Test Generation and Recognition with Formal Methods

Paul E. Ammann
George Mason University
Information & Software Eng. Dept.
Fairfax, Virginia 22033 USA
pammann@gmu.edu

Paul E. Black
National Institute of Standards and Technology
100 Bureau Dr., Stop 8970
Gaithersburg, Maryland 20899 USA
paul.black@nist.gov

Abstract

The authors are part of a larger group at the National Institute of Standards and Technology (NIST), George Mason University (GMU), and the University of Maryland, Baltimore County (UMBC). Projects directed by group members use formal methods, particularly model checking, to investigate the generation and recognition of test sets for software systems. Our positions, in order of increasing potential controversy, are 1) the use of specifications is an important complement to code-based methods, 2) test set recognition is as important as test set generation, and 3) in spite of some known limitations, our generic framework for testing, with a test criterion as a parameter and a model checker for an engine, is a general approach that can handle many interesting specification-based test criteria.

1. Our Relevant Work

For the context of our position, we summarize our recent contributions to specification-based testing using a model checker. Model checkers, which evaluate finite state machines with respect to temporal logic constraints, are chosen in favor of theorem proving approaches because 1) significantly less expertise is required of the end user, thereby enhancing automation, 2) model checkers are enjoying an explosive growth in applicability, and 3) the counterexamples from a model checker may be directly interpreted as test cases.

In our original paper on the topic [4], we defined mutation testing for model checking specifications, specifically SMV descriptions. We defined one class of mutation operators that changed the state machine description; these operators result in failing tests, that is, tests that a correct implementation must reject. We defined another class of mutation operators that changed the temporal logic constraints on the state machine; these operators result in passing tests, that is, tests that a correct implementation must accept. The model checker identifies equivalent mutants: these are temporal logic constraints that are consistent. We generated tests for a small example, ran them against a target implementation, and measured code branch coverage.

Generating tests to “kill” all mutants is the first test criterion we investigated. Some other specification-based criteria are stuck-at faults [1], CCC partitions [6], MC/DC [7], automata theoretic [8], branch coverage [10], disconnection or redirection faults [11], and transition pair coverage [13]. Test generation then is the problem of finding tests which fulfill the goals embodied in the criterion. Test set recognition is the conjugate of test generation. Whereas test generation asks, “What tests will satisfy the test criteria?”, test recognition asks, “How much of the test criteria do these tests satisfy?”

In follow-on work [3], we addressed test set recognition for a refinement of the mutation analysis scheme. In particular, we defined a metric in terms of number of mutants killed by a given test set compared to the total number of killable mutants. We showed how to turn tests from a candidate test set into ”forced” state machines and then use the model checker to compute the metric. We analyzed various factors that could introduce distortions, such as semantically equivalent mutants and mutants that are killed by every test case, were analyzed.

We analyzed different mutation operators both theoretically and empirically [5]. For theoretical analysis, we applied predicate differencing and a hierarchy of fault classes [12]. To experimentally confirm the conclusions, we generated tests using many mutation operators for three different small examples and compared relative coverage of the different operators. Although mutation operators do not correspond exactly to fault classes, we found good correlation between them. We defined a composite mutation operator which gave the maximum coverage, and found a single mutation operator which gave nearly-maximum coverage using far fewer mutants.

Although the above methods work well for state machine specifications, most specifications are written at higher levels in Z, UML, OCL, SCR, etc. So to be practical, there must be (semi-)automatic ways of extracting simpler pieces which can be analyzed. In [2] we defined a new algorithm to abstract a simple state machine, focusing on some states of interest to an analyst, from an unbounded description. We proved that the algorithm is sound for test generation. That is, any test produced corresponds to a passing tests in the original unbounded description.

We also applied the work to the problem of network security [14], particularly cases where configuration changes on one machine can lead to vulnerabilities on other machines in a network. Network configurations were encoded as a state machine, along with the transformations produced by known attacks. Security policies are stated in the temporal logic in forms such as "Under a set of assumptions, someone outside the firewall cannot obtain root access on machine X." If the configuration in fact allows such access given the set of known attacks, a counterexample is produced illustrating the attack.

In work underway, we encoded different test criteria as temporal logic constraints. Using a model checker, we analyzed branch coverage [10], uncorrelated full-predicate coverage (similar to Multiple Condition/Decision Coverage or MC/DC [7]), and transition-pair coverage [13], in addition to mutation coverage. We found that different metrics are easily encoded into temporal logic, with some limitations, and that interesting theoretical comparisons between metrics are facilitated by formalizing them.

To scale these methods up to problems of useful size and general nature, we successfully applied them to several different examples. We began with small, well-known examples such as Cruise Control and Safety Injection. We also modeled the operand stack of a Java virtual machine and several functional source code benchmarks for unit testing, then generated good test sets. Currently we are applying the method to a part of a flight guidance system from an aerospace firm and to a secure operating system add-on for a Unix derivative.

