The Use of Automated Provers to Provide Stronger Guarantees About the Correctness of Code in Safety Critical Applications

(Technical Topic)

Understanding the Transition and Acceptance of Formal Software Verification Methods

(STS Topic)

A Thesis Project Prospectus
In STS 4500
Presented to
The Faculty of the
School of Engineering and Applied Science
University of Virginia
In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science in Computer Science

By
Garrett Burroughs

April 2, 2024

Advisors

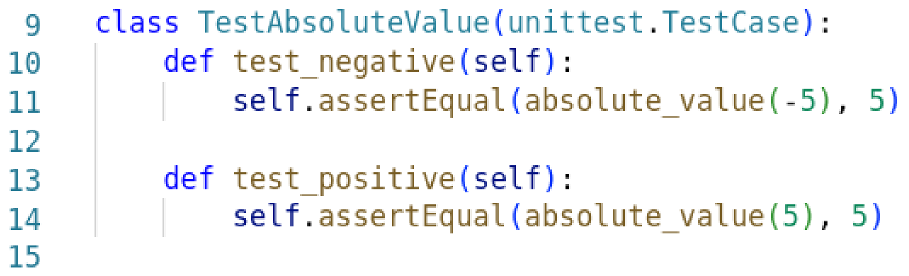
Kathryn A. Neeley, Department of Engineering and Society

Matthew B. Dwyer, Department of Computer Science

Introduction

When it comes to creating software, writing code that performs the required task is only a small portion of the development process. A significant amount of time and development cost is spent in the testing and verification of the software, and as software complexity continues to increase, so will the amount of testing that is required (Dustin, 2009, Foreword). As time goes on, developers aim to create new strategies that make the testing process easier and more efficient, with unit testing libraries (eg. JUnit, NUnit) currently being the most used testing strategy according to a survey of canadian developers (Garousi 2013, p. 24).

Continuous testing, regression testing, and the periodic application of verification and validation analysis have been identified as the primary means to control software technical quality (Barbareschi, 2022, p. 1079). These testing strategies rely on the developer providing a sample input, as well as the expected output of the program, and then testing to see if the program acts as intended. For example, in order to test a program that is supposed to compute the absolute value, a test case for such a program might be that the program should output 5 when the input is -5. An example of two tests written for such a program in python can be seen in figure 1.



```python
class TestAbsoluteValue(unittest.TestCase):
    def test_negative(self):
        self.assertEqual(absolute_value(-5), 5)

    def test_positive(self):
        self.assertEqual(absolute_value(5), 5)

```

Figure 1. Automated tests that test the functionality of an absolute value program. In this test, it is expected that the absolute value of -5 will be 5, and the absolute value of 5 will be 5. If these conditions match, the test is considered to have passed.

The test inputs are provided to the program, and the program's output is compared to the expected output. While this testing methodology can be very useful, it can only verify that the program works for the provided test inputs, and only indicates that no problems were found when testing. For example, if the above tests pass, there is no guarantee that the program will give 6 for the input of -6.

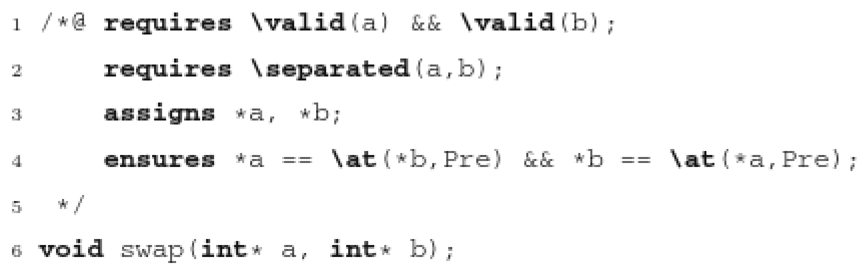
Despite the inherent limitations of automated testing in software, it has been a useful tool for many years. With the advent of AI powered developer tools such as ChatGPT and Github Copilot, the landscape is rapidly changing. These tools are powerful in terms of speeding up the software development process however, they often produce code that seems correct, and may work for some inputs, but upon closer inspection contains various vulnerabilities and bugs (Vaidya, 2023, p. 38). Due to the nature of the bugs in AI generated code, they may pass test suites that do not cover a wide enough range of test inputs. The technical portion of this report aims to further examine the problems with current testing strategies, as well as offer a potential solution in the form of formal proof verification. The STS portion of this prospectus will examine the adoption of such a verification method, and draw on the areas of social psychology to better understand how such a change might be implemented.

