models of computation, including pushdown automata and Turing machines will need to be developed to complete the planned hypertextbook. Much of the FSA Simulator's user interface code can be reused for this purpose, although the code for the internal model of these automata will be significantly different.

Unlike FSAs, the languages recognized by pushdown automata and Turing machines can not be compared for equivalence [72]. Thus, construction exercises using simulators for these models of computation will not be able to definitely tell the student whether the exercise has been successfully completed or not. That, however, does not mean that no feedback can be given. When developing exercises, the author of the exercise can provide a set of strings (in and outside of the language) with which to test the student's automaton. A danger of this approach would be that the student could construct an automaton to accept only the test strings, but not the rest of the strings in the target language. It may be possible, however, to develop dynamic test string generators that would be resistant to such tactics.

Theorems and Proofs

The theorems and proofs found in other sections of the hypertextbook will be even more challenging than those mentioned in Chapter 6. Theorems and proofs related to context-free languages can be animated with an enhanced version of the Parse Tree Applet. Theorems and proofs related to decidability and tractability will be the most challenging. The "Holy Grail" of this line of research would be a set of animations that would illustrate problem reductions and the subject of NP-Completeness (intractability).

Evaluation

Much more work needs to be done in evaluating how the FSA Simulator affects learning. The evaluations done in Chapter 5 were very limited and represent just a start. Much more work needs to be done to reach definite conclusions.

The evaluation process needs to be enhanced to improve the quality of the results. A larger pool of subjects needs to be found for testing so that better selection methods can be used to make the subjects more representative of the target student population. In this case, the investigator was aware of which students were in the test and control groups when the test results were analyzed. It would be better to do a "double blind" study so that the investigator is not aware of which subjects are in the test group. Also, the test instruments need to be more carefully designed to ensure that they are indeed testing students for the correct information.

In addition, there are many situations in which the Simulator needs to be evaluated other than a laboratory setting.

The results of the second evaluation experiment (see page 89) suggest that students may derive more benefit from the FSA Simulator as their knowledge and experience increases. An important follow-up evaluation would be to observe use of the Simulator as a stand-alone application throughout a theory of computation course. How would the FSA Simulator affect learning if students saw it demonstrated during lectures, used it to complete several homework exercises outside of class, and then used it while taking an exam? It would seem that such extensive exposure would have a strong positive effect on how well students would understand finite state automata, but the results are difficult to predict.

It would also be helpful to do a formal evaluation of students' attitudes toward using the Simulator. If it could be shown that use of the FSA Simulator significantly increases students' enthusiasm for the theory of computation, that would be a compelling benefit even without an accompanying increase in students' understanding of the subject.

Another relevant evaluation would be to study the FSA Simulator's effectiveness when used as an integral part of a comprehensive hypertextbook on the theory of computation (see page 4). Ultimately, we will want to determine the effect of a complete hypertextbook on teaching and learning as compared to traditional textbooks.

Conclusion

The development of the FSA Simulator represents a significant progress in creating and evaluating active learning animation software to support the teaching and learning of the theory of computation. The author 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. The door has been opened for the development and evaluation of similar software for other topics in the theory of computation and also for topics that lend themselves to animation in other disciplines.

REFERENCES CITED

[1] Almstrum, V. L., Dale, N., Berglund, A., Granger, M., Little, J. C., Miller, D. M., Petre, M., Schragger, P., and Springsteel, F. Evaluation: turning technology from toy to tool: report of the working group on evaluation. In *Proceedings of the conference on Integrating technology into computer science education* (1996), ACM Press, pp. 201–217.

[2] Arnold, K., Gosling, J., and Holmes, D. *The Java Programming Language*, third ed. Addison-Wesley, 2000.

[3] Badre, A., Beranek, M., Morris, J. M., and Stasko, J. Assessing program visualization systems as instructional aids. Tech. Rep. GIT-GVU-91-23, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, October 1991.

[4] Baecker, R. *Sorting Out Sorting*: A case study of software visualization for teaching computer science. In *Software Visualization: Programming as a Multimedia Experience.* The MIT Press, 1997, ch. 24, pp. 369–381.

[5] Barwise, J., and Etchemendy, J. *Turing's World 3.0: An Introduction to Computability Theory.* CSLI Lecture Notes. CSLI, 1993.

[6] Barwise, J., and Etchemendy, J. Computers, visualization, and the nature of reasoning. In *The Digital Phoenix: How Computers are Changing Philosophy*, T. W. Bynum and J. H. Moor, Eds. Blackwell, London, 1998, pp. 93–116.

