ASE Reflection Paper: From Formal Verification To Test Generation

MINSUNG CHO, Northeastern University, USA

As software becomes increasingly complex, the assurance that software is correct becomes ever more important. Computer scientists have studied the problem of software correctness through two primary lenses: *formal verification* and *software testing*. This reflection papers details my attempt at understanding the landscape and development of software testing techniques as a researcher in verification, and how the two approaches have influenced one another.

1 INTRODUCTION

As the need for correct software increases, two approaches to writing correct software became prominent research directions.

(1) *Formal verification* works with mathematical abstractions of programming language behavior to provide a mathematical proof that software behaves correctly.
(2) *Software testing* aims to validate correct program output under certain inputs to provide empirical confidence that software behaves correctly.

In [Parnas 1985], David Parnas argues that formal verification is not the correct approach. His argument has two main points: (1) the size of software (in lines of code) that can be verified is small compared to the size that can be robustly tested and (2) there are some programming languages that lack semantic models that capture all the behavior that a programmer may want to verify.

However, this article is nearly 40 years old! Formal verification has developed a lot since then. Although it's true that many verification tasks are computationally hard, the verification tools we have today can verify entire compilers [Kiam Tan et al. 2019] and microprocessors [Hardin 2010], as well as prove powerful theorems in pure mathematics. Although the fact that some languages lack an all-encompassing semantic model is still true today[1], the fact that testing can detect bugs about, say, concurrency, fairly reliably has not stopped some from falling through the cracks [Gates 2023].

My primary experience as a computer science researcher is in the field of formal methods, of which verification is one of the primary applications. As such, my ill-informed perspective on software testing is biased at best, and skeptical at worst. This reflection paper details my journey into the world of software testing–specfically, software test generation. I first ease into test generation by looking at an application of formal verification, specifically software model checking, to test generation. Then I explore property-based and metamorphic test generation techniques as a weaker application of formal verification. Lastly, I look at KLEE, a symbolic engine for test generation, as a marriage of the two techniques addressed in previous sections.

Disclaimer. For the duration of this paper I work with the definition of software test given by [Fink and Bishop 1997], which is "a set of executions of a given program using different input data for each execution". As such, I don't talk much about testing that syntactically changes the program code (e.g. mutation testing).

[1] whether or not this is a worthwhile research endeavor in programming languages is something I have hot takes on beyond the scope of this paper.

Author's address: Minsung Cho, Northeastern University, Boston, Massachusetts, USA, minsung@ccs.neu.edu.

2 FORMAL VERIFICATION TO GENERATE TESTS

In 2002, Henzinger, Jhala, Majumdar, and Sutre published a seminal paper [Henzinger et al. 2002] that describes an *abstract-check-refine* loop to verify properties of `C` programs. Without getting too far into the details, the loop does the following:

- *Abstract*: both the program and the safety properties we want to check are abstractly represented in mathematical language.
- *Check*: The abstracted program is checked if it satisfies the abstracted safety property. The way we do this is the bulk of the technical material of [Henzinger et al. 2002] and beyond the scope of this reflection paper. If we do satisfy the property, then all is well in the world and we terminate the loop.
- *Refine*: If the abstracted program does not satisfy the property, we generate an abstract counterexample to our safety property. If this counterexample corresponds to a valid program path, then our program violates the safety property. If not, we need to to add additional predicates in our abstractions to account for this unrealizable counterexample and run the loop again.

This loop seems super powerful! Thousands of papers have experimented on extending the features of this algorithm, making it more efficient, and adapting it to more programming languages and abstractions. One such paper is [Beyer et al. 2004], where the authors use this loop to generate test suites.

The key insight in [Beyer et al. 2004] is that each abstract counterexample that corresponds to a valid program execution path can be translated into a concrete program test that a software engineer can test against. Furthermore, when the program is checked for the safety property in the *check* step, we can additionally generate more examples that can serve as more tests the program should pass. The hope is that these generated test suites are robust enough that with high confidence it **fully realizes the safety properties** even if the original code is modified or extended. Furthermore, this method of test suite generation can be useful when we **do not want to reveal the source code but give strong guarantees of correctness** to a client: we can verify our software, use the verification to generate a test suite to give to a client, who can then test amongst themselves to be assured of software correctness.

So, what are the pros and cons of this method of test case generation? The main pro is the **automatic generation** of test cases with **theoretical guarantees of robustness**. As software model checking quite literally looks at all valid program execution paths, this approach will never miss an edge case. Furthermore, as long as the abstraction is mathematically sound[2], we can reliably translate abstract examples into real program tests.

However, there are several cons. The main cons are actually not within the test generation step itself, but within the *abstract-check-refine* loop the test generation relies on. There are flaws with each step of the loop:

- The *abstract* step suffers from two flaws. One flaw is that the abstraction used is particular to imperative programs (such as `C`) and does not translate very well to other programming paradigms. Furthermore, abstractions have the flaw
- The *check* step is a semidecidable procedure: it can **potentially run forever**. This does not seem like a very desirable property when you want to prove correctness for *any* program. The authors gloss over this point, and do so in subsequent papers, which is a major flaw in this type of loop.

