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

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 "hypertext book." 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

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 Rockford J. Ross 11/26/02
(Signature) Date

Approved for the Department of Computer Science

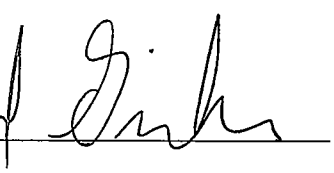
Rockford J. Ross Rockford J. Ross 11/26/02
(Signature) Date

Approved for the College of Graduate Studies

Bruce McLeod Bruce R. McLeod 11-26-02
(Signature) Date

STATEMENT OF PERMISSION TO USE

In presenting this dissertation in partial fulfillment of the requirements for a doctoral degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. I further agree that copying of this dissertation is allowable only for scholarly purposes, consistent with fair use as prescribed in the U.S. Copyright Law. Requests for extensive copying or reproduction of this dissertation should be referred to Bell & Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, Michigan 48106, to whom I have granted the exclusive right to reproduce and distribute my dissertation in and from microform along with the non-exclusive right to reproduce and distribute my abstract in any format in whole or in part.

Signature 

Date 11/26/2002

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 "hypertext-book." 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 hypertext-book [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 prelimi-

nary 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 hypertext-book 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

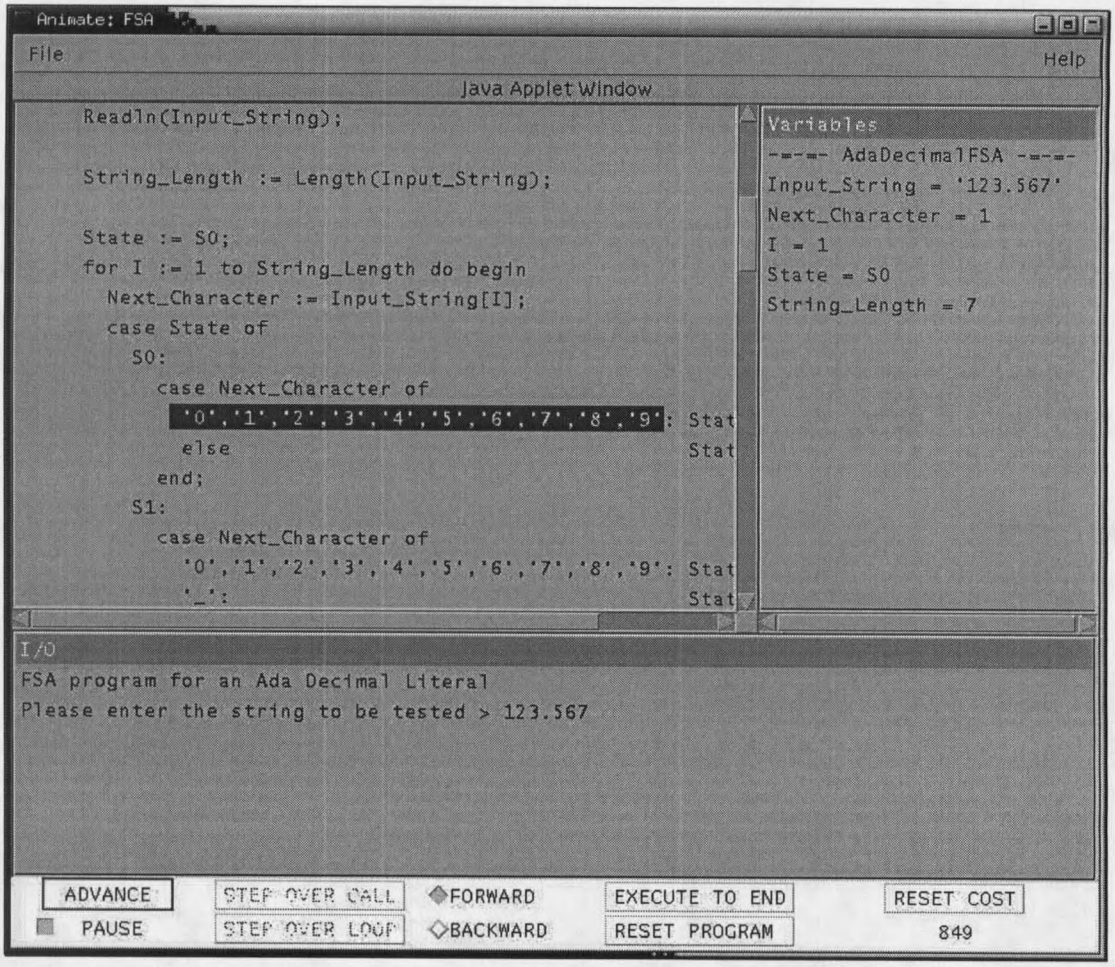


Figure 2.1: Java version of the Dynalab program animator

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.

