Testing Formal Methods

Jeffrey M. Voas

Reliable Software Technologies
Suite 250, 21515 Ridgetop Circle, Sterling, VA 20165 USA
Tel: (703) 404-9293
Fax: (703) 404-9295
Email: jmvoas@rstcorp.com

Abstract

If formal methods are correctly applied in each stage of the development process, very high quality code should result. But except for niche application domains, the marketplace is content with buggier code that reaches the marketplace quickly and is not touched by formal methods. I propose a research initiative for how to best marry formal methods with traditional and non-traditional testing approaches, and I give an example of doing so for a COTS components that are wrapped.

Keywords: Components, formal methods, testing, life-cycle, integration

Workshop Goals: To propose that research projects are started that address how to best integrate formal methods and testing during the software engineering life-cycle.

1 Background

My background is in software testing and software fault injection technologies. I am currently working on a team to develop a profile-independent component dependability model that is composable for the US National Institute of Standards and Technology. The approach first involves developing a dependability metric (as opposed to a reliability model) that can stamp a value onto a component that is similar to a reliability estimate (in [0, 1]). From there, we hope to be able to assess a worst-case system-level dependability estimate by analyzing how components are composed in a system and from what their individual dependability scores are.

2 Introduction

Besides the difficulty for the average person to understand how to use formal methods, probably the next greatest criticism against formal methods is the absence of good measures for how much better formal methods make software. Increased quality is desirable; quality must be affordable. Few people would argue that formal methods are detrimental to software quality, but many people argue that for the added cost and difficulty-of-use, formal methods are not practical.

Further, formal methods advocates (or "formal methodists") for years have attacked the software testing community over well-known theoretical and practical weaknesses in various software testing techniques. Software testing suffers from an inability to exhaustively test, reliance on operational profiles and correct oracles, and an infeasibility to test to the higher levels of assurance. For the last decade, these two software engineering camps have fought instead of collaborated. (A classic example of this bickering was the debate between Boris Beizer and the "Cleanroom" advocates back in 1995 which is still waging in the presses [1].) Neither side claims to have the silver bullet, yet there are still those from each camp that seem to enjoy debating more than improving the state-of-the-art. Fortunately, the majority of practitioners seem eager to put aside personal preferences and find common ground.

This position paper addresses the need to fund research that marries both communities into a single community geared toward higher assurance software. In my opinion, formal methods are best suited to the earlier parts of the software engineering life-cycle, while testing is best left toward the end of the software engineering life-cycle. Because of this, the redundant work done by each different approach can be lessened by separating their use according to life-cycle phase and by learning where the key redundancies are. To our knowledge, this is still an open research issue, even though small advances in pre- and post-conditions embedded as testing assertions have been made.

Further, testing can augment the value-added by formal methods by demonstrating that the formal methods used really improved a program's quality. And as I said earlier, there are not good measures for how much better formal methods make software. Formal proofs throughout the life-cycle assert that certain behaviors cannot be exhibited. Such assertions need to be scrutinized. Software testing and several alternate testing methods can provide this scrutiny. By testing software that has been formally developed and proven, you are not just testing the software, but you are also testing the development processes. In this manner, testing provides a means for measuring the quality-gains afforded by formal methods. This measurement can begin to replace the anecdotal evidence that formal methods enable better software with empirical evidence.

3 Time-to-market vs. Quality

Before I propose the research initiative that I have long considered is needed in software engineering, I'd like to discuss why I believe that practitioners dismiss formal methods as academic exercises in mental gymnastics. This is a personal opinion, but I believe it to be accurate.

Even the most dedicated formal methods zealot would admit that formal methods are a discipline where the divergence between "academic" and "practitioner" is enormous. On a day to day basis, my company deals with software organizations throughout the United States that develop the full spectrum of software applications: from operating systems and word processors to control software (avionics, air-traffic, embedded medical device, etc.) to games. And if there is one reoccurring theme throughout these organizations, it is that formal methods will never catch on until formal methods lesson the time-to-market and cost the same as the techniques they replace. After all, there is an awful lot of good quality software in use today that got hacked together overnight without formal methods. And as long as industry can operate in this mode, they will.

