Program Analysis of Educational Hardware-Description Language

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

James Leo Yuan Huang
Spring, 2022

Rosanne Vrugtman, Department of Computer Science

Program Analysis of Educational Hardware-Description Language

CS4991 Capstone Report, 2022
James Huang
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
jh7qbe@virginia.edu

Abstract
Traditional program-testing methods can reveal a program's incorrectness but not its correctness. I developed a program-analysis tool to analyze the correctness of student programs written in HCLRS, an educational hardware-description language. Though the tool produced useful output for some assignments, suggesting its potential use in an auto-grading system, it failed to produce useful output for other assignments, and its formal correctness was not investigated. Future work on this tool should address these issues.

1. Introduction

A common way to test a program's correctness is to feed it some inputs and make sure it produces the expected outputs. The more 'test cases' a program can demonstrate it correctly handles, the more confident the tester can be that the program is correct. One problem with this 'test case' testing method is that the tester cannot be sure that the program is correct until they have exhaustively tested every possible input, which is impractical for all but the simplest programs. Another problem is that programs do not usually provide much information about *why* they failed a test case when they do so, and so the programmer must fix their incorrect program without knowing much about exactly *how* it is incorrect.

Test-case testing is often used by computer-science courses to evaluate the correctness of student submissions, but these testing setups usually leave a lot to be desired. Information like *what part* of the submission was wrong and *how* it was wrong can only be roughly inferred from even the most meticulously designed test suites. In addition, this “correct is not failing test cases” approach may cause some students to produce submissions that pass test cases but are incorrect otherwise.

Given the downsides of test-case testing, alternative testing methods are potentially valuable. I developed a tool based on such methods, designed to analyze the correctness of student HCLRS-program submissions across eight different assignments. The previous HCLRS auto-grading system used test-case testing and so suffered from the aforementioned issues, mainly that the system could not provide any useful information beyond 'X test failed'. I approached the problem with a very broad 'static program analysis' approach which eventually narrowed to a more 'symbolic execution' approach.

2. Review of Research

The tool I made takes inspiration from the general idea of 'symbolic execution', the idea of executing a program with abstract 'symbols' instead of concrete values to collect information about the program. Other than this, the tool is not very relevant among current symbolic-execution research and is more an application of the general concept to the specific task of grading HCLRS assignments. In particular, much research seems to be on tracking program state through lines of imperative code to verify that certain program states can or cannot occur, as explained in Baldoni, et. al. (2018) [1]; in comparison, HCLRS code is relatively declarative, and in terms of grading, an explanation of *how* a program failed/ succeeded is much more interesting than simply *if* it failed/succeeded—one of the goals of this tool was to provide the feedback that the preexisting test-case-testing auto-grader could not.

3. Process Design

The tool was developed incrementally under the guidance of an advisor over the span of 14 weeks. In weekly meetings, the advisor and I came up with goals for the next week based on the progress made in the previous week. The project proceeded as follows:

3.1. Week 1

Roughly prototyped a basic analysis tool. At this point I only knew I was going to make some kind of 'program analysis' tool, and the thought to base it on specifically 'symbolic analysis' had not yet emerged. This week's rough prototype was based on the idea of checking whether two 'parts' of the processor described by the given HCLRS program were connected.

3.2. Week 2

Improved week 1's tool. Week 2's revision notably began to introduce the concept of 'writing tests' that the tool could then evaluate given an HCLRS program.

3.3. Week 3

Rewrote week 2's tool (written in TypeScript) in Haskell. This rewrite helped me understand what exactly I was trying to do. It also introduced the symbolic-execution-flavored concept of simplifying programs' parse trees into intermediate representations and determining their equivalence, an idea that would last until the end of development.

3.4. Week 4

Designed and implemented a test-specification language for the tool. Designing and refining a language specifically for writing tests would help to formalize what the tool was exactly supposed to do, which I was still unsure of.