TUMS 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.

NPDA, FLAP and JFLAP 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

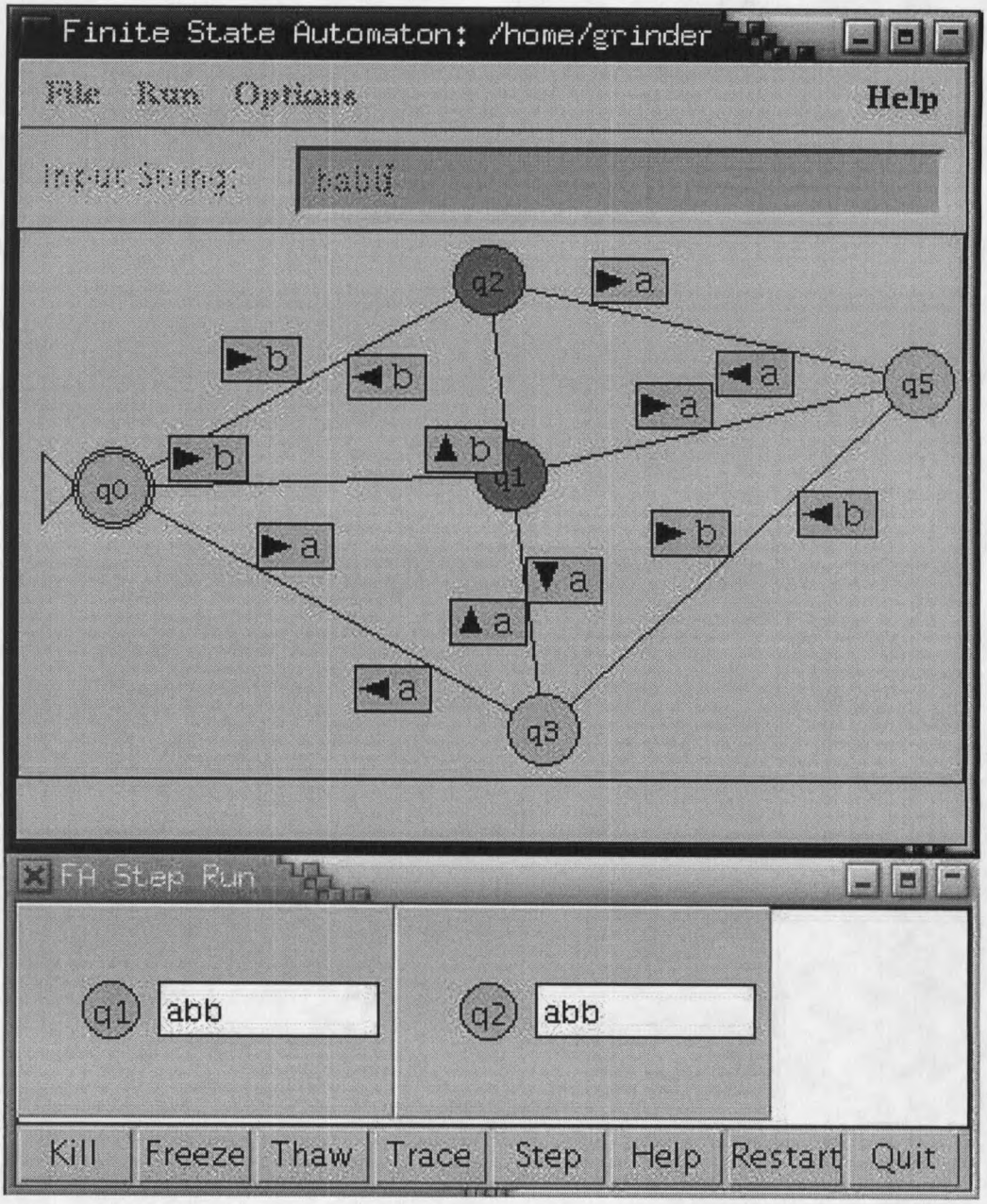


Figure 2.2: A Nondeterministic FSA in JFLAP

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