The Use of Automated Provers to Provide Stronger Guarantees About the Correctness of Code in Safety Critical Applications

As time has gone on, software has become more present in our day to day lives. Software systems have found themselves in simple daily tasks such as calculating a navigation route, all the way to life altering events like diagnosing cancer patients (Chazette et al., 2022, p. 457). As software continues to reach its way into more safety critical applications, such as aviation control, autonomous vehicles, and the medical profession, it is crucial that the software does not contain any faults. Currently it is estimated that software failures have likely cost the U.S. economy at least $25 billion and maybe as much as $75 billion (Charette, 2005, p. 45). When considering safety critical operations, the cost of software failure goes from dollar amounts to human lives. For this reason, there are already rigorous requirements put in place for software quality in safety critical systems.

For example, the FAA provides many guidelines on the review process that software software used in airborne applications needs to go to; however, one of the main technical metrics is the amount of code coverage that the testing provides (FAA, 2018, p. 4-2). Code coverage measures how much of the code in the program was executed while running the test. While high code coverage can indicate a step in the right direction when writing tests, it only indicates the amount of the system that is being tested, and says nothing about the quality of the tests being run. One study found that "...even if the test suite satisfies a 100% code coverage, using all of the five mentioned criteria, 7% to 35% of the faults may still be undetected" (Hemmati, 2015, p. 151). Even more concerning, a survey of developers found that "Developers claim to write unit tests systematically and to measure code coverage, but do not have a clear priority of what makes an individual test good" (Daka, 2014, p. 208). This highlights one of the main issues with current testing practices, in that they are almost never able to indicate whether the program is working as intended, but only that they failed to find any defects.

An alternate approach to automated testing, is the formal verification of programs. A function contract specifies limitations on the function input, while also making guarantees about the function's output. A function is the smallest building block of a computer program, and multiple functions are combined in order to create the full functionality of a software system. If function contracts are defined using a formal logic system, then it is possible to attempt a rigorous proof, whereupon conclusion, you can be certain that the function meets its contract given any underlying assumptions about the hardware system hold true (Kirchner et al., 2015, p. 579). This contrasts the automated testing approach, where the formal verification is able to indicate that the code written does match its intended function. An example of such a contract can be seen in Figure 2. The specification of what it means to swap two values, as well as the required conditions are all laid out in the ASCL formal specification language. If verified, then it can be concluded that this function is correct for all valid inputs a and b, as opposed to an automated testing approach which would only allow for a finite set of memory addresses to be tested.



```
/*@ requires \valid(a) && \valid(b);
    requires \separated(a,b);
    assigns *a, *b;
    ensures *a == \at(*b,Pre) && *b == \at(*a,Pre);
 */
void swap(int* a, int* b);
```

Figure 2. **A C function annotated with a formal function contract written in ASCL.** By providing an unambiguous specification, a formal proof can ensure that the code correctly matches the specification, and therefore carries out its intended function. (Kirchner et al., 2015, p. 577)

Historically, formal proof needed to be carried out by hand, causing it to be a time and labor intensive task. More recently, static analysis tools such as Frama-C have improved to the point where a large amount of the proof process can be offloaded to automatic proovers (Blanchard, 2020, p. 8). Without the need for humans to carry out these proof tasks, formal verification methods are no longer a time and labor intensive task, giving the benefits of a much stronger guarantee about the program for all acceptable inputs.

While this technology is very powerful, and solves many of the problems that exist with existing testing methods, the capabilities are still being developed, and there are not many examples of it being used in larger codebases. Throughout my technical research, I plan to conduct a case study of formal verification methods being used on a moderately sized code base with about 1000 lines of code, to develop strategies for writing formal specifications based on the code that needs to be verified, as well as document the strengths and weaknesses of using formal methods on non-trivial use cases. This case study will serve as an example of how to adopt these methods into already existing codebases.

Understanding the Transition and Acceptance of Formal Software Verification Methods

The formal verification of software systems provides many benefits over current verification methods, however, the techniques are yet to be universally adopted. As with any system, there are costs associated with the implementation. As it stands currently, there are many software projects which employ no verification at all, with the mindset that the application is not critical enough to warrant the extra time spent writing tests to ensure quality of code. An example of this is the Cortext library (Hamil, 2020), which is a dependency of React (Facebook, 2024), a widely used web technology with over 23 million weekly downloads. When examining the source code of Cortex, it does not contain any testing strategies. The main costs when considering software verification are the time that it takes to carry out the verification, and the cost to the entity developing the software. Figure 3 shows the overall effect of the use of formal techniques on time, cost, and quality aggregated from a survey of software teams which adopted formal verification methods (Woodcock et al., 2009, p.19:8).

