Proc. of the International ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005), Lisbon, Portugal, September 2005.

An Empirical Framework for Comparing Effectiveness of Testing and Property-Based Formal Analysis*

Jeremy S. Bradbury, James R. Cordy, Juergen Dingel,
School of Computing, Queen's University, Kingston, Ontario, Canada
{bradbury, cordy, dingel}@cs.queensu.ca

ABSTRACT

Today, many formal analysis tools are not only used to provide certainty but are also used to debug software systems – a role that has traditional been reserved for testing tools. We are interested in exploring the complementary relationship as well as tradeoffs between testing and formal analysis with respect to debugging and more specifically bug detection. In this paper we present an approach to the assessment of testing and formal analysis tools using metrics to measure the quantity and efficiency of each technique at finding bugs. We also present an assessment framework that has been constructed to allow for symmetrical comparison and evaluation of tests versus properties. We are currently beginning to conduct experiments and this paper presents a discussion of possible outcomes of our proposed empirical study.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics

General Terms

Verification, Experimentation.

Keywords

bug detection, empirical software engineering, model checking, mutation testing, static analysis.

1. INTRODUCTION

Testing and formal analysis are two complementary quality assurance techniques. Testing can be lightweight but incomplete while formal analysis tends to be more heavyweight but complete. Our interest in exploring the complementary relationship between testing and formal analysis is motivated on the one hand by advances in the theory and practise of formal analysis and on the other hand, by a need for improved quality assurance techniques for industrial code – especially concurrent code.

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 [15]. An example of this shift is demonstrated by the approximately a quarter of a million assertions in the Microsoft Office code [8]. The primary application of these assertions is "...not to the proof of program correctness, but to the diagnosis of their errors" [8]. 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. For example, in the SLAM project automated refinement allows for direct static analysis of device driver source code [3]. 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. For example, in many formal analysis tools a counter-example is produced if the verification of a property fails. When a counter-example is produced it can be used to locate the error in source 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 simply knowing that there was a failure of a possibly global test case.

The majority of software systems currently developed in industry are single-threaded sequential programs [16]. However, 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" [16]. 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.

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. In this paper we present an assessment framework to empirically

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

PASTE 2005

evaluate and identify the tradeoffs between formal analysis and testing with respect to bug detection. Our goal is to use the assessment framework to answer several open questions that have not been addressed in previous research:

- How good is property-based formal analysis at finding bugs in source code?
- How efficient is a formal analysis technique at finding bugs in comparison to testing or in comparison to another formal analysis technique?
- Can a hybrid approach that combines formal analysis and testing ever find more bugs or be more efficient than either approach used in isolation?

Furthermore, we hope to use our framework and future empirical results to identify ways to integrate formal analysis and testing to better aid in debugging software.

2. PROPOSED EXPERIMENTAL STUDY

The goal of our proposed study is to statistically evaluate test suites and sets of formal properties using metrics to determine both the quantity of bugs found and the efficiency of each approach. We will outline our experimental setup and procedure before discussing possible research outcomes.

2.1 Experimental Setup

Selection of Metrics. We will use mutation testing metrics [2] to assess the quantity of bugs found. 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 mutant faults were a good measure of real faults [2].

To evaluate the efficiency of testing and formal analysis at finding bugs we use execution cost as a metric. We record the execution time to run each test case and verify each property and then average the time over the number of mutants killed by each to determine the cost to kill each mutant.

Selection of Testing and Formal Analysis Techniques. In our experiments we plan to evaluate existing test suites and properties. We want to evaluate properties using analysis tools that satisfy the following characteristics: ability to take source code of a modern programming language such as C/C++ or Java as input, ability to analyze a program with respect to a set of properties, and the ability to produce a counter-example if a property is not satisfied. We also want tools that are lightweight, scalable, and automatic because we want formal analysis to be a viable and practical complement to testing even in industrial settings.

Our set of criteria suggest the use of static analysis tools which are scalable but incomplete. After considering a number of tools we decided to assess one formal static analysis tool in our first experiments – Path Inspector [1], a plug-in for GrammaTech's[1] CodeSurfer that can analyze sequential C/C++ source code. In Path Inspector the user can specify sequencing properties based on temporal logic property patterns [5]. For example, one type of property is "*P occurs globally*" where *P* can be any program point (e.g. a function call). A property in Path Inspector is verified by analyzing the control flow graph of the system. If the property is not satisfied a counter-example is displayed. Since Path Inspector is incomplete spurious results are possible because some of the paths in the control flow graph maybe infeasible during execution.