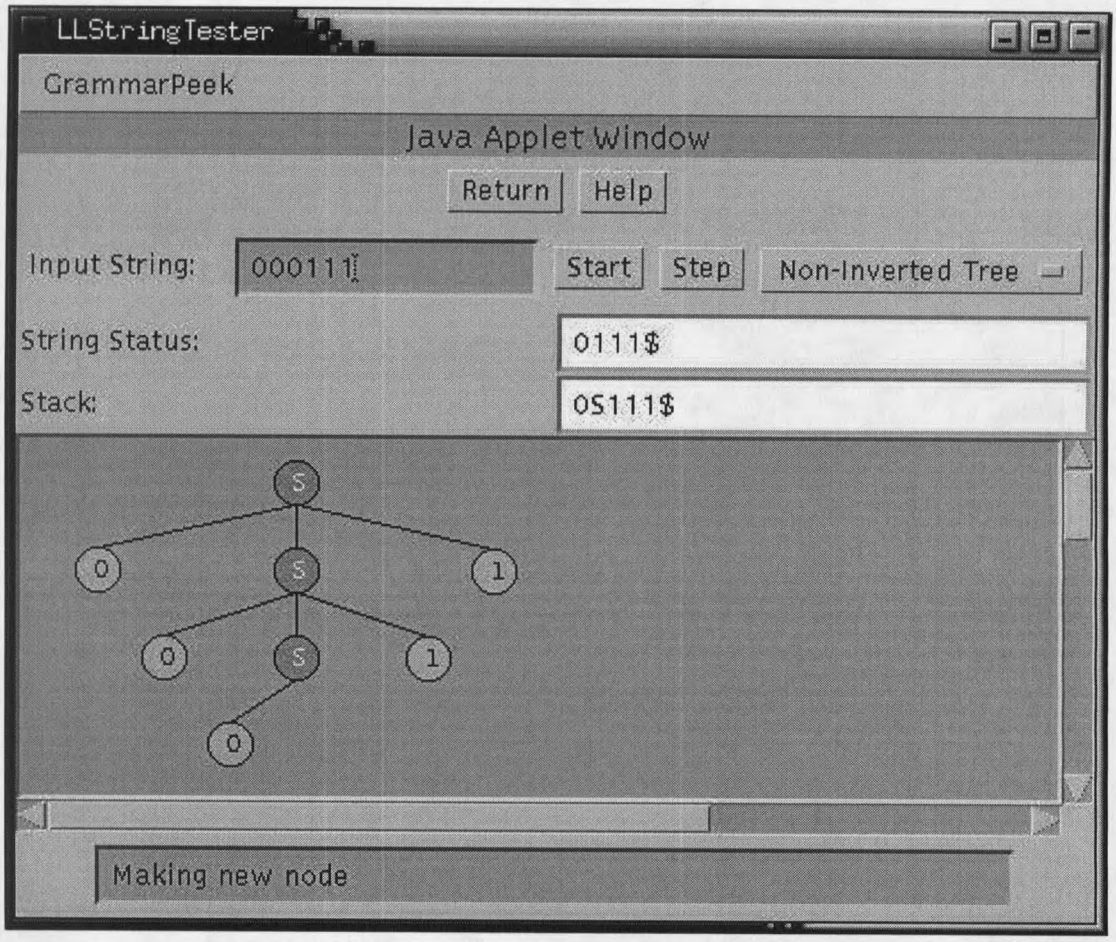


Figure 2.3: JeLLRap's Parse Tree View

"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

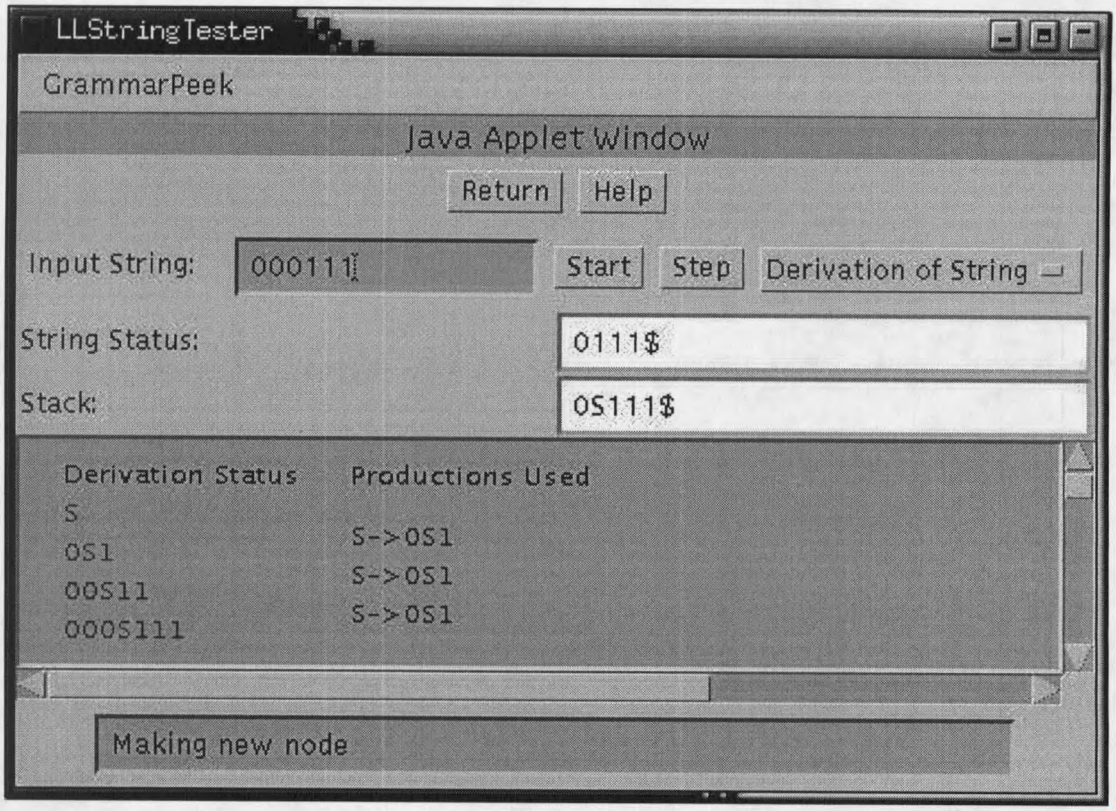


Figure 2.4: JeLLRap's String Derivation View