The argument that bugs caught earlier cost less over the life-time of a product does not outweigh a different concern: "correct software that costs 100 times as much to develop and that misses the market window." And rightly so. The software industry moves fast. Time to market and costs are *all* that matters. The first "usable" product to market captures 100% of its marketshare. Recognize that even if a competitor of Microsoft were to come along with a bug-free document-editing package, they have no chance of stealing the market held by Microsoft Word.

If formal methodists (researchers) truly want the "fruits of their labor" used in the mainstream, they need to: (1) understand that generally speaking, marketing concerns outweigh quality concerns to development organizations, and (2) find ways to lessen the costs of employing them. If (2) cannot be successfully addressed, formal methodists will find their techniques only used in regulated industries.

4 Protecting Against What You Fear

Ensuring that software only does what you want it to do is a multi-phased process. A good beginning is to *define* precisely what events you do not want the software to cause. Then it is necessary to *design* and *develop* the software such that it cannot cause those events, and finally to *demonstrate* that the code cannot exhibit those undesirable behaviors.

There are a variety of formal analyses such as fault tree analysis and Petri net analysis that can demonstrate that only "desirable" functionality is provided by a software design (and that undesirable functionality is not). Generally speaking, protecting against those events that are feared is a straightforward process: (1) system-level analysis defines those events that must be protected against, and (2) a design technique (like static fault-tree analysis combined with proofs-by-contradiction) mitigates the possibility of those events occurring. If the system includes real-time constraints, formal Petri Net analysis can prove that timing constraints are not violated.

By defining undesirable events early on in the life-cycle, half of the battle of protecting against them is waged. The next task is to actually implement the protection. But even if formal analyses assert that the protection provided in the design and software is adequate, prudence suggests that a real demonstration is needed that executes the software to test the protection.

Experimentation using *testing* and *fault injection* are ideal here. These informal analyses provide confidence in the software as well as in the formal analyses. The point here is that the aforementioned informal analyses can measure software quality. Formal methods do not. But a measurement of quality for software that was formally developed is also a measure of process quality. For example, if formal methods say that output event E cannot occur, and the results from testing show that the likelihood of E occurring is less than P, and P is tiny, then testing has effectively substantiated the proposition. In this manner, formal and informal methods can complement each other nicely.

5 Protecting Against What You Have Forgotten About

It is far easier to protect against what you fear than what you have failed to fear. Formal methods, if properly applied, can do much for the former and little for the latter.

Successfully applying formal methods can, at best, result in software that does precisely what was specified. If the specifications are wrong and certain undesirable output behaviors were accidentally not defined in the specification (and thus not protected against in the code), formal methods will be helpless.[1]

Here, one variation on traditional software testing, *software fault injection*, can help warn of undesirable capabilities of the software that the specification/requirements have failed to warn against. Although this process will involve manual effort (thus making the process less than foolproof), it can force the software to reveal outputs that were not known to be possible.

Fault injection is a technique that can flush out certain types of undesirable output events after the software is forced to experience certain types of anomalous behaviors [3, 2]. To perform fault injection analysis, the user of the technique is required to define the classes of undesirable (or hazardous) output events. Fault injection then observes whether outputs that satisfy these definitions are produced.

Fault injection forces internal states of the software to become corrupted in a variety of ways. These internal corruptions may or may not force the output of the software to be modified. Sometimes the modified output will be instances of what was supposedly deemed as impossible to produce according to the formal analysis. When this occurs, fault injection warns that the software can still produce undesirable outputs that were thought to be impossible. Here, fault injection is testing the development processes just as much as it is testing the code.

Just because an output is produced that was not deemed as undesirable according to the specification does not mean that the output is desirable. After all, the specification defining undesirable output behaviors may be incomplete. Therefore it is valuable to have a person manually sift through those outputs that appear to be acceptable to see if there are any output events that are actually undesirable. If there are, then the specification that defined "good" versus "bad" output behavior failed to define certain types of outputs as bad. Thus the specification needs to be modified. In this manner, fault injection, in conjunction with manual effort, can analyze the specification for completeness of those definitions that define undesirable behavior.