[2] in a technical sense

- The *refine* step is, fortunately, decidable. However, depending on what logical framework we are working in, SMT queries **can be very slow**, to the order of doubly-exponential time. Also, if the counterexample fails to be a valid program trace, the heuristics to design what new predicates should be added are empirical and, in particular, are **not universal**.

Conclusions. Formal verification is an extremely strong and desired tool. Knowing that your software is provably correct is the strongest possible guarantee, and generating robust test suites as a consequence of this guarantee is a great perk. However, as this test generation method relies on formal verification, it falls victim to the same problems of formal verification: the lack of a fully abstract model of all possible program behaviors and the computational inefficiency.

3 PROPERTY-BASED AND METAMORPHIC TESTING: A MIDDLE GROUND?

We learned from the previous section that relying on formal verification to generate our tests suffers from some serious issues. Where can we relax our approach to avoid the same issues that full-on formal verification face?

One place we can relax our approach is full automation. Instead of having our test suite be fully automatically generated, [Fink and Bishop 1997] proposes a hybrid method, in which a software engineer serves as an oracle throughout the process. To aid our oracle, [Fink and Bishop 1997] gives our *abstract-check-refine* loop new tools to firmly ground the high level of abstraction into a framework a software engineer can work with.

- The *abstract* step now has a new specification language to specify safety properties to lower the level of abstraction. Furthermore, program *slices*, or subprograms, are generated with respect to our property that the oracle can provide a first pass on how important certain program slices are to test.
- The *check* step still generates tests based on the program specifications. But, whereas [Beyer et al. 2004] needed to translate abstract counterexamples to actual program variable assignments, this paper skips this step and uses the method of *iterative contexts* to sequentially generate symbolic (input) variable assignments. Of course, the oracle can provide input here as well.
- The *refine* step now is a direct comparison between symbolic execution histories of the program and abstract program slices, which identifies gaps in test coverage. Oracle input is necessary here to identify and address these gaps.

Note that [Fink and Bishop 1997] never explicitly mentions the *abstract-check-refine* loop. In fact, Henzinger's paper comes five years after Fink's! Fink, instead, outlines an 8-step process of his test generation and evaluation paradigm. One of the outcomes of my reflection that Fink's 8-step process and Henzinger's 3-step loop serve basically the same purpose: provide an end-to-end outline of how to generate tests, evaluate them, and generate new tests based on coverage.

This testing paradigm provides a middle ground between completely automated testing and completely human testing. It seems to be **the first paper describing an abstraction-guided semi-automated approach to testing**. It benefits from pros borrowing from both approaches. First, since we still work with abstractions of programs, we can potentially detect bugs in edge cases a human may not be able to detect. Second, we have humans evaluating whether or not certain program paths are worth testing or not, which can directly translate to our **tests checking exactly what software engineers need confidence in**. Third, we **overcome the potential nonterminating program** by having manual intervention when necessary.

But, there are still cons! The first main con is that this work **lacks an implementation** for generic programs, and rather mostly describes a case study on a specific program. The authors admit that implementing their testing paradigm with good heuristics is a "nontrivial problem" [Fink

and Bishop 1997] and **leave it to future work to address these heuristic issues**. The other cons look similar to those of [Beyer et al. 2004]. First, the program abstractions they use still suffer from (1) **paradigm-specific restrictions** and (2) **semantic inexpressivity**. Next, although we remove semidecidability from the equation, we still have potential inefficiency. Where does this inefficiency stem from? Mostly, it stems from the fact that the **tests we generate are informed by the structure of the program** that we are testing. At this point, we have failed to separate *concrete tests* that are assignments to program variables giving a certain output by going down a certain program execution path and *symbolic tests* that are constraints on program specifications that make the program behave a certain way[3].

3.1 The original `QuickCheck`

Nowadays, when we think of property-based testing, we think of `QuickCheck`. `QuickCheck` has been adapted to numerous different programming languages, and sees widescale usage in both research and industry. But what did the original implementation of `QuickCheck` [Claessen and Hughes 2000] accomplish?

(1) It **automatically tested programs in Haskell, a functional language.** In

4 SYMBOLIC TESTING

5 CONCLUSIONS

REFERENCES

Dirk Beyer, Adam J Chlipala, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Generating tests from counterexamples. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 326–335.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279.

George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes* 22, 4 (1997), 74–80.

Dominic Gates. 2023. After Alaska Airlines planes bump runway while taking off from Seattle, a scramble to 'pull the plug'. https://www.adn.com/alaska-news/aviation/2023/02/20/after-alaska-airlines-planes-bump-runway-a-scramble-to-pull-the-plug/

David S Hardin. 2010. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer.

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 58–70.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019), e2. https://doi.org/10.1017/S0956796818000229

David Lorge Parnas. 1985. Software Aspects of Strategic Defense Systems. *Commun. ACM* 28, 12 (dec 1985), 1326–1335. https://doi.org/10.1145/214956.214961

[3]Can you tell where I'm going with this?