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

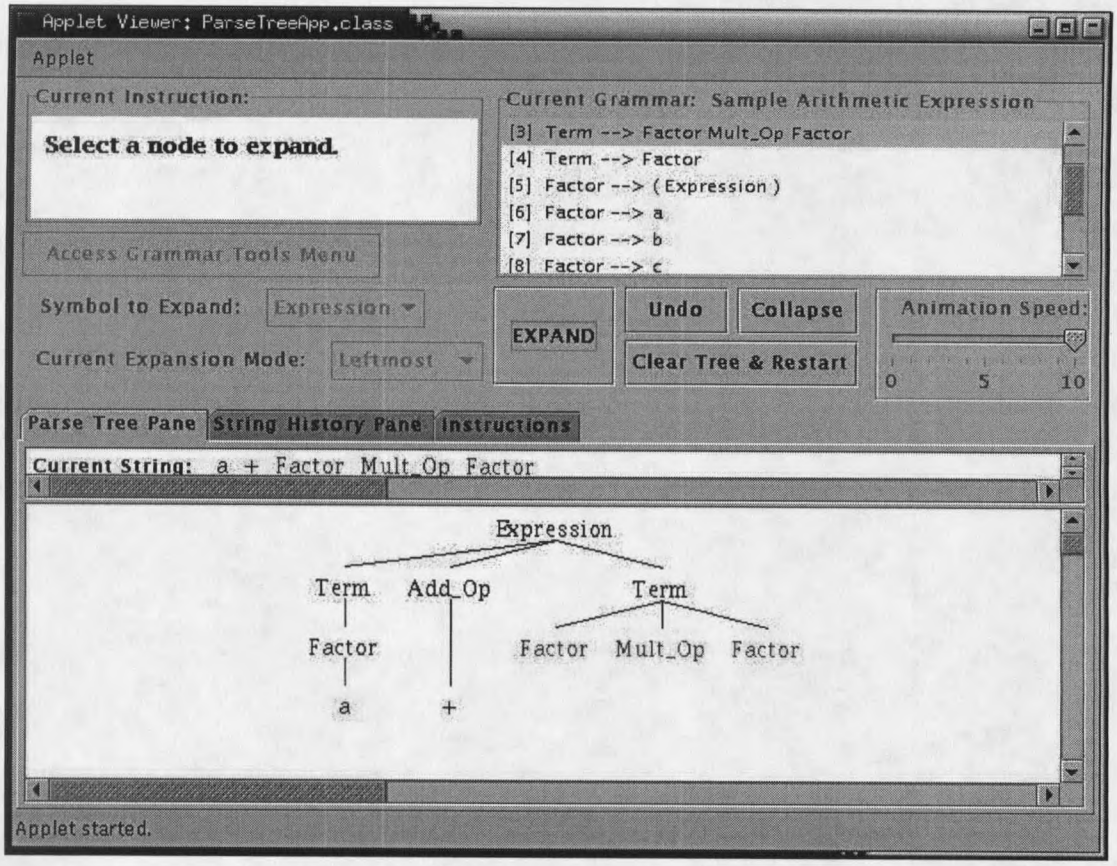


Figure 2.5: The Parse Tree Applet

be investigated. PumpLemma restricts the languages to relatively simple strings such as $a^n b^n$ 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