[7] Bazik, J., Tamassia, R., Reiss, S. P., and van Dam, A. Software visualization in teaching at brown university. In *Software Visualization: Programming as a Multimedia Experience.* The MIT Press, 1997, ch. 25, pp. 383–398.

[8] Bilska, A. O., Leider, K. H., Procopiuc, M., Procopiuc, O., Rodger, S. H., Salemme, J. R., and Tsang, E. A collection of tools for making automata theory and formal languages come alive. In *Twenty-eighth SIGCSE Technical Symposium on Computer Science Education* (1997), pp. 15–19.

[9] Birch, M. L. An emulator for the e-machine. Master's thesis, Montana State University, June 1990.

[10] Blythe, S., James, M., and Rodger, S. LLparse and LRparse: Visual and interactive tools for Parsing. In *Proceedings of the Twenty-fifth SIGCSE Technical Symposium on Computer Science Education* (1994), pp. 208–212.

[11] 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.

[12] Boroni, C. An ms-windows animator for dynalab (unpublished). Master's thesis, Montana State University, 1996.

[13] Boroni, C. PhD thesis, Montana State University, 2002. Work in progress.

[14] Boroni, C. M., Goosey, F. W., Grinder, M. T., Lambert, J. L., and Ross, R. J. Tying it all together: Creating self-contained, animated, and interactive, web-based resources for computer science education. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education* (March 1999), vol. 31, pp. 7–11.

[15] 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.

[16] Boroni, C. M., Goosey, F. W., Grinder, M. T., Ross, R. J., and Wissenbach, P. Weblab! a univeral and interactive teaching, learning and laboratory environment for the world wide web. In *The Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education* (March 1997), vol. 29, pp. 199–203.

[17] Brase, C. H., and Brase, C. P. *Understandable Statistics: Concepts and Methods.* Houghton Mifflin Company, 1999.

[18] Brooks, Jr, F. P. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1982.

[19] Brown, M. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988.

[20] Brown, M. H. Zeus: A system for algorithm animation and multi-view editing. Tech. Rep. 75, Systems Research Center, Digital Equipment Corporation, April 1992.

[21] Brown, M. H., and Hershberger, J. Color and sound in algorithm animation. Tech. Rep. 76a, Systems Research Center, Digital Equipment Corporation, August 1991.

[22] Brown, M. H., and Najork, M. A. Algorithm animation using 3d interactive graphics. In *Proceedings of the sixth annual ACM symposium on User interface software and technology* (1993), ACM Press, pp. 93–100.

[23] Brown, M. H., and Najork, M. A. Collaborative active textbooks: A web-based algorithm animation system for an electronic classroom. In *IEEE Symposium on Visual Languages* (September 1996), pp. 266–275.

[24] Brown, M. H., and Najork, M. A. Algorithm animation using interactive 3d graphics. In *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1997, ch. 9, pp. 119–135.

[25] Brown, M. H., and Najork, M. A. Three-dimensional web-based algorithm animations. Tech. Rep. 170, Compaq Systems Research Center, July 2001.

[26] Brown, M. H., Najork, M. A., and Raisamo, R. A java-based implementation of collaborative active textbooks. In *IEEE Symposium on Visual Languages* (September 1997), pp. 372–379.

[27] Byrne, M. D., Catrambone, R., and Stasko, J. T. Do algorithm animations aid learning? Tech. Rep. GIT-GVU-96-18, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, August 1996.

[28] Camp, T. The incredible shrinking pipeline. *Communications of the ACM 40*, 10 (October 1997), 103–110.

[29] Caugherty, D., and Rodger, S. H. Npda: a tool for visualizing and simulating nondeterministic pushdown automata. In *Computational Support for Discrete Mathematics*, N. Dean and G. E. Shannon, Eds., vol. 15 of *DIMACS Series in Discrete Mathematics an Theoretical Computer Science*. American Mathematical Society, 1994, pp. 365–377.

[30] Crescenzi, P., Demetrescu, C., Finocchi, I., and Petreschi, R. Reversible execution and visualization of programs with leonardo. *Journal of Visual Languages & Computing 11*, 2 (April 2000), 125–150.

[31] Diva Home Page. http://www.gigascale.org/diva/. Cited September 25, 2002.

[32] Eneboe, T. An ANSI C Compiler for the E-Machine. Master's thesis, Montana State University, June 1995.

[33] Goosey, F. W. A minipascal compiler for the e-machine. Master's thesis, Montana State University, Apr. 1993.