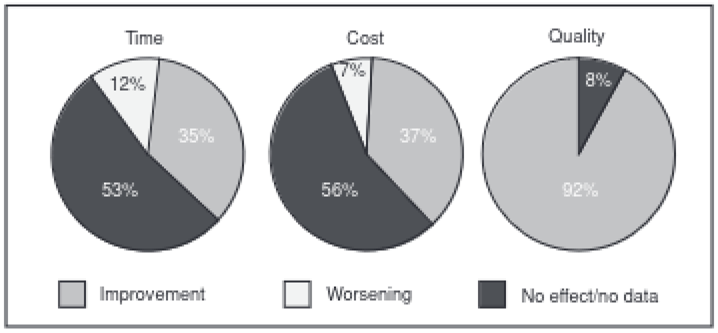


Figure 3. The effect of formal techniques on time, cost, and quality Results from a survey asking the question "Did the use of formal techniques have an effect on time, cost, and quality". Despite an overwhelming indication that formal methods are the same or better than existing solutions, they are still not widely adopted. (Woodcock et al., 2009, p.19:8).

While the majority of respondents couldn't tell if the adoption of the formal methods had an impact on the time or cost to develop software, the responses that indicated improvement far outweighed the responses that indicated worsening in both categories. In regards to code quality it is clear that an overwhelming majority indicated that formal methods led to an improvement.

With data to show that the main concerns of adoption are ill founded, it is unclear why there has not been a larger move towards the use of formal verification methods. One likely reason is that the current verification methods are widely used and rooted in the software development process, with developers having strong views on what the best practices are, as well as organizations releasing guidelines and requirements for testing. Unit and functional/ system testing are currently the two most common test types, with almost 200 out of about 300 respondents reporting that they use unit testing (Garousi 2013, p. 18). Drawing from the field of social psychology, a certain proportion of a population is needed in order to shift the social norms. This is called the social tipping point. As seen in figure 4, past a certain percentage, a committed minority is able to change the views of the group as a whole (Centola et al., 2018, p. 1116).

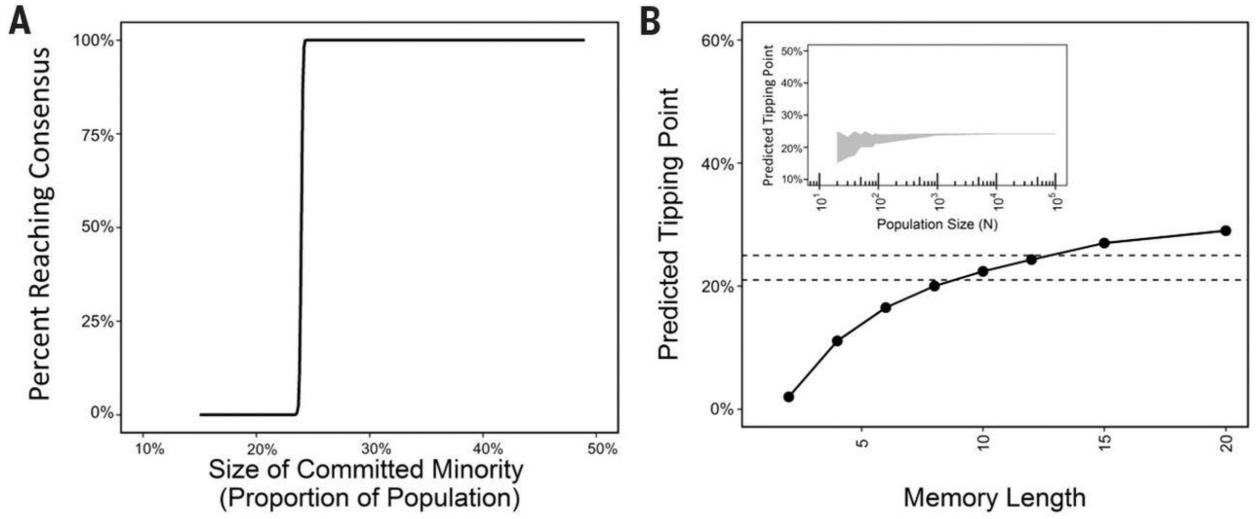


Figure 4. Predicted tipping points in social stability. It only takes a small portion of a committed population to have effects on the group as a whole, however, if below this point, consensus tends to stay very low (Centola et al., 2018, p. 1116).

