ProofChecker: An Accessible Environment for Automata Theory Correctness Proofs

Matthias Stallmann, Suzanne Balik, Robert Rodman,
Sina Bahram, Michael Grace, and Susan High
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
{mfms, spbalik, rodman, sbahram, mcgrace, sdhigh}@ncsu.edu

ABSTRACT

ProofChecker is a graphical program based on the notion of formal correctness proofs that allows students, both sighted and visually impaired, to draw a deterministic finite automaton(DFA) and determine whether or not it correctly recognizes a given language. Sighted students use the mouse and graphical controls to draw and manipulate the DFA. Keyboard shortcuts, together with the use of a screen reader to voice the accessible descriptions provided by the program, allow visually impaired students to do the same.

Because the states of a DFA partition the language over its alphabet into equivalence classes, each state has a language associated with it. Conditions that describe the language of each state are entered by the student in the form of conditional expressions with function calls and/or regular expressions. A brute-force approach is then used to check that each state's condition correctly describes all of the strings in its language and that none of the strings in a state's language meet the condition for another state. Feedback is provided that either confirms that the DFA correctly meets the given conditions or alerts the student to a mismatch between the conditions and the DFA.

A student's DFA can be saved in an XML file and submitted for grading. An automated checking tool, known as ProofGrader, can be used to compare a student's DFA with the correct DFA for a given language, thus greatly speeding up the grading of student assignments.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*finite automata*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*regular languages*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

ITiCSE'07, June 23–27, 2007, Dundee, Scotland, United Kingdom.

General Terms

Theory, Verification, Human Factors

Keywords

finite automata, correctness proof, accessibility

1. INTRODUCTION

ProofChecker was developed at North Carolina State University to help students design and determine the correctness of a DFA for a particular language. As noted in [11], drawing DFA's using paper and pencil and testing them is tedious for student and instructor alike and often error prone. Packages such as JFLAP[12] and jFAST[14] allow students to draw and test DFA's by simulating them with individual strings. ProofChecker provides a different theoretical and pedagogical approach that allows students to draw a DFA and define the language of each state. A large number of strings over the alphabet are then automatically generated and tested against those languages. ProofChecker has the added advantage of being accessible to both sighted and non-sighted students.

ProofChecker has been used in our Automata, Grammars, and Computability course since Fall 2003. Students enjoy drawing DFA's graphically and receiving feedback as to their correctness — 96% of the students surveyed in Spring 2004 indicated that ProofChecker was helpful in their understanding of DFA's. Teaching assistants appreciate the decrease in grading time made possible by the ProofGrader program that compares a student's DFA with the correct DFA for equivalence. If they are *not* equivalent, ProofGrader lists short strings in the language of one DFA and not the other and vice-versa. This feature helps to provide insight into the student's error(s) as well as with assigning partial credit.

In Fall 2004, a visually impaired student was enrolled in the automata course. Because much of the material for the course — finite automata, push-down automata, and Turing machines — is presented in visual form, this posed quite a challenge for both the instructor and the student. Raised print diagrams were used by the student during lecture as an alternative to what was being drawn on the board. The instructor learned to give an audible description of the automaton being presented in addition to the visual one. In order to complete homework and tests, the student described his automata to a teaching assistant who drew them for him.

The student in question was intrigued by the ProofChecker GUI-based program. He worked with the teaching assistant to make ProofChecker accessible to him through the use of keyboard shortcuts, the Java Accessibility API [5], the Java Access Bridge [10], and the JAWS® for Windows® screen reader [6]. This allowed him to independently design and draw his own DFA's and submit them for grading. Since that time, the accessible interface has been improved and the extension of ProofChecker to other formalisms such as context-free grammars is underway. This development has great potential for helping other visually impaired students learn automata theory as well.

The remainder of this paper is organized as follows. Section 2 discusses the theoretical background behind formal correctness proofs for DFA's. Section 3 presents an overview of ProofChecker and Section 4 presents the accessible interface. Future directions are considered in Section 5.

