Using Mutation for the Assessment and Optimization of Tests and Properties*

Technical Report 2006-518

Jeremy S. Bradbury

School of Computing, Queen's University
Kingston, Ontario, Canada
$bradbury@cs.queensu.ca$

August 2006

Abstract

We are interested in exploring the complementary relationship and tradeoffs between testing and property-based analysis with respect to bug detection. In this paper we present an empirical approach to the assessment of testing and property-based analysis tools using metrics to measure the quantity and efficiency of each technique at finding bugs. We have implemented our approach in an assessment component that has been constructed to allow for symmetrical comparison and evaluation of tests versus properties. In addition to assessing test cases and properties we are also interested in using each to optimize the other as well as to develop hybrid quality assurance approaches. We hypothesize that the synergies of using testing and property-based analysis in combination will allow for optimizations in test suites and property sets that are not possible by using both approaches in isolation.

1 Introduction

The goal of the proposed research work is to increase the quality assurance of software systems by exploiting the synergies that exist between testing and property-based analysis. Our interest in exploring the complementary relationship between testing and property-based analysis is motivated on the one hand by advances in the theory and practise of property-based analysis, especially formal analysis, and on the other hand, by a need for improved quality assurance techniques for industrial code – especially concurrent code. We believe that the property-based formal analysis tools that are now available offer the potential to substantially aid in the debugging of industrial concurrent code. Intuitively, the detection of a property or assertion violation, such as a violation of a method pre-condition, a loop invariant, a class representation invariant, an interface usage rule, or a temporal property should be more insightful than the failure of a possibly global test case.

A shift in the focus of formal methods from proofs of correctness to debugging and testing has been advocated by a number of researchers including Rushby [20]. In recent years, tool development in the formal analysis community has matured and the current generation of tools are automatic, scalable, and only leave a small semantic gap between the source artifacts used by developers and the model artifacts required for analysis. The ability to directly analyze source code and the increase in size of systems that can be analyzed has helped formal analysis become a viable option for software debugging.

*This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)

While the majority of software systems currently developed in industry are single-threaded sequential programs, there is mounting evidence that "applications will increasingly need to be concurrent if they want to fully exploit CPU throughput gains that have now started becoming available and will continue to materialize over the next several years" [21]. The shift from sequential to concurrent systems provides an opportunity for the application of formal analysis techniques which can often succeed at debugging concurrent systems while testing in this setting is often insufficient or impractical.

2 Hypothesis

> *Using a mutation-based approach to testing and property-based analysis will allow for the assessment and comparison of test suites and property sets. Furthermore, the synergies of using testing and formal analysis in combination will allow for optimizations in test suites and property sets that are not possible by using both approaches in isolation. These optimizations can be integrated efficiently into a modern software quality assurance process and can help to increase the effectiveness and efficiency of software quality assurance processes.*

The term *assessment* in our hypothesis refers to the statistical evaluation of a test suite or property set using mutation testing metrics. Mutation testing uses mutation operators to generate faulty versions of the original program called mutants. If we assume the original program as being correct then a mutant version that is non-equivalent can be thought of as having a bug. The percentage of non-equivalent mutants detected (killed) by a test suite or property set is the mutant score. We have chosen to use a mutation metric because a recent study found that for the programs being studied, mutant faults were a good measure of real faults [2].

The term *optimization* refers to two activities: generation and reduction. On the one hand, test cases and properties are generated to allow a given test suite or property set to achieve a better evaluation using the mutation testing metric. On the other hand, test suites and property sets are reduced when the removal of test cases and property sets will not reduce the evaluation result using the metric.

3 Research Approach

Our research approach consists of two phases. The first phase is the assessment phase where we empirically evaluate and explore the synergies between testing and property-based analysis. The second phase is the optimization phase where we take the results from out empirical assessment and try to use it to enhance the quality assurance of a given program by improving the test suite and property set as well as develop hybrid techniques. Figure 1 shows our proposed framework to support our research approach. The current state of the framework is that the assessment component has been completed but the optimization components have not. We will now outline the assessment phase and the optimization phase in more detail. Due to space we will not discuss our experimental setup which was presented in a previous paper [4].