I plan to build upon this research in social psychology to gain a greater understanding of how the idea of using formal verification methods might propagate through the software engineering ecosystem. Due to current methods already being ingrained into both the social norms of the software development process, as well as many companies such as salesforce having requirements such as 85% code coverage to be deployed (Salesforce, 2022), a significant shift in verification methodology has to take place. According to figure 3, due to the large population of software engineers, and long standing practices, it is likely that there will need to be a committed majority of at least 30% of developers adopting these new practices to propagate throughout the population.

Similarly, these requirements need to be implemented by organizations that pass regulations on software quality, such as the FAA outlining software requirements in airborne software systems in ORDER 8110.49A (FAA, 2018, p. 4-2). I plan to take a TOC model of analysis towards this, as there are many different technical, organizational, and cultural ideas at play. Breaking down the interactions between the different actors may provide further insight into how greater support can be garnered, and new regulations can be implemented where it matters most.

Conclusion

Through the technical work of my project, I hope to gain a better understanding of automated verification systems such as Frama-C to provide a way for software engineers to speed up the development process without having to compromise on the quality of the software. If the technical deliverable is successful, developers will have a better understanding of how to adopt formal methods within their software systems. Throughout my STS research I hope to gain a better understanding of how we might transition from the current methods of testing to more rigorous formal verification methods. If the STS deliverable is successful, it will aid in a successful transition from automated testing to formal verification. (1865 words)

References

Barbareschi, M., Barone, S., Carbone, R., & Casola, V. (2022, December 1). Scrum for safety: an agile methodology for safety-critical software systems. Software Quality Journal, 30(4), 1067 - 1088.

Blanchard, A. (2020, July 1). *A gentle introduction to C code verification using the Frama-C platform*. Zeste De Savoir.

Centola, D., Becker, J., Brackbill, D., & Baronchelli, A. (2018). Experimental evidence for tipping points in social convention. *Science*, *360*(6393), 1116–1119.

Charette, R. (2005, September 6). Why software fails [software failure]. *IEEE. IEEE Spectrum*, 42(9), 42-49.

Chazette, L., Brunotte, W., & Speith, T. (2022, December 1). Explainable software systems: from requirements analysis to system evaluation. *Requirements Engineering*, 27(4), 457 - 487.

Dailler, S., Hauzar, D., Marché, C., & Moy, Y. (2018). Instrumenting a weakest precondition calculus for counterexample generation. *Journal of Logical and Algebraic Methods in Programming*, *99*, 97–113.

Dalal, S., & Chhillar, R. S. (2012). Case studies of most common and severe types of software system failure. *International Journal of Advanced Research in Computer Science and Software Engineering*, *2*(8).

Daka, E., & Fraser, G. (2014, November). A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering* (pp. 201-211). IEEE.

Dustin, E., Rashka, J., & Paul, J. (1999). *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional.

Facebook. (2024, April 26). *React*. NPM. https://www.npmjs.com/package/react

Federal Aviation Administration (FAA). (2018, March 29). ORDER 8110.49A, Software Approval Guidelines. U.S. Department of Transportation

Garousi, V., & Zhi, J. (2013). A survey of software testing practices in Canada. *Journal of Systems and Software*, *86*(5), 1354-1376.

Hamill A. (2020, February 7). *Cortex*. Github. https://github.com/arhamill/cortex

Hemmati, H. (2015). How effective are code coverage criteria?. *2015 IEEE International Conference on Software Quality, Reliability and Security*. 151-156.

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles J., & Yakobowski B., (2015). Frama-C: A software analysis perspective. *Formal Aspects of Computing*, *27*(3), 573–609.

Milkoreit, M. (2022). Social tipping points everywhere?—patterns and risks of overuse. WIREs Climate Change, 14(2).

Nylin, W. (2009). Foreword. Dustin, E., Garrett, T., & Gauf, B. (2009). *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education.

Salesforce. (2022, October 13). *Instructions to test Apex code.* Salesforce. https://help.salesforce.com/s/articleView?id=000385650&type=1

Vaidya, J., & Asif, H. (2023). A critical look at AI-generated software: coding with the new AI tools is both irresistible and dangerous. *Ieee Spectrum*, *60*(7), 34-39.

Woodcock, J., Larsen P., Bicarregui, J., Fitzgerald, J. (2009, October). Formal methods: practice and experience. *ACM Computing Surveys* Association for Computing Machinery, 41(4)