[34] Gramond, E., and Rodger, S. H. Using JFLAP to interact with theorems in automata theory. In *Thirtieth SIGCSE Technical Symposium on Computer Science Education* (1999), pp. 336–340.

[35] Grinder, M. T., Kim, S. B., Lutey, T. L., Ross, R. J., and Walsh, K. F. Loving to learn theory: active learning modules for the theory of computing. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education* (March 2002), ACM Press, pp. 371–375.

[36] Hannay, D. G. Hypercard automata simulation: Finite-state, pushdown and turing machines. *SIGCSE Bulletin 24*, 2 (June 1992), 55–58.

[37] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[38] Hung, T., and Rodger, S. H. Increasing visualization and interaction in the automata theory course. In *Thirty-first SIGCSE Technical Symposium on Computer Science Education* (2000), pp. 6–10.

[39] Hunter, J., and McLaughlin, B. Jdom api documentation. http://www.jdom.org/docs/apidocs/index.html, Mar. 2002. Cited September 25, 2002.

[40] Kehoe, C. M., and Stasko, J. T. Using animations to learn about algorithms: An ethnographic case study. Tech. Rep. GIT-GVU-96-20, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, September 1996.

[41] Kehoe, C. M., Stasko, J. T., and Taylor, A. Rethinking the evaluation of algorithm animations as learning aids: An observational study. Tech. Rep. GIT-GVU-99-10, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, March 1999.

[42] LaLonde, W. R., and Pugh, J. R. *Inside Smalltalk, Volume II.* Prentice Hall, 1991.

[43] 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.

[44] Lewis, H. R., and Papadimitriou, C. H. *Elements of the Theory of Computatioin.* Prentice-Hall, 1981.

[45] Lieberman, H., and Fry, C. ZStep 95: A Reversible, Animated Source Code Stepper. In *Software Visualization: Programming as a Multimedia Experience.* The MIT Press, 1997, ch. 19, pp. 277–292.

[46] LoSacco, M., and Rodger, S. FLAP: A Tool For Drawing and Simulating Automata. In *ED-MEDIA 93, World Conference on Educational Multimedia and Hypermedia* (1993), pp. 310–317.

[47] Luce, E., and Rodger, S. A Visual Programming Environment for Turing Machines. In *Proceedings of the IEEE Symposium on Visual Languages* (1993), pp. 231–236.

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

[49] Mayer, R. E., and Anderson, R. B. Animations need narrations: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology 83*, 4 (1991), 484–490.

[50] 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.

[51] Mayer, R. E., and Gallini, J. K. When is an illustration worth ten thousand words? *Journal of Educational Psychology 82*, 4 (1990), 715–726.

[52] 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.

[53] McFall, R., and Dershem, H. L. Finite state machine simulation in an introductory lab. In *Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education* (1994), ACM Press, pp. 126–130.

[54] The Montana State University Bulletin, June 2002.

[55] Najork, M. Web-based algorithm animation. In *38th Design Automation Conference* (June 2001), pp. 506–511.

[56] Naps, T. L. Design of a completely general algorithm visualization system. In *Proceedings of the 22nd Small College Computing Symposium* (April 1989), pp. 233–241.

[57] Naps, T. L. Algorithm visualization in computer science laboratories. In *Proceedings of the twenty-first SIGCSE technical symposium on Computer science education* (1990), ACM Press, pp. 105–110.

[58] Naps, T. L. Algorithm visualization served off the world wide web: why and how. In *Proceedings of the conference on Integrating technology into computer science education* (1996), ACM Press, pp. 66–71.

[59] Naps, T. L. Algorithm visualization on the world wide web?the difference java makes! In *Proceedings of the conference on Integrating technology into computer science education* (1997), ACM Press, pp. 59–61.

[60] Naps, T. L., and Bressler, E. A multi-windowed environment for simultaneous visualization of related algorithms on the world wide web. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education* (1998), ACM Press, pp. 277–281.

[61] Naps, T. L., Eagan, J. R., and Norton, L. L. Jhav??an environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education* (2000), ACM Press, pp. 109–113.

[62] Naps, T. L., and Hundhausen, C. D. The evolution of an algorithm visualization system. In *Proceedings of the 24th Small College Computing Symposium* (April 1991), pp. 259–263.

[63] Naps, T. L., and Swander, B. An object-oriented approach to algorithm visualization?easy, extensible, and dynamic. In *Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education* (1994), ACM Press, pp. 46–50.