Other Work

The earliest work we know of on generating tests using model checkers is when Callahan, Schneider, and Easterbrook [6] mentioned that counterexamples generated from SPIN, MurΦ, or SMV model checkers can be used as test cases.

Engels, Feijs, and Mauw [9] named some general concepts, such as "test purposes" (some goals to achieve with testing) and "never-claim" (submit the negation of what you want so the model checker finds a positive instance). They discussed positive and negative testing. Positive testing checks that the system does what it should, which is appropriate for general system checks. Negative testing looks for a particular action the system should *not* do. The disadvantage is that one must specify the errors to look for, but it may be useful in searching for particular errors.

Most recently Gargantini and Heitmeyer [10] developed a requirements branch or case coverage test purpose using the SPIN or SMV model checkers. Also conditions in requirements may be elaborated in the test purposes to exercise boundary conditions, for instance, $x \geq y$ may be split into $x > y$ and $x = y$.

2. Research Questions

In January 2000 the group held an informal workshop at NIST. Some 20 scientists, professors, and students spent half a day sharing their views on the work, listing programs we need, and defining research topics and questions, such as:

1. What are the effects of semantically identical, but syntactically different specification styles on test set quality?

2. How do we make tests observable?

3. How can we partition a huge model between light- and heavy-weight formal methods, then combine their results to get tests?

4. What are the advantages and disadvantages for test generation or expressibility with SMV and SPIN (CTL vs. LTL)?

5. How can (should) we trade off number of tests and coverage?

6. What are good (semi-)automatic abstractions from large, even infinite descriptions for test generation?

7. Can we use state machine mutations (failing tests) to check systems for safety?

8. What are a good set of mutation operators, e.g., for larger models.

9. How do duplicate mutants affect coverage metrics? Do some sets of mutation operators produce many or few duplicates?

3. Position Statement

- The use of specifications is an important complement to code-based methods.

This is an old position, but we argue that recent trends in software development and testing make it more compelling. The traditional argument, which is still valid, is that without a specification, we do not know to test for features which are entirely missing from the source code. More importantly, in acceptance tests of binary programs or conformance testing without a reference implementation, there is no source code available at all. During rapid development it may be helpful to write tests in parallel with or even preceding coding; such a model is directly supported by the use of "use-cases" in requirements analysis. Use-cases are essentially system tests, and analyzing use-cases with respect to specification-based test metrics is an important research area. Further, there is a body of research that aims to introduce formal methods into industrial development by amortizing the cost of developing formal specifications over other, traditionally expensive, lifecycle phases, particularly testing. Model-checkers are a relatively new, but powerful tool in achieving this objective.

- Test set recognition is as important as test set generation.

There are basically two thrusts to this argument, one theoretical and the other practical. The theoretical argument is that scientific comparisons between test methods benefit greatly if a test set produced by method A can be evaluated directly and without bias with respect to method B. Test methods that focus purely on test generation do not satisfy this objective. The practical argument is that industry has an enormous investment in existing test sets, primarily in regression test sets, but also in new development artifacts such as use-cases from requirement analysis. To retain the value of this investment, it is much more helpful to critique the existing artifacts with statements of the form, "Tests of type X and Y are missing," rather than merely providing a new test set that bears no relation to the existing ones.

- In spite of known limitations, our generic framework for testing, with a test criterion as a parameter and a model checker for an engine, is a general approach that can handle many interesting specification-based test criteria.

We define a model whereby a test criterion is paired with a specification of a specific application, and, with as much automation as possible, test requirements specific to the application are generated and satisfied with specific tests, either new or old. The key is the degree of automation. We believe that using a temporal logic to express the test requirements and a model checker to create and/or match test cases to test requirements is a general purpose approach suitable for many specification-based test methods. As described above, this approach has been successful for a variety of interesting test criteria. One interesting aspect of this line of research has been in discovering where the method falls short. The significant result so far is that any test requirement that places constraints on pairs of tests (as opposed to individual tests) is not well handled by a model checker, since counterexamples are typically generated one at a time. An example is the

MC/DC metric, popular in avionic applications. In MC/DC, pairs of tests are required to differ in the value of exactly one condition. The research question is how to work around this expressibility constraint.

References

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital System Testing and Testable Design*. IEEE Computer Society Press, New York, N.Y., 1990.

[2] Paul Ammann and Paul E. Black. Abstracting formal specifications to generate software tests via model checking. In *Proceedings of the 18th Digital Avionics Systems Conference (DASC99)*, volume 2, page 10.A.6. IEEE, October 1999. Also NIST IR 6405.

[3] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.

[4] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, December 1998.

[5] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In 15^{th} *IEEE International Conference on Automated Software Engineering (ASE2000)*, October 2000. Submitted.

[6] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.

[7] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[8] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[9] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer-Verlag, April 1997.

[10] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, September 1999. To Appear.

[11] Jens Chr. Godskesen. Fault models for embedded systems. In *Proceedings of CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.

[12] D. Richard Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[13] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE Fifth International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.

[14] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings 2000 IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA, May 2000. To Appear.