Although our criteria suggest the use of static analysis tools, other techniques traditionally associated with formal analysis (e.g., model checking, theorem proving) could also be used provided they satisfy the criteria. In the future we plan to run experiments using the model checker Bogor [13]. Bogor includes a plug-in called Bandera which allows for the analysis of Java programs using Linear Temporal Logic (LTL) properties or assertions. The benefits of using Bogor are the analysis is complete and spurious results are not possible[2], and concurrent programs as well as sequential programs can be analyzed. In addition to Bogor, tools based on automatic abstraction refinement (e.g., SLAM, Magic, Blast) could be used.

Selection of Example Programs, Test Suites, and Property Sets. In selecting example programs to use we are limited in size by the scalability of the formal analysis. We have tried to find example programs that have a mature test suite and an existing property specification. Unfortunately, it is difficult to find suitable real-world source code, and even more difficult to find examples with a mature test suite and property specification. Therefore, for the purpose of our experiments we had to consider examples with no preexisting tests or properties.

To start, we will use a set of small programs created by Siemens [6, 10, 14] that include a pattern replace program, priority schedulers, lexical analyzers and others. The sample programs are written in C, which will allow us to evaluate properties in Path Inspector. Each program has existing test suites, some that provide branch coverage and others that are generated at random. These programs do not have existing temporal logic properties that can be evaluated in Path Inspector. As a first step we plan to generate properties randomly and compare them to randomly generated test suites. We also plan to hand-craft a set of properties for comparison with more sophisticated and mature test suites.

In addition to the Siemens programs, we plan to run our experiments on several other programs involving standard data structures and sorting routines.

2.2 Experimental Procedure

To support the experimental procedure we have developed an assessment framework with a high-degree of automation to allow for a large number of experiments to be carried out as efficiently as possible. Our assessment framework (see Fig. 1) 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 testing with formal analysis using Path Inspector or a model checker such as Bogor. However, for simplicity we will only explain the assessment framework in the context of comparing testing with formal analysis using Path Inspector. When comparing testing with Path Inspector analysis the framework requires as input a C/C++ program and an

[1]GrammaTech home page: `http://www.grammatech.com`

[2]Spurious results may occur in Bogor if abstraction is used in the Bandera translation from Java to BIR(the Bogor modeling language).

accompanying test suite and property set. Once the inputs have been selected, the framework implements the four main steps of the experimental procedure:

1. *Mutant generation:* A built in set of mutation operators can be selected individually to allow for the generation of mutant C/C++ programs to be customized. In the future we also plan to add new mutation operators depending on the domain. For example, in a concurrent programming domain we might adapt a general re-order statements mutation operator to only re-order statements from inside to outside a critical region and vice versa.

2. *Formal Analysis:* Our application calls an auto-generated script which allows all of the formal analysis 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 formal 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 auto-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 verification results of the original program with the results of executing and verifying the mutant programs to see if each test case and property was able to distinguish the mutant programs from the original (see Table 1).

The current status of the assessment framework is that we have implemented the mutation, testing, and formal analysis and are currently completing the display of results. We are also working to make the framework more customizable and flexible to support a wide range of experiments using different programs, languages, tools, and properties. For example, we would like to compare different formal analysis techniques (e.g. Path Inspector vs. model checking) and compare different types of properties (e.g. assertions vs. LTL properties).

2.3 Possible Outcomes

We are currently completing the results reporting and therefore we can only discuss possible outcomes that are based on preliminary results. These results involve one program using Path Inspector and one program using Bandera in combination with a standard model checker. Both programs were evaluated using a small set of properties and mutant operators. Based on our preliminary results we believe that further experimentation will show that in certain cases formal analysis can find bugs that testing can not find and vice versa. For example, in a concurrent setting testing may not be able to find certain bugs at all, while formal analysis can. Moreover, we suspect that formal analysis using Path Inspector is not as beneficial as testing when it comes to finding bugs in conditional expressions (e.g. incorrect relational operators) because of the imprecision inherent to static analysis.