3.1 Assessment Phase

To support the experimental procedure we have developed the assessment component of our framework such that it features a high-degree of automation and customizability and thus allows for a large number of experiments to be carried out as efficiently as possible. Our assessment component essentially consists of a Java application that acts as a wrapper to all of the other tools and scripts used. The framework is generic enough to allow for the comparison of multiple testing and property-based analysis approaches.

For simplicity we will only explain the assessment framework in the context of comparing one testing technique (e.g., concurrent testing using a randomized scheduler with ConTest [8]) with one property-based analysis technique (e.g., static analysis approach using Path Inspector [1]) or formal analysis approach using

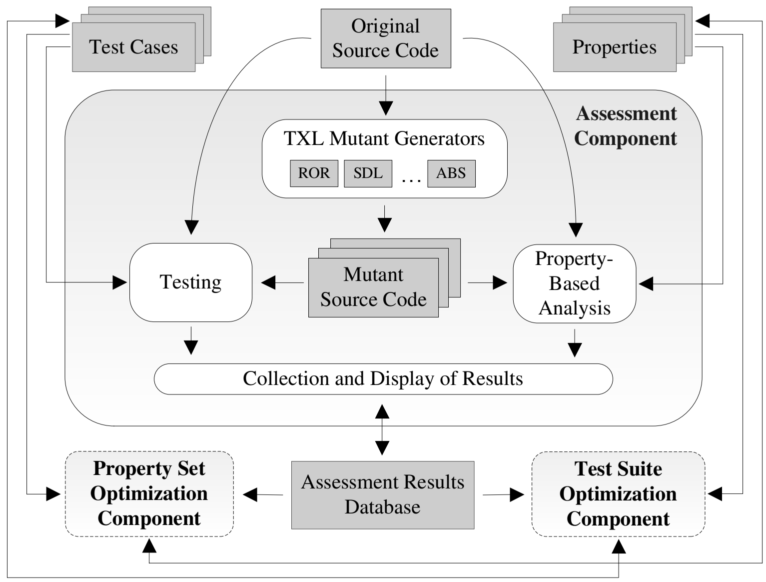

Figure 1: Proposed Assessment and Optimization Framework

Bandera [6]/Bogor [19]). When comparing testing with a property-based analysis the component requires as input a program (e.g., written in C, C++, Java) and an accompanying test suite and property set. Once the inputs have been selected, the assessment component implements four main steps in the experimental procedure:

1. *Mutant generation:* A built-in set of mutation operators can be selected individually to allow for the generation of mutant programs to be customized.
2. *Property-based Analysis:* Our application calls an automatically generated script which allows all of the property-based analyses to be performed automatically. For our set of properties we first evaluate each property using the original program to determine the expected outputs. Next we evaluate our property set for all of the mutant versions of the original program. During property-based analysis all of the verification results, generated counter-examples, and analysis execution times of each property with each program are recorded.
3. *Testing:* Our application calls an automatically generated script which compiles the source code and executes the testing, recording the output result and execution time for each test case with each program.
4. *Collection and display of results:* We compare the execution and analysis results of the original program with the results of executing and analyzing the mutant programs to see if each test case and property was able to distinguish the mutant programs from the original (see Table 1).

Currently, we have implemented all four steps of the experimental procedure in the assessment component and sample screenshots of the component are given in Figure 2. We have made the component customizable and flexible to support a wide range of experiments using different programs, languages, tools, and properties. For example, we plan to compare different property-based analysis techniques and compare different types of properties (e.g. assertions vs. LTL properties). We will discuss the specific experiments we are currently planning in Section 4.

