# Programming and Proofs, Teaching Logic in Computer Science *

Matthias F. Stallmann

February 2001

An important gap that needs to be bridged when it comes to the role of logic in computing is that of educating computer scientists, particularly undergraduates, on the important connections between proofs and programming. In most undergraduate curricula, students are exposed to proofs primarily in a discrete math course that, while emphasizing concepts related to computer science, is not integrated with experiences the students regard as practical. A deeper exposure to logic may occur in an automata theory or programming language theory course. None of these courses are likely to have a major laboratory component. Exceptions fall into three categories:

1. Use of simulators of various automata (examples abound, but among the more notable is an early nondeterministic-pushdown-automaton simulator [1]).
2. Study of programming languages that require logical sophistication, such as Prolog or ML.
3. Software to guide the process of outlining and writing proofs (a unique example of this is a MacIntosh program by Dan Velleman, also the author of [7]).

The main drawback of these laboratory experiences is that they tend to encourage brute-force trial-and-error on the part of the student. Exercises are either so simple that trial-and-error always succeeds or complex to the point where use of the tool or language becomes a distraction from learning the underlying logic.

**Proofs and programs.** An exploration of the relationship between proofs and the activity of programming was recently undertaken by Hayashi et al. [3]. The authors advocate and formalize the concept that proofs, like programs, can be tested by "executing" them on examples. Why is this important, or more specifically, how can this represent a move *away from* the encouragement of trial-and-error?

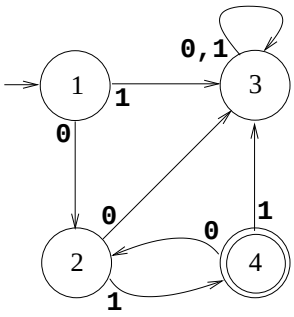
The key difference is that in the traditional world of simulation (programming), the student constructs the automaton (program) and attempts to verify its correctness by testing. When proofs are interpreted as programs, it is the *proof of correctness* of the automaton (program) that is checked by the student — the student has to supply the proof instead of relying on testing alone.

The prototype proof animation system of Hayashi et al. [3] operates in the world of abstract mathematics and appears to be too tedious for use in undergraduate computer science classes. A common fallacy among computer science undergraduates,[1] that proofs are magic incantations for conjuring up truth (and therefore independent of whether or not the thing to be proved is true), is more likely to be encouraged rather than discouraged by such a system.

However, Hayashi et al. make an excellent case for an effort by the author already underway. The correctness of a deterministic finite automaton can be proved (inductively) by making an assertion about the set of strings leading to each state.

*An excerpt from a 2001 proposal to NSF.

[1] Or, apparent fallacy — this observation is based on the behavior of students rather than on what they say




| | |
|---|---|
| **State 1** | `emptyString()` |
| **State 2** | `endsWith("0") && odd(length())` |
| **State 3** | `numberOf("00") > 0 && numberOf("11") > 0` |
| **State 4** | `startsWith("0") && endsWith("1") && numberOf("00") == 0 && numberOf("11") == 0` |

Figure 1: An automaton with assertions attached to the states — initial attempt.

**A dfa proof checker.** For example, consider the automaton pictured in Figure 1. The prototype tool[2] allows the student to draw a dfa and make simple assertions about the states using Java syntax (shown in the boxes below the dfa). In this case the assignment is to design a dfa for the language described by the regular expression $0(10)^*1$. The assertions shown in Figure 1 were a first attempt by a student testing the prototype. When the student asks the tool to "check states", an error message appears.

```
The string "1" does not satisfy the assertion for state 3:
numberOf("00") > 0 || numberOf("11") > 0
Continue checking?                                          YES   NO
```

Continued checking would reveal that other strings beginning with `1` run into the same problem, so the student might revise the assertion to say `numberOf("00") > 0 || numberOf("11") > 0 || startsWith("1")`. Another error message that would eventually arise is the following.

```
endsWith("0") && odd(length()) is true for strings in more than one state
string = 000, states = 2,3
Continue checking?                                          YES   NO
```

The string `000` correctly matches the assertion for state 3 (where it ends up), but also matches the assertion for state 2. A friendlier version would suggest that the student might want to tighten up the assertion for state 1 to exclude that particular string and even give some hints for doing so. Correct assertions for the example are given in Figure 2. In order to achieve error-free "execution" of the automaton with assertions, the student is forced to think logically about the problem at hand. The descriptions for the states must be mutually exclusive, must encompass all possible strings, and must be consistent with the transitions.

The current prototype is able to execute the automaton with all possible strings (using the two-symbol alphabet) of length up to 10 or so (depending on machine speed), which is good enough to ferret out most logic errors likely to occur in an undergraduate theory course.

| | |
|---|---|
| **State 1** | `emptyString()` |
| **State 2** | `startsWith("0") && endsWith("0") && numberOf("00") == 0 && numberOf("11") == 0` |
| **State 3** | `numberOf("00") > 0 \|\| numberOf("11") > 0 \|\| startsWith("1")` |
| **State 4** | `startsWith("0") && endsWith("1") && numberOf("00") == 0 && numberOf("11") == 0` |

Figure 2: Correct assertions for the DFA in Figure 1.

[2]Currently implemented as an add-on to JFLAP [5, 6] and called ProofChecker

In order to deal with larger examples, or to provide the student with more tutorial help, the techniques of constraint satisfaction or direct use of sat solvers are likely to come into play. The examples from automata theory may become a source of benchmark problems for sat solvers, or they may simply become a motivation for their development. In either case, both activities will benefit.

The next logical step is to extend the idea of assertions for dfa states to proofs of correctness of context-free grammars. Here, the assertions describe the strings generated by each variable in the grammar and the corresponding proof checker must analyze the parsing process (or perform systematic brute-force generation of strings).

Recent work by Susan Rodger and her students (see, e.g., [2, 4]), though not directly relevant, represents a comprehensive effort to use interactive animation for all the important topics in automata theory. We are not aware, however, of any careful studies to elicit the extent to which such efforts enhance understanding. Such studies would be too ambitious for a small grant, but may be worth doing when several competing tools and techniques have been refined and some simple pedagogical goals are defined.

## References


[1] D. Caugherty and S. H. Rodger. NPDA: A tool for visualizing and simulating nondeterministic pushdown automata. In N. Dean and G. E. Shannon, editors, *Computational Support for Discrete Mathematics*, volume 15 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 365–377. American Mathematical Society, 1994.

[2] E. Gramond and S. H. Rodger. Using JFLAP to interact with theorems in automata theory. In *Proc. 30th SIGCSE Tech. Symp. on Computer Science Education*, pages 336–340, 1999.

[3] S. Hayashi, R. Sumitomo, and K. Shii. Towards the animation of proofs — testing proofs by examples. *Theoretical Computer Science*, 272:177–195, 2002.

[4] T. Hung and S. H. Rodger. Increasing visualization and interaction in the automata theory course. In *Proc. 31st SIGCSE Tech. Symp. on Computer Science Education*, pages 6–10, 2000.

[5] M. Procopiuc, O. Procopiuc, and S. H. Rodger. Visualization and interation in the computer science formal languages course with JFLAP. In *Frontiers in Education*, pages 121–125, Salt Lake City, Utah, 1996.

[6] Susan H. Rodger. JFLAP 3.1. `http://www.cs.duke.edu/ř̆odger/tools/jflap/index.html`.

[7] Daniel J. Velleman. *How To Prove It: A Structured Approach.* Cambridge University Press, 1994.