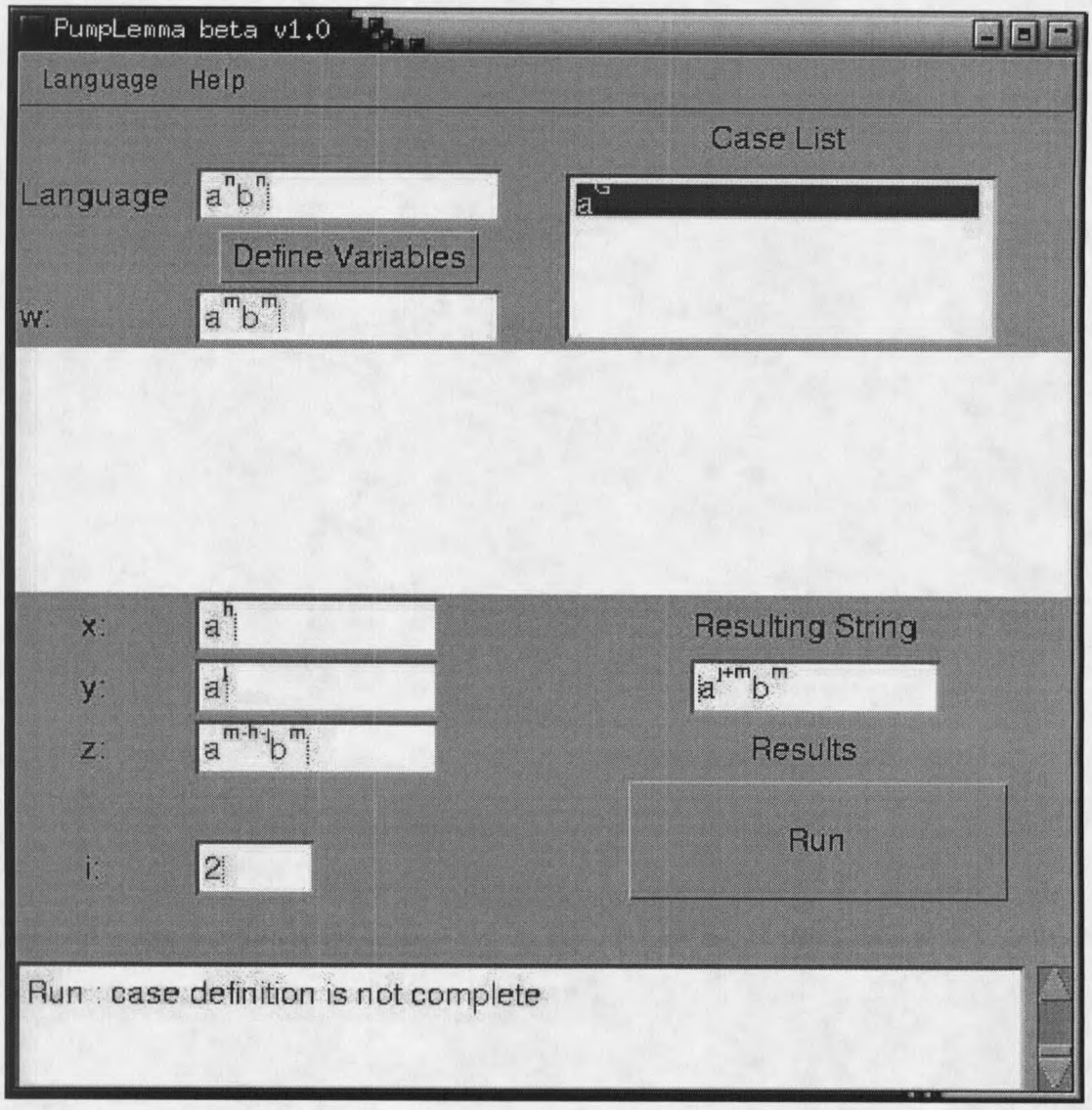


Figure 2.6: PumpLemma in action

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 pro-

cess. 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 stu-

dents' 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 videotaped 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

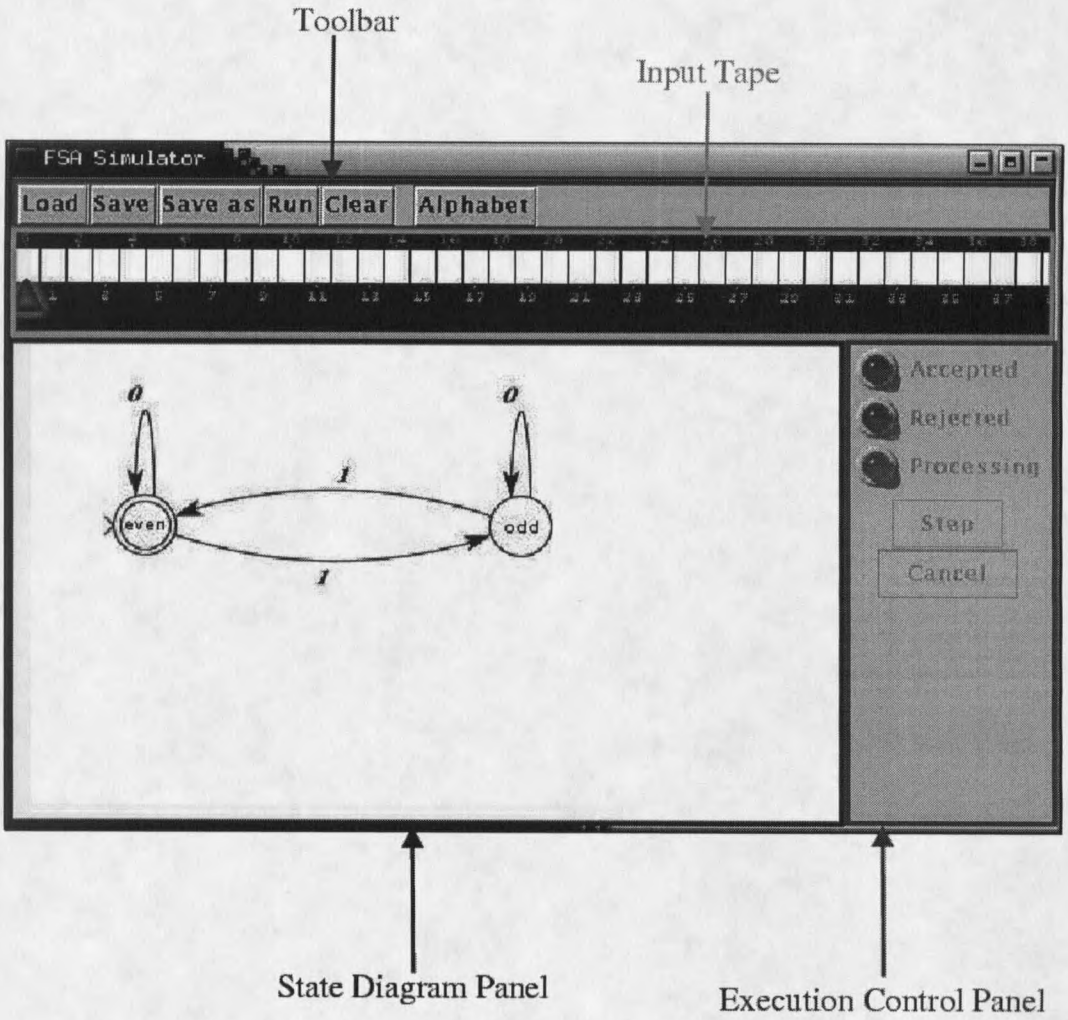


Figure 3.1: User Interface of the FSA Simulator

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

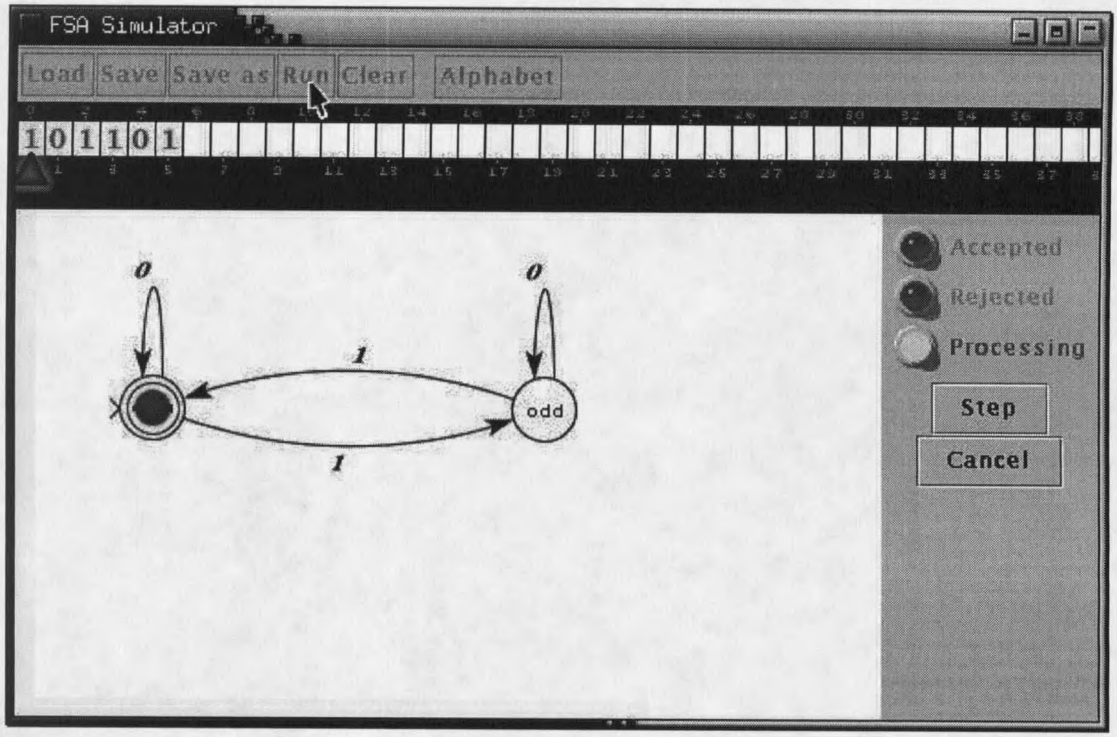


Figure 3.2: The FSA Simulator After Entering "Run" Mode

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

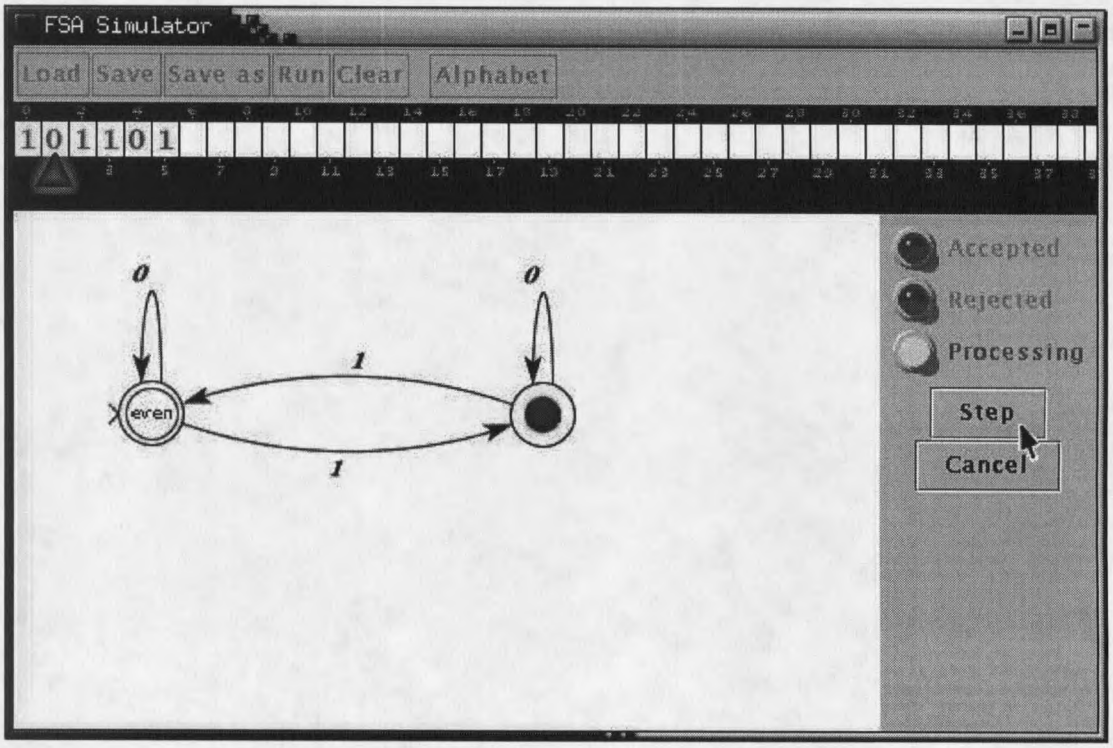


Figure 3.3: The FSA Simulator During Execution

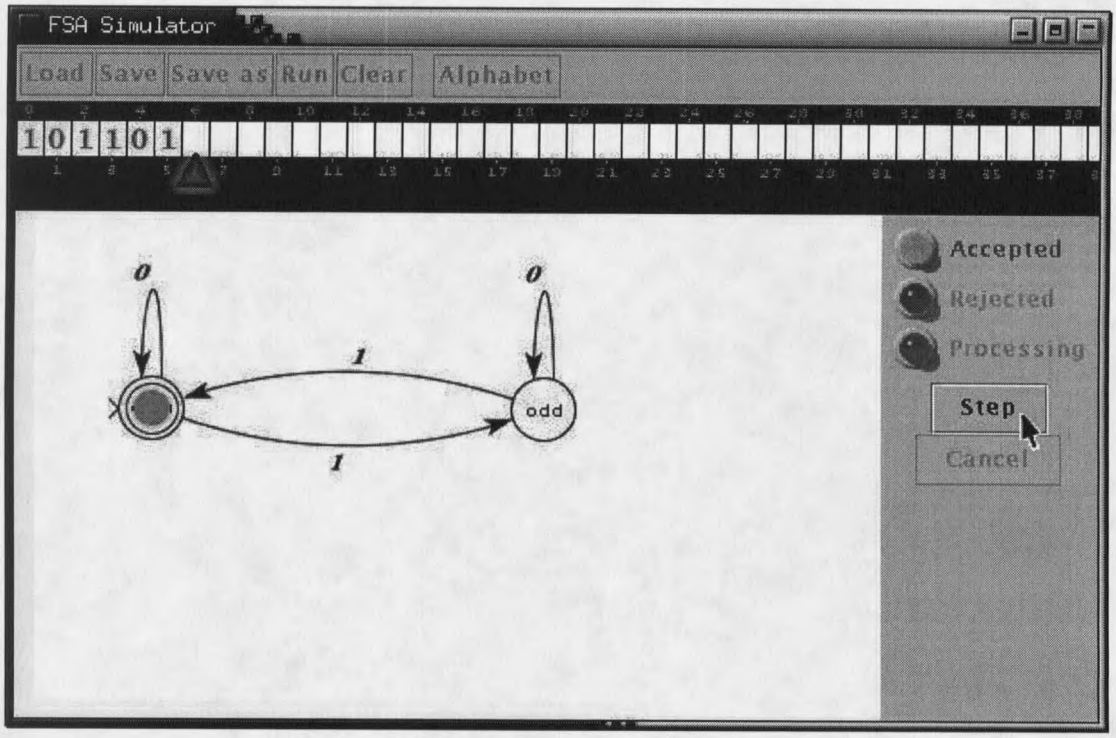


Figure 3.4: The FSA Simulator Accepting a String

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