	Assessment Results Reported by the Framework
Tests	• Mutant score for each test case/test suite • Execution cost for each test case/test suite • Number of test cases that kill each mutant
Properties	• Mutant score for each property/property set • Execution cost for each property/property set • Number of properties that kill each mutant • Mutant score for each property pattern type • Types of mutants killed by each property pattern type
Integrating tests & properties	• Hybrid set of tests and properties that achieve the highest mutant score • Hybrid set of tests and properties that achieve a certain mutant score (e.g. 90%) and ○ has the lowest execution cost ○ has the smallest set of tests and properties

Table 1: Types of results collected and reported

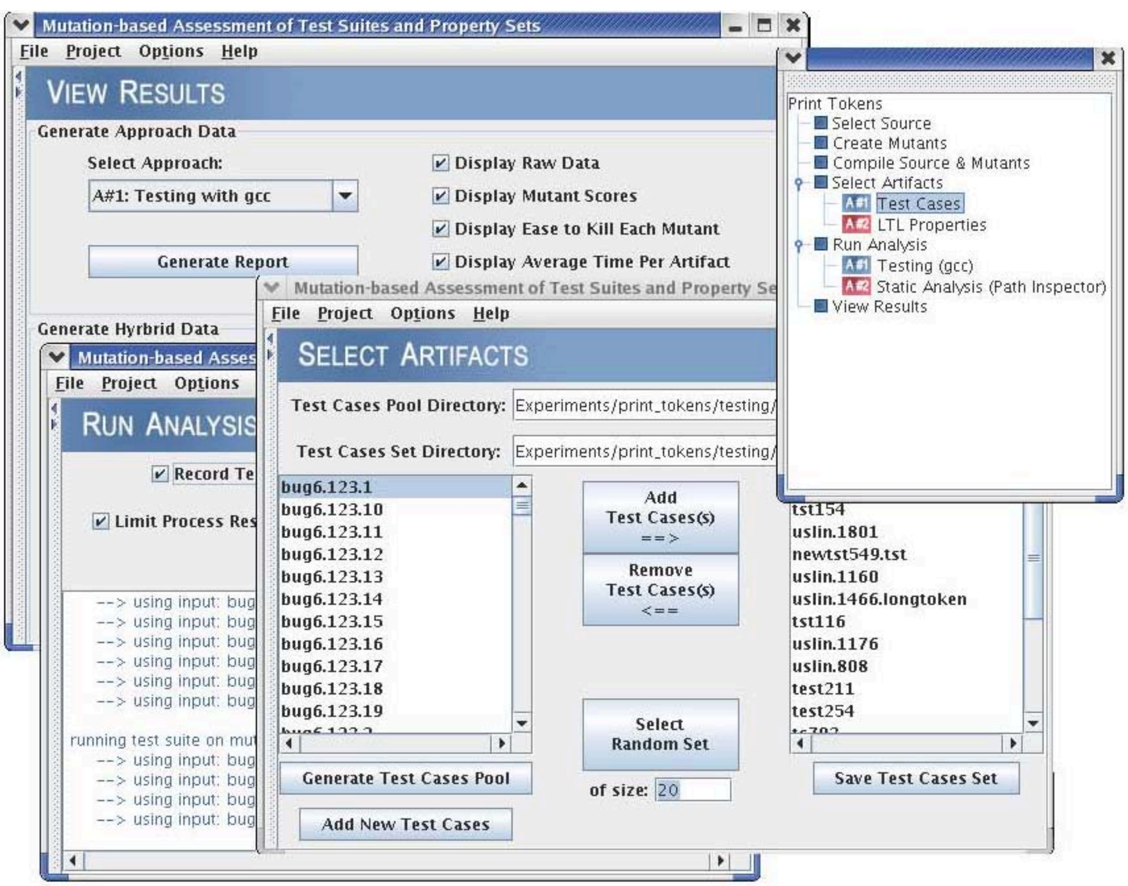

Figure 2: Assessment Component Screenshots

3.2 Optimization Phase

The optimization phase of our research is currently future work and will not begin until we have collected all of the empirical results from the assessment phase. The optimization phase will consist of 4 steps and will be supported by the development of two components; a test suite optimization component and a property set optimization component:

1. *Test case generation:* If there exists a mutant that is killed by a property and no test case exists to kill the mutant then we plan to use a counter-example generated by the formal analysis to develop a test case that will kill the mutants. Similar approaches to the test case generation are described in [3, 12].
2. *Property generation:* If there exists a mutant that is killed by possibly several test cases and no property exists to kill the mutant the use of dynamic slicing or run-time monitoring of the test cases for the generation of a property that will kill the mutant could be investigated. Related work includes Daikon [10] and Terracotta [24], that both use tests as input.
3. *Test suite reduction:* If one property kills several mutants and several test cases kill the same mutants then we might be able to generate a test case that kills all of the mutants that can replace multiple test cases.
4. *Property set reduction:* On one hand, we will reduce the set of properties by comparing the mutants killed by each property. If the mutants killed by a given property are all killed by other properties it could be removed. On the other hand, if one test case kills several mutants and several properties in our set kill the same mutants then we might be able to generate a single property that kills all of the mutants that can replace multiple properties.

Our approach to optimization should allow for both the test suite and property set to be enhanced for use in isolation and in hybrid approaches. Our optimization should provide a more comprehensive test suite and property set that will allow the testing and property-based analysis to each be more comprehensive and succinct. Furthermore, if we re-run our assessment on the improved test suite and property set we might also be able to improve the hybrid approaches identified by our assessment component.

4 Evaluation Plan

To answer the first part of our hypothesis and to evaluate our assessment component we plan to run at least the following four experiments:

1. Random Testing *vs.* Random Property-based Static Analysis using Path Inspector
2. Coverage-based Testing *vs.* Hand Crafted Property-based Static Analysis using Path Inspector
3. Assertion-based Testing *vs.* Model Checking Assertions
4. Coverage-based Testing *vs.* Model Checking Assertions *vs.* Model Checking Temporal Logic Properties

For Experiments 1 and 2 we will use a set of small C programs created by Siemens [16] that include a pattern replace program, priority schedulers, lexical analyzers and others. For Experiments 3 and 4 we will be using concurrent Java programs (e.g., the file system Daisy and a banking example). Based on our preliminary results (discussed in [4]) we believe that further experimentation will show that in certain cases property-based analysis can find bugs that testing can not find and vice versa.

To answer the second half of our hypothesis and evaluate our optimization research approach we plan to investigate optimization techniques using each of our example programs from the assessment phase.

5 Related Work

Test suite assessment. The assessment of test suites is a well developed area. Relevant related work includes code coverage techniques (e.g., branch coverage) as well as the random schedulers often used to assess test suites for concurrent code [8].

Property set assessment. Unlike test suite assessment, property assessment with respect to source code does not appear to be well researched. Instead, properties are often assessed with respect to an abstract model of the code (e.g., finite state machines(FSMs), first-order logic). The use of mutation metrics in formal analysis primarily occurs at the model level, for example, to assess state-based coverage of FSMs [15]. The previous uses of mutation in formal analysis therefore differ from the research proposed here in the level at which the coverage techniques are applied – we propose a source code metric not a modelling language or FSM metric. Approaches that use mutation of abstract models instead of source code have benefits as a coverage metric but do not provide an assessment metric that can be easily used to compare property-based formal analysis to testing.

Test suite optimization Test case generation falls into two distinct categories: code-based generation and specification- or model-based generation. Our work is most related to specification-based generation approaches which have used FSMs and model checkers [11, 12], the Z language and type checker [5] and other specification representations as the basis for generation. An interesting variation on specification-based generation is operational abstraction using Daikon [14]. Operational abstraction uses the invariants generated from source code to generate test cases. Test suite reduction or minimization can also be divided into code-based and specification-based approaches. In [22], Extended FSM (EFSM) dependence analysis is used to identify and remove redundant tests with respect to a particular requirement.