[64] Patton, S. D. The e-machine: Supporting the teaching of program execution dynamics. Master's thesis, Montana State University, June 1989.

[65] Poole, D. An Ada/CS compiler for the e-machine. Master's thesis, Montana State University, July 1994.

[66] Pratt, M. An OSF/motif program animator for the dynalab system. Master's thesis, Montana State University, May 1995.

[67] Procopiuc, M., Procopiuc, O., and Rodger, S. Visualization and interaction in the computer science formal languages course with JFLAP. In *1996 Frontiers in Education Conference* (1996), pp. 121–125.

[68] Reiss, S. P. Visualization for software engineering – programming environments. In *Software Visualization: Programming as a Multimedia Experience*. The MIT Press, 1997, ch. 18, pp. 259–276.

[69] Rieber, L. P., Boyce, M., and Assad, C. The effects of computer animation on adult learning and retrieval. *Journal of Computer-Based Instruction 17*, 2 (1990), 46–52.

[70] Robinson, M. B., Hamshar, J. A., Novillo, J. E., and Duchowski, A. T. A java-based tool for reasoning about models of computation through simulating finite automata and turing machines. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education* (1999), ACM Press, pp. 105–109.

[71] Rodger, S. H. An interactive lecture approach to teaching computer science. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* (1995), pp. 278–282.

[72] Sipser, M. *Introduction to the Theory of Computation.* PWS Publishing, 1997.

[73] Slavin, R. E. *Research Methods In Education.* Prentice-Hall, 1984.

[74] Stallmann, M., Cleaveland, R., and Hebbar, P. Gdr: A visualization tool for graph algorithms. Tech. rep., Department of Computer Science, North Carolina State University, 1991.

[75] Stasko, J. TANGO: A framework and system for algorithm animation. *IEEE Computer 30*, 3 (Sept. 1990), 27–39.

[76] Stasko, J. Smooth, continuous animation for portraying algorithms and processes. In *Software Visualization: Programming as a Multimedia Experience.* The MIT Press, 1997, ch. 8, pp. 103–118.

[77] 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.

[78] SMC Home Page. http://smc.sourceforge.net. Cited November 3, 2002.

[79] Strout, J. J. Learning from leonardo. *MacTech 15*, 10 (October 1999).

[80] Wirth, N., Jensen, K., and Mickel, A. *Pascal User Manual and Report: ISO Pascal Standard*, 4th ed. Springer Verlag, 1991.

[81] Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/2000/REC-xml-20001006, Oct. 2000. Cited October 9, 2002.

APPENDICES

APPENDIX A

XML FILE FORMAT

DOCUMENT TYPE DEFINITION FOR FSA FILES

```
<!-- This DTD defines the format of a graph.-->

<!-- ============================================================ -->
<!-- fsa                                                          -->
<!-- ============================================================ -->

<!ELEMENT fsa (description, alphabet, states, initial_state,
               final_states, transitions)>

<!-- ============================================================ -->
<!-- alphabet                                                     -->
<!-- ============================================================ -->

<!ELEMENT description (#PCDATA)>

<!ELEMENT alphabet (#PCDATA)>

<!ELEMENT states (state*)>

<!ELEMENT initial_state EMPTY>
<!ATTLIST initial_state stateid IDREF #REQUIRED>

<!ELEMENT final_states EMPTY>
<!ATTLIST final_states stateids CDATA #REQUIRED>


<!ELEMENT transitions (transition*)>

<!-- ============================================================ -->
<!-- state                                                        -->
<!-- ============================================================ -->

<!ELEMENT state (property*)>
<!ATTLIST state id        ID          #REQUIRED>


<!-- ============================================================ -->
<!-- transition                                                   -->
<!-- ============================================================ -->
```

```
<!ELEMENT transition (property*)>
<!ATTLIST transition id         ID    #REQUIRED
                     state      IDREF #REQUIRED
                     symbol     CDATA #REQUIRED
                     next_state IDREF #REQUIRED>

<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #REQUIRED>
```