ProofChecker and ProofGrader may be downloaded from `http://research.csc.ncsu.edu/accessibility/ProofChecker/`.

2. THEORETICAL BACKGROUND

ProofChecker is based on the fact that the states of a DFA partition the language over its alphabet into equivalence classes. In automata theory textbooks, this concept is used primarily in the minimization of DFA's [4, 8, 9] and only occasionally in their design [9, 13]. Defining these equivalence classes is an important part of a proof of correctness for a DFA as demonstrated by Hopcroft & Ullman [4, pp. 48-53]. For example, the DFA below recognizes the following language where $\Sigma = \{a, b\}$:

$L = \{w \in \Sigma^* \mid w$ has an odd number of a's and ends with b$\}$

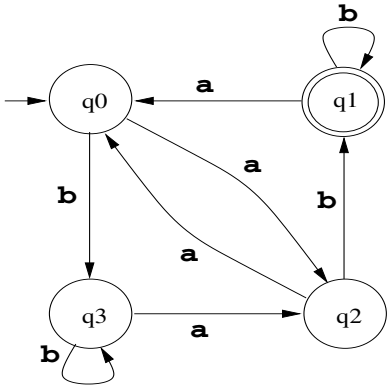

Figure 1: DFA Example

This DFA partitions Σ^* into the following equivalence classes (or languages) associated with each state:

$L(q_0)$ = $\{w \mid w$ has an even number of a's and does not end with b$\}$
$L(q_1)$ = $\{w \mid w$ has an odd number of a's and ends with b$\}$
$L(q_2)$ = $\{w \mid w$ has an odd number of a's and ends with a$\}$
$L(q_3)$ = $\{w \mid w$ has an even number of a's and ends with b$\}$

The languages of each state can alternatively be defined using regular expressions:

$L(q_0) = (b^*ab^*a)^*$
$L(q_1) = b^*(ab^*ab^*)^*ab^+$
$L(q_2) = b^*(ab^*ab^*)^*a$
$L(q_3) = b^*(ab^*ab^*)^*b$

A proof of correctness for this (or any) DFA involves showing that the DFA is well-formed, that all strings in Σ^* are included in exactly one of the state languages, and that the transitions are consistent with the specification. Also, the union of the languages of the accepting states must be equivalent to the language recognized by the DFA. Informally using this approach can help students design and verify the correctness of a DFA as well as determine the language of an existing DFA — in fact, students in our automata theory course are routinely taught and required to analyze DFA's in this way. Exposure to DFA correctness proofs also helps to prepare students for reasoning about the correctness of programs and concurrent systems.

ProofChecker provides an automated mechanism for students to draw a DFA, determine whether or not it is well-formed, and define the language of each state. It then tells them whether or not those languages are disjoint and whether they accurately describe the set of strings for each state. This effectively automates the formal proof that

$$\delta^*(q_0, w) = q \text{ if and only if } w \in L(q) \text{ where } w \in \Sigma^*$$

The ProofChecker rendering of the DFA in Figure 1 is shown in Figure 2. The language descriptions for states `q0` and `q1` have been entered using ProofChecker-style functions while those for states `q2` and `q3` are given as regular expressions as shown below:

```
q0:  even(numberOf("a")) && !endsWith("b")
q1:  odd(numberOf("a")) && endsWith("b").
q2:  b*(ab*ab*)*a
q3:  b*(ab*ab*)*b.
```

The dialog box with the congratulatory ProofChecker Message, "`Wow. All of the conditions you selected to check are correct.`" indicates that the student was successful in correctly describing the language of each state.

3. OVERVIEW OF PROOFCHECKER

ProofChecker is designed to be easy to use and intuitive. Functionality exists to create and edit a DFA, enter conditions which describe the language of each state, and to check whether these languages are correct. DFA's can be saved in XML files and reopened with ProofChecker.

The ProofChecker GUI has three main panels. The largest panel is a canvas where the DFA is drawn. To the left of the canvas is a scrolling panel with a text area for each state in which a condition describing its language can be entered. The top left corner contains a panel where the student can enter comments about the DFA, for example, a description of its language or its homework problem number.