3.5. Week 5

Dealt with correctness issues. The tool had, until this week, classified submissions into three main categories of correctness: *correct*, *incorrect*, and *impossible to evaluate*. Given an expression to simplify, a submission was *correct* if what it produced matched the expected simplified value in a series of hard-coded rules accounting for a number of equalities, e.g. `x + y = y + x`. It was *impossible to simplify* if there was some error in execution, and *incorrect* otherwise. I soon realized that there were cases where a program neither failed to execute nor matched a hard-coded correctness rule. For example, I had not thought to program in a `x - y = x + (-y)` rule, and so the tool would mark a submission that had produced `x + (-y)` instead of the expected `x - y` as *incorrect*. My solution to this was to mark incorrectness in the same way as correctness: by matching specific, hard-coded rules. And if the tool encountered something it had no rules for, e.g. `x - y = x + (-y)`, it would place it in a new category of correctness termed *wrong-maybe*.

3.6. Weeks 6-10

Test-writing, testing, and a few feature adjustments. These weeks were a routine cycle of writing tests to test submissions for HCLRS assignments, making sure these tests were working as expected by comparing their results to the previous auto-grading system, and adding/adjusting features as necessary.

3.7. Week 11

Rewrote tool in Rust. I discovered a performance issue in week 10 that I decided to solve by rewriting the whole tool in a different language, taking the opportunity to revise and refine a few ideas along the way.

3.8. Weeks 12-14

Continued refinement. Nearing the end, I became more concerned with polishing what I already had rather than trying larger changes. These weeks I mostly spent wrapping up, documenting, and reflecting on the work I had done, ready to leave my problems to future work.

4. Results

The fruit of these 14 weeks was an HCLRS auto-grading tool capable of analyzing four of the eight assignments graded by the previous auto-grading system. Compared to the previous system, though, it offers more detailed and relevant feedback and avoids many double-jeopardy situations.

One extreme example I came across was a submission that failed 113/116 of the previous system's tests but only 3/167 of the new tool's conditions. It turned out that the submission had a small error in a basic functionality that the previous system's tests relied on heavily, placing it in double-jeopardy and causing it to fail many tests. By comparison, the new tool was able to test functionalities relatively independent of each other and so correctly identified that most of the submission was correct.

The tool's nature of analyzing the structure of a program more than its behavior on some concrete values also lends itself to providing more relevant feedback. The old system could only show how the program *behaved* incorrectly given some inputs; the new tool can show how the *structure* of the program is incorrect given some conditions, depicting more clearly how the program was written incorrectly in the first place.

5. Conclusion

The developed tool demonstrates that auto-grading tools based on program-analysis methods may be able to offer more insightful feedback than those based on traditional test-case testing. By evaluating and testing against symbolic rather than concrete values, the HCLRS auto-grading tool was able to provide not only more-thorough guarantees of program correctness but also useful evaluations of incorrect programs that could help programmers fix their issues. Though narrow in scope, I hope this work could inspire and stimulate future developments in auto-grading and educational programming-language tools.

6. Future Work

The tool only managed to grade four of the eight assignments. It likely can grade assignments five and seven without substantial modification; the main challenge would simply be to write the tests. Assignments six and eight might have some trickier bits that require more tool functionality. Six and eight also suffer, in particular, from a performance issue in which “unknown” values are repeatedly simplified unnecessarily.

The formal correctness of the tool is one of its biggest weaknesses. Whether the simplification and equivalence rules programmed into the tool are mathematically correct and will thus always produce correct results is an important issue that I did not address.

Finally, the tool was developed incrementally and thus has many inelegant, clunky features that exist simply to “get the job done.” For example, to test the functionality of arithmetic flags, I introduced a “register-matching” feature to detect specifically a zero-flag and sign-flag and wrote tests that manually tested each combination of these flags. A more elegant, ideal solution would test these flags' behavior without relying on their internal implementation. Developing better abstractions to clean up these kinds of features would greatly help the tool.

References

[1] Baldoni, R., Coppa, E., D’elia, D. C., Demetrescu, C., & Finocchi, I. (2018). A Survey of Symbolic Execution Techniques. ACM Comput. Surv., 51(3). https://doi.org/10.1145/3182657