EXAMPLE FSA FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE fsa SYSTEM "fsa.dtd">
<fsa>
  <description />
  <alphabet>01</alphabet>
  <states>
    <state id="s0">
      <property name="x">75.14644622802734</property>
      <property name="y">110.0</property>
      <property name="label">even</property>
      <property name="description">Even number of 1s</property>
    </state>
    <state id="s1">
      <property name="x">289.0</property>
      <property name="y">110.0</property>
      <property name="label">odd</property>
      <property name="description">Odd number of 1s</property>
    </state>
  </states>
  <initial_state stateid="s0" />
  <final_states stateids="s0" />
  <transitions>
    <transition id="e0-1" state="s0" symbol="1" next_state="s1" />
    <transition id="e1-0" state="s1" symbol="1" next_state="s0" />
    <transition id="e0-0" state="s0" symbol="0" next_state="s0" />
    <transition id="e1-1" state="s1" symbol="0" next_state="s1" />
  </transitions>
</fsa>
```

APPENDIX B

FSA SIMULATOR DEVELOPMENT INFORMATION

FSA SIMULATOR DEVELOPMENT INFORMATION

Implementation Language: Java
Number of source code files: 58
Total Physical Source Lines of Code (SLOC): 7,056
Special libraries used: Diva, JDOM, and util.concurrent

File and SLOC counts were generated using 'SLOCCount' by David A. Wheeler.

APPENDIX C

EVALUATION INSTRUMENTS

STUDENT INFORMATION SHEET

Webworks Project Evaluation Lab

Student Information Sheet

April 16, 2002

This lab will attempt to teach you about finite state automata, an important concept used throughout computer science. We are evaluating some software that is designed to help students learn about finite state automata. Some of you will be using the software. Others will be doing traditional pencil-and-paper exercises. All of you will take a test at the end of the lab to try to gauge how much you learned about finite state automata during the lab. Please fill out some information about yourself below. This information will be kept confidential and will only be used for statistical purposes. To help protect your privacy, you have been given a number that only your TA will be able to associate with your name.
Number: _______

Sex: Male___ Female___
Age ____
Major: ___________________________________

Year in College (circle one): Freshman Sophomore Junior Senior Graduate Other
If you circled Other, please give a brief explanation:

Are you familiar with the terms finite automaton, finite state automaton, finite state machine, regular language, regular expression, or regular grammar? If so, where did you learn about them?

GPA (circle approximate range): 3.5-4.0 3.0-3.5 2.5-3.0 2.0-2.5 <2.0
ACT/SAT score(s) (if you remember them): _________

TEST

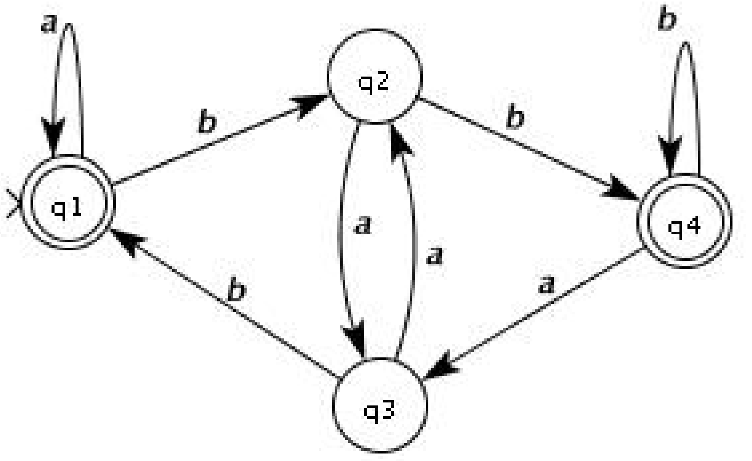
Webworks Project Evaluation Lab

Finite Automata Test

April 16, 2002

Number: ________

1.) Answer the questions below using the following state diagram of a finite automaton:



a.) What is the start state?

b.) What is the set of accept states?

c.) What is the set of input symbols?

d.) What sequence of states does the automaton go through on input aabb?

e.) What is an example of a string that is accepted?

f.) What is an example of a string that is rejected?

g.) Does it accept the string aabb?

h.) Does it accept the string babb?

i.) Does it accept the string abaab?

j.) Does it accept the string aabaabbaab?

2.) Give the formal description of the finite automaton pictured in Question 1.

3.) Draw a state diagram of a finite automaton that recognizes the language using the input symbols $\{0, 1\}$ where all strings in the language begin with a 1 and end with a 0. Examples of strings in this language include 10, 10110, 11100.

4.) Draw a state diagram of a finite automaton that recognizes the language using the input symbols {0, 1} where all strings in the language contain at least three 1s. Examples of strings in this language include 111, 000101010111, and 0101010.

5.) Draw a state diagram of a finite automaton that recognizes the language over the alphabet {0, 1} where all strings in the language do not contain the substring 110. Examples of strings in this language include 0, 1, 00011, 11, and 010101.