For an example that brings this back to component-based software engineering, suppose

[1] There are some exceptions here however.

that you have a COTS component that has been “wrapped” using formal methods. The formal methods have asserted that certain events from this component are no longer possible. To test this claim, fault injection can force corrupt information into the component, and see what the component does. (Regular testing can be used also.) If we are unable to force the component to exhibit undesirable behaviors, then we can have confidence in the formally-derived wrappers. And if undesirable behaviors are exhibited that were not protected against in the wrapper, then we can use fault injection to define additional wrapping requirements to the wrapper's specification.

6 How Marrying Formal Methods and Testing Could Aid Component-based Systems

Using formal methods during the design process for building systems from components is prudent. Formal methods can prove properties about component *wrappers*. Formal methods can also foretell how well the components will interact with other components. Both of these applications of formal methods to component-based systems is beneficial and adds an engineering-like rigor to the process of software development.

But proofs are not foolproof. Informal methods such as testing and fault injection can decrease the likelihood that errors “slip through the formal cracks” by validating the claims made as a result of the formal analyses. For example, suppose that two components, A and B, are composed in the manner where A calls B. If we have proven certain properties about these two components in isolation, that information can be used to prove properties about A(B). Ideally, we would test the components in isolation as well as test the composite. Further, if a component is “wrapped” to nullify the possibility of certain outputs being produced, we can test to see if those outputs can be forced to occur via testing and/or fault injection testing.

Formal proofs are commonly used for showing that the composition between two “good” components will produce a “good” composite. (Here, “good” can refer to qualities as secure, correct, etc.) Ultimately, these proofs can be wrong, or the belief that the components are good could be in error. By testing the subsystem, confidence that the results from the proofs are correct is gained. This uses “testing” to test the accuracy of formal analyses while also testing the quality of the software.

Marrying these approaches would serve two specific purposes: one would be to determine when these approaches overlap in terms of the value-added to a program's quality, and the second would be to provide better data substantiating the claim that formal methods are cost-effective. Clearly, we cannot apply formal methods to all parts of today's enormous systems, and we need to apply formal methods to those regions where testing is least effective at detecting errors. There are certain classes of errors that each technique is likely to overlook, and it would be ideal to hone these techniques toward what they are good at and not chastise them for their deficiencies.

Component-based software development is slowly becoming a reality, and there are those that feel that formal methods, by themselves, will solve the key hurdles that are associated with building “systems from systems.” We take a more cautious stance and believe that the claims from using formal methods will always need to be tested out for each individual application.

The key impediment to component-based development becoming state-of-the-practice is the issue of component quality. If that problem were solved, and if a component consumer knew how an acquired component would behave in their environment after adoption, then the opportunity costs associated with selecting one component over the other could be better decided. Our research team at Reliable Software Technologies is currently working on one solution to this problem as discussed in the Background Section.

In summary, testing and formal methods are complementary approaches that can be leveraged to terrific advantage in component-based systems. We believe that a marriage between the two different approaches is long overdue.

Acknowledgements

This work has been partially supported by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160.

References

[1] B. Beizer. Cleanroom process model: A critical examination. *IEEE Software*, 14(2):14–16, March 1997.

[2] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, 14(4):73–83, July 1997.

[3] J. Voas, G. McGraw, L. Kassab and L. Voas. A Crystal Ball for Software Liability. *IEEE Computer*, 30(6):29–36, June 1997.

7 Biography

Dr. Jeffrey Voas is a Co-founder and Chief Scientist of Reliable Software Technologies and is currently the principal investigator on research initiatives for DARPA and the National Institute of Standards and Technology. Voas has also recently served as a Principle Investigator on efforts for NASA, National Science Foundation, and the USAF. He has published over 85 refereed journal and conference papers. Voas has coauthored a text, "Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995)." Voas has co-authored a second text, "Software fault-injection: inoculating programs against errors," to be published by Wiley (Fall, 1997). Voas will be the special editor of an *IEEE Computer* theme issue on "Commercial off-the-shelf software" in 1998. Voas was the General Chair for COMPASS'97, and serves on the Editorial Board for the Software Quality Professional Journal. Voas is currently writing a chapter on *software liability* for the "Advances in Computer" book series. Voas's current research interests include: information security metrics, software liability, certification, software dependability metrics, COTS-based systems, and information warfare tactics.