The GUI also contains two tool bars, a general one which will be used for other formalisms as well, and one specific to manipulating DFA's. A menu bar is available for working with files, checking the DFA, and accessing Help.

This section describes how a sighted user would typically interact with ProofChecker using the mouse, keyboard, and graphical components. The next section describes how a visually impaired user can do the same things using only the keyboard and the Jaws® for Windows® screen reader.

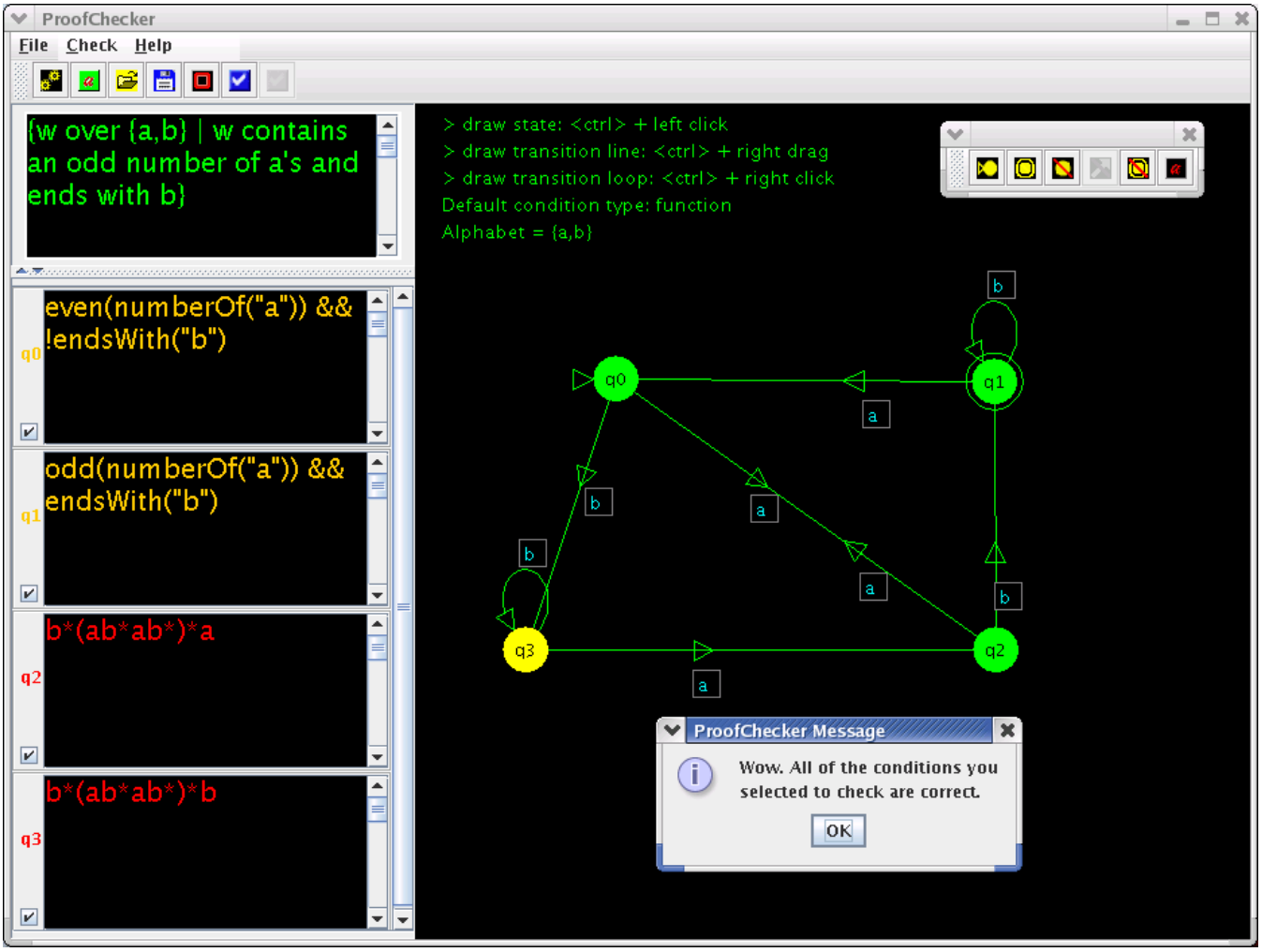

Figure 2: ProofChecker in action, showing the DFA of Figure 1

3.1 Creating and Editing a DFA

A new state is created by holding down the `ctrl` key and left-clicking with the mouse on the canvas area. The `ctrl` key plus a right-click on a state creates a self-transition. Right-dragging from one state to another while holding down the `ctrl` key creates a transition between them. Each transition has an associated text box where its alphabet symbol(s) can be entered. Clicking on the text box for an existing transition allows for editing of its symbols.

A state and its transitions can be moved by simply dragging the state to a new location on the canvas. Tool bar buttons are used to mark a state as the start and/or an accepting state as well as to remove it and its transitions from the DFA. A button to remove a single transition is also available.

3.2 Entering State Conditions

Each state has an associated text area where a condition describing its language may be entered. The default for expressing conditions is Function Mode in which the strings in a state's language are described in the form of a conditional expression using one or more ProofChecker functions. Right-clicking on the text area brings up a popup menu containing the available function choices as well as the option to switch to Regular Expression Mode. Table 1 lists these functions together with examples of their use.

Arithmetic, relational, and logical operators may be used in Function Mode to form more complex conditions to describe the strings of a language, such as

```
even(numberOf("0")) && length() > 0

numberOf("a") % 3 == 2
```

This rather complicated language from an exercise in Hopcroft and Ullman [4]:

$$\{w \in \{0,1\}^* \mid w = x(10)^k, \text{where } x \text{ does not end in 10 and } k \text{ is odd}\}$$

can be defined as the concatenation of two languages using the ProofChecker expression:

```
concat(!endsWith("10"),
star("10") && odd(numberOf("10")))
```

Function prototype	Example condition	Strings in $\{a, b\}^*$ that match the example
`bool const("string")`	`const("ab") || const("ba")`	`ab, ba`
`bool startsWith("string")`	`startsWith("b")`	`b, ba, bb, baa, bab, bba, bbb, ...`
`bool endsWith("string")`	`endsWith("ba")`	`ba, aba, bba, aaba, abba, baba, bbba, ...`
`bool star("string")`	`star("ab")`	ε, `ab, abab, ababab, abababab, ababababab, ...`
`int length()`	`length() < 3`	ε, `a, b, aa, ab, ba, bb`
`int numberOf("string")`	`numberOf("aa") == 0`	ε, `a, b, ab, ba, bb, aba, abb, bab, bba, bbb, ...`
`int even()`	`even(numberOf("a"))`	ε, `b, aa, bb, aab, aba, baa, bbb, ...`
`int odd()`	`odd(length())`	`a, b, aaa, aab, aba, abb, baa, bab, bba, bbb, ...`
`bool concat(bool, bool)`	`concat(const("b"), star("ab"))`	`b, bab, babab, bababab, babababab, ...`

Table 1: ProofChecker Functions

In Regular Expression Mode, the DFA's alphabet symbols may be combined with `|` or `U` (union), and the `*`, `+`, and `()` characters to form regular expressions describing the state languages. The empty string is represented by the character `e`. Escaping any of these characters by prefixing it with a backslash (\) will cause it to be treated as a symbol of the alphabet Σ.

3.3 Checking the DFA for Correctness

When the **Check** button is pressed or **Check > State Conditions** is selected from the menu, the DFA is checked for correctness. The student is first alerted if the DFA is not well-formed, i.e., does not contain a start state, does not have a transition from each state on every alphabet symbol, etc. If it is well-formed, then the state conditions the student has selected will be checked. This is done by generating all strings over the alphabet Σ up to an arbitrary length ℓ, which results in the generation of $(|\Sigma|^{\ell+1} - 1)/(|\Sigma| - 1)$ strings. For each string, the program checks that (a) it meets the condition of the state in which it ends up and (b) that it meets the condition of *only* that state.