Property set optimization In the area of property set optimization the most relevant related work is a property generation approach that uses Daikon [10] and ESC/Java [7]. The approach has two steps: invariant detection and invariant verification [17]. First, invariants (possible properties) are generated dynamically using Daikon. Second, the ESC/Java static checker verifies the correctness of the generated invariants statically. Invariant verification is necessary because the invariants generated dynamically may be unsound. Other invariant generation approaches include dynamic analysis approaches like DIDUCE [13] and Carrot [18], static analysis approaches (e.g., [9]) and the use of the model checker SPIN [23].

6 Expected Contributions

We propose to conduct an empirical study to explore the relationship and synergies between testing and property-based analysis and the usefulness of property-based static and formal analysis in detecting bugs in industrial source code. To the best of our knowledge our proposed study is a novel approach since no other work has used mutation metrics at the source code level as a method of comparing property-based analysis techniques with testing. Some of the contributions of our study include an experimental assessment component and empirical data (expected). In addition to using our assessment component to conduct experiments, other future work includes extending it to include the ability to optimize test suites and property sets.

7 Acknowledgments

I would like to thank my co-supervisors James R. Cordy and Juergen Dingel for their contributions to this work.

References

[1] P. Anderson. CodeSurfer/Path Inspector. In *Proc. of the IEEE Int. Conf. on Software Maintenance (ICSM'04)*, page 508, Sept. 2004.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of ICSE 2005*, pages 402–411, 2005.

[3] D. Beyer, A. J. Chlipala, T. A. Henzinger, et al. Generating tests from counterexamples. In *Proc. of ICSE 2004*, pages 326–335, May 2004.

[4] J. S. Bradbury, J. R. Cordy, and J. Dingel. An empirical framework for comparing effectiveness of testing and property-based formal analysis. In *Proc. of Int. Work. on Program Analysis for Software Tools and Engineering (PASTE 2005)*, Sept. 2005.

[5] S. Burton, J. Clark, and J. McDermid. Testing, proof and automation. an integrated approach. In *Proc. of the Int. Work. of Automated Program Analysis, Testing and Verification*, Jun. 2000.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, et al. Bandera: extracting finite-state models from java source code. In *Proc. of ICSE'00*, pages 439–448. ACM Press, 2000.

[7] D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, Dec. 1998.

[8] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Soft. Eng.*, 27(2):1–25, Feb. 2001.

[11] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. of ESEC/FSE-7*, pages 146–162, 1999.

[12] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *Proc. of the Int. Conf. on Soft. Eng. and Formal Methods (SEFM'04)*, pages 261–270, Sept. 2004.

[13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of ICSE 2002*, pages 291–301, 2002.

[14] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proc. of ICSE 2003*, pages 60–71, May 2003.

[15] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. of the ACM/IEEE Conf. on Design Automation*, pages 300–305, 1999.

[16] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of ICSE'94*, pages 191–200, May 1994.

[17] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proc. of ISSTA 2002*, pages 232–242, Jul. 2002.

[18] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proc. of the Int. Work. on Automated and Algorithmic Debugging (AADEBUG'2003)*, pages 273–276, Sept. 2003.

[19] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of ESEC/FSE 2003*, pages 267–276, 2003.

[20] J. Rushby. Disappearing formal methods. In *Proc. of the High-Assurance Systems Eng. Symp. (HASE'00)*, pages 95–96, Nov. 2000.

[21] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Mar. 2005.

[22] B. Vaysburg, L. H. Tahat, and B. Korel. Dependence analysis in reduction of requirement based test suites. In *Proc. of ISSTA 2002*, pages 107–111, 2002.

[23] M. Vaziri and G. Holzmann. Automatic generation of invariants in SPIN. In *Proc. of the Int. SPIN Work. (SPIN '98)*, Nov. 1998.

[24] J. Yang and D. Evans. Dynamically inferring temporal properties. In *Proc. of the Int. Work. on Program Analysis for Software Tools and Engineering (PASTE 2004)*, Jun. 2004.