	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

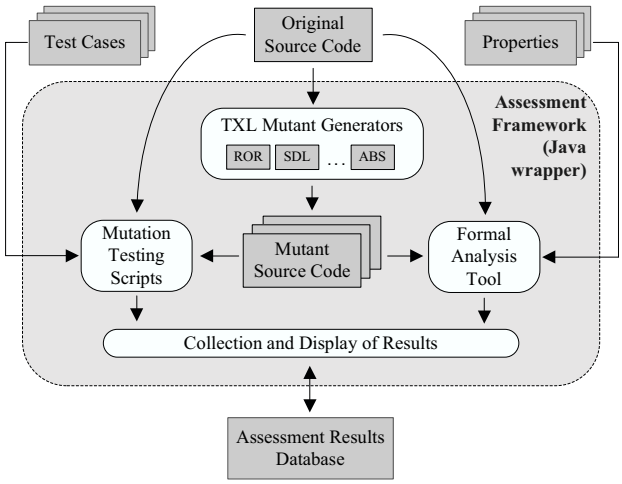

Figure 1: Assessment framework

In terms of efficiency, one possibility is that we will find that formal analysis can be as efficient or better than testing in certain cases. For example, if a given property kills a number of mutants that are normally killed by a set of test cases, it is possible that sometimes the execution time for verifying the property is less than the time required to run the test cases. If this is the outcome of our research then an optimal quality assurance approach with respect to execution cost might be hybrid and include both formal analysis and testing. If the outcome is that testing is always more efficient then formal analysis with a given tool we would still like to know by what factor is it more efficient because the analysis of the properties might still provide increased insight to developers that can be factored against its increased cost. Therefore, we would also like to know the kinds of properties that kill mutants and see if they have other benefits for developers in addition to finding bugs.

3. RELATED WORK

Mutation testing has been used in the testing community for over 25 years and has proven to be a useful comparison metric for assessing and improving multiple test suites. 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,

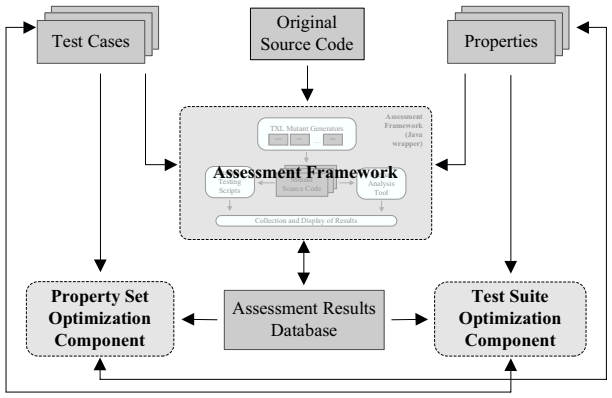

Figure 2: Future directions – extending the assessment framework to include optimization

mutation metrics have been used to assess state-based coverage of FSMs in model checking [9]. 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 formal analysis to testing.

4. CONCLUSIONS AND FUTURE WORK

Many people have already argued convincingly that there is a need for more empirical, quantitative research in software engineering in general [4] as well as better empirical practises [11, 12]. Moreover, many people have also already argued, perhaps less convincingly, that testing and formal analysis are complementary and offer powerful synergies that could be leveraged through a combined use. Our work is motivated by both of these observations. In particular, we propose to conduct an empirical study to explore the relationship and synergies between testing and formal analysis and the usefulness of 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 formal analysis techniques with testing. Some of the expected contributions of our study include an experimental assessment framework and empirical data.

In addition to using our assessment framework to conduct experiments, other future work includes extending it to include the ability to optimize test suites and property sets (see Fig. 2). We plan to investigate existing model-based test generation approaches [7] to generate tests from our properties and conversely develop a property generation approach, similar to Daikon [6] and Terracotta [17], that uses tests as input. We also plan to use the assessment results from our framework to reduce test suites and property sets by removing tests and/or properties that are superfluous in terms of the metrics we use.

5. REFERENCES

[1] P. Anderson. CodeSurfer/Path Inspector. In *Proc. of the IEEE Int. Conf. on Software Maintenance (ICSM04)*, 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] T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of Symp. on Principles of Programming Languages (POPL 2002)*, pages 1–3, Jan. 2002.

[4] S.S. Brilliant and J.C. Knight. Empirical research in software engineering: a workshop. *SIGSOFT Software Engineering Notes*, 24(3):44–52, 1999.

[5] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of ICSE'99*, pages 411–420, May 1999.

[6] 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.

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

[8] C.A.R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2), Apr.-Jun. 2003.

[9] Y. Hoskote, T. Kam, P. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. of the Conference on Design Automation (DAC 1999)*, pages 300–305, Jun. 1999.

[10] 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.

[11] N. Juristo, A.M. Moreno, and S. Vegas. Towards building a solid empirical body of knowledge in testing techniques. *SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[12] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proc. of the Work. on the Evaluation of Software Defect Detection Tools*, June 2005.

[13] Robby, M.B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of ESEC/FSE-11*, pages 267–276, Sept. 2003.

[14] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. on Soft. Eng.*, 25(6):401–419, Jun. 1998.

[15] J. Rushby. Disappearing formal methods. In *Proc. of the High-Assurance Systems Engineering Symp. (HASE'00)*, Nov. 2000.

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

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