Setting the value of ℓ to $n + 2$, where n is the number of states, allows the checking to be done in a reasonable amount of time and is generally successful in ferreting out student errors. For example, in the case of a 4 state DFA with a 2 character alphabet, 127 strings would be generated and checked. After many semesters of using ProofChecker, only one counterexample has been found in which a student's conditions were correct for strings of length up to $n+2$, but the DFA was not. This interesting counterexample is posted on our website. In this case, using strings of length up to $n + 3$ would have caught the error and is certainly doable.

If the state conditions do not correctly match the DFA, an error message is given. For example, if the condition for state q0 in the aforementioned DFA was incorrectly given as `even(numberOf("a")) && endsWith("a")`, the following **State Condition Error** would be shown in a popup window:

```
the empty string does not satisfy state q0 condition:
even(numberOf("a")) && endsWith("a")
```

If instead, the condition for state q0 was incorrectly entered as `even(numberOf("a"))`, the following **State Condition Error** would be shown:

```
Condition for state q0: even(numberOf("a"))
is true for strings in more than one state;
e.g., it is also true for "b",
which ends up in state q3.
```

4. THE ACCESSIBLE INTERFACE

The same functionality described in the previous section is provided to visually impaired students through keyboard shortcuts and audible feedback. This is accomplished by mapping keystrokes to actions and setting the accessible descriptions provided through the Java Accessibility API [5] appropriately. In order for a Windows-based assistive technology such as Jaws [6] to interact with the Java Accessibility API, the Java Access Bridge [10] must first be installed. Examples and guidelines for making Java applications accessible may be found in [3] and on our website located at `http://research.csc.ncsu.edu/accessibility/`.

4.1 Navigation

While sighted students can visually examine the GUI and select states, transitions, menu options, and text areas with the mouse, non-sighted students must rely on audible feedback and the keyboard to do the same things.

The up/down arrows are used to move from state to state in a DFA. The `enter` key is used to toggle between a state and its transitions. When a state's transitions have focus, the up/down arrows are used to move between them. Thus a state or transition can be selected analogous to clicking on it with the mouse.

Each time a state is selected, its name and whether or not it is the start state and/or an accepting state is voiced. The voicing of information about a state's "to" and "from" transitions may be turned on/off using `alt-t` and `alt-f` respectively. This allows the DFA to be audibly examined in the most efficient way for the task at hand.

The menu bar and menus are accessed through the use of mnemonics, with `alt-f`, `alt-c`, and `alt-h`, used to select the **File**, **Check**, and **Help** menus respectively. Pressing `F1` brings up a list of all available keyboard shortcuts.

4.2 Creating and Editing a DFA

`Ctrl-n` is used to create a new state. States are drawn on the canvas in rows of 5 each with state qi being drawn at row $i/5$, column $i \bmod 5$. This can make for quite a jumble of states and transitions which is of no concern to a non-sighted student. The states can be easily moved to make the DFA more visible to a sighted fellow student or instructor. A selected state is marked as the start state with `ctrl-s` and an accepting state with `ctrl-a`. A transition from state qi to qj is created by first selecting state qi, entering `ctrl-shift-n`, then selecting state qj and entering `ctrl-shift-n` to complete the transition. A self-transition on a selected state is created by entering `ctrl-shift-n` twice. `Ctrl-r` is used to remove states and transitions.

Keystroke	Comment	Accessible description
`ctrl-n`	q0 drawn at position (0,0)	"State q0 is selected."
`ctrl-s`	Incoming arrow drawn to q0	"State q0 is selected. It is the start state"
`ctrl-n`	q1 drawn at position (0,1)	"State q1 is selected."
`ctrl-a`	Ring drawn around q1	"State q1 is selected. It is an accepting state"
`ctrl-shift-n`	Starts drawing transition from q1	""
`ctrl-shift-n`	Self-transition drawn to q1; focus in textfield	"q1 self-loops on the symbols:"
`b`	b entered in transition textfield	"q1 self-loops on the symbols: b."
`enter`	q1 is highlighted	"State q1 is selected. It is an accepting state"
`ctrl-shift-n`	Starts drawing transition from q1	""
↓	q0 is highlighted	"State q0 is selected. It is the start state. Currently creating a transition starting from state q1."
`ctrl-shift-n`	Transition drawn from q1 to q0; focus in textfield	"q1 transitions to state q0 on the symbols: "
`a`	a entered in transition textfield	"q1 transitions to state q0 on the symbols: a."

Table 2: Accessible Interface Example

4.3 Entering State Conditions

`Ctrl-g` is used to move from the selected state and its condition and vice-versa. `Ctrl-o` is used to popup the function choices; the up/down arrows are used to move between them and the space bar is used to select one of them. `Ctrl-m` toggles between Function Mode and Regular Expression Mode.

4.4 Checking the DFA for Correctness

Entering `alt-c` followed by `alt-s` initiates checking of the DFA. The contents of the dialog box that alerts students to errors or success is read aloud by JAWS.

Table 2 gives an example of using the keyboard to create part of the DFA in Figure 2 together with the audible feedback.

5. FUTURE DIRECTIONS

To make ProofChecker more accessible to color blind students and those with low vision, colors and font sizes will be made user-configurable thus allowing them to customize the program to meet their needs [1]. Key bindings will also be made user configurable since blind users may want to customize these as well [7]. Context-sensitive help and audible cues as well as the ability to simulate the processing of a single string are being added as well. To further aid low vision students, conversion of the GUI from Swing to SVG (Scalable Vectorized Graphics) [2] to make magnification of the DFA possible is being considered.

A version of ProofChecker which allows students to input context-free grammars and define the language of each variable has been beta-tested and will be available soon. The extension of ProofChecker to other forms of automata, such as NFA's and PDA's, is also being considered.

We believe the continued development of ProofChecker has much to offer automata theory students, sighted and non-sighted alike, and that the ideas involved may aid the visually impaired community at large.

6. ACKNOWLEDGMENTS

The authors would like to thank X.Y. Li for his help in introducing ProofChecker to the classroom.

7. REFERENCES

[1] E. Bergman and E. Johnson. Towards accessible human-computer interaction. *Advances in human-computer interaction (vol. 5)*, pages 87–113, 1995.

[2] Accessibility features of svg. Available at `http://www.w3.org/TR/SVG-access/`.

[3] R. F. Cohen, A. V. Fairley, D. Gerry, and G. R. Lima. Accessibility in introductory computer science. In *Proc. 36th SIGCSE Tech. Symp. on Computer Science Education*, pages 17–21, 2005.

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

[5] Package javax.accessibility. Available at `http://java.sun.com/j2se/1.5.0/docs/api/javax/accessibility/package-summary.html`.

[6] Jaws® for Windows®. Available at `http://www.freedomscientific.com/fs_products/software_jaws.asp`.

[7] S. H. Kurniawan and A. G. Sutcliffe. Mental models of blind users in the windows environment. In *ICCHP '02: Proceedings of the 8th International Conference on Computers Helping People with Special Needs*, pages 568–574, London, UK, 2002. Springer-Verlag.

[8] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1998.

[9] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, 2003.

[10] S. D. Network. Java access bridge. Available at `http://java.sun.com/products/accessbridge/`.

[11] S. H. Rodger, B. Bressler, T. Finley, and S. Reading. Turning automata theory into a hands-on course. In *Proc. 37th SIGCSE Tech. Symp. on Computer Science Education*, pages 379–383, 2006.

[12] S. H. Rodger and T. W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett, 2006.

[13] T. A. Sudkamp. *An Introduction to the Theory of Computer Science Languages and Machines*. Addison-Wesley, 2006.

[14] T. M. White and T. P. Way. jFAST: a Java finite automata simulator. *SIGCSE Bull.*, 38(